SIXTEENTH ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

Joint with

ASQ Software Division's

EIGHTH INTERNATIONAL CONFERENCE ON SOFTWARE QUALITY

OCTOBER 13 - 14, 1998

Oregon Convention Center
Portland, Oregon

TABLE OF CONTENTS

Process - CMM—October 14

Process - Models—October 14

Testing—October 14

Preface

Sue Bartlett – PNSQC
Terry Deupree – ASQ Software Division

Welcome to the Sixteenth Annual Pacific Northwest Software Quality Conference and the Eighth International Conference on Software Quality. We believe this conference is a superb example of how great things happen when two talented organizations get together on an idea. In the past year or so, we have seen Windows NT starting to take over in areas once in the UNIX domain. We are also seeing an increased demand in the marketplace for higher quality software in the shrink wrap market. Companies continue the quest for more productivity and less people. This makes our job as software professionals that much more demanding to both produce quality software and to continuously search for better ways to support a more productive environment.

This Conference offers an excellent opportunity to learn from successful practitioners and well-known experts. We are pleased to present these Proceedings, and the CD of the Proceedings, which contain the 32 papers of the technical program. These papers were selected from over 60 abstracts submitted by professionals from all over the globe. In addition to the selected papers, we have invited seven exciting speakers and two special panels to share their expertise and insights.

We are delighted with the opportunity to bring Tom DeMarco as a keynote speaker this year. He has been a mainstay of the software industry from his first book on structured analysis in 1978 which set a standard for programming quality. A personal favorite, *Peopleware: Productive Projects and Teams* brings to light some common sense ideas and some not-so-common sense research reports on how to give a person an opportunity to be more successful by paying attention to their environment. His latest book, *The Deadline: A Novel about Project Management* draws on his abundance of experience running projects, hard learned over the years.

On the second day we bring you Tom Gilb, the author of one of the first books on software metrics. With years on the front line, Tom has published additional books on software engineering management and has been very active in software process debates. He will discuss the merits of the current software process push in relation to his experience in the Software Quality arena. A veteran of the many "Quality Initiatives" over the years, he can provide us with insightful views of the current process wars.

We would like to thank Theresa Hunt and Hilly Alexander, the Program Committee Chairs, for putting in the hard work and many hours it takes to organize the program. Many thanks also to the Program Committee members for their effort in selecting these papers from over 60 abstracts and their tireless reviews of the draft papers. Additional thanks to Linda Westfall, from the ASQ Software Division, and Sandy Raddue, from the PNSQC, who were priceless in their part of working out the agreements between the two organizations and all the details that resulted.

The members of all the committees have volunteered many hours and we thank them for their contributions to the success of this Conference. Feel free to talk to any of the committee members as you see them walking about (they wear ribbons marked "Committee"). We all welcome your suggestions.

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

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Sue Bartlett - PNSQC President/Chair
OrCAD, Inc.

Terry Deupree - ASQ Software Division 8ICSQ Chair
DSC Communications

Sandy Raddue - PNSQC Vice President, ASQ Software Division Liaison
Cypress Semiconductor

Linda Westfall - ASQ Software Division Program Chair
DSC Communications

Dennis Ganoe - PNSQC Secretary
Wacom Technologies

Ray Lischner - PNSQC Treasurer, CD-Rom Production
Tempest Software

Tom Griffin - CD-Rom Production
Auburn University

Rick Clements - Keynote Chair 1998, Publicity Chair
Flir Systems

Mike Kress - Publicity Co-Chair
Boeing Commercial Airplane Group

Theresa Hunt - Program Chair
ECC International

Hilly Alexander - Program Co-Chair
ADP Dealer Services

Laurie Duff - PNSQC Software Excellence Award
ADP Dealer Services

Shauna Gonzales - Birds of a Feather
ImageBuilder Software

Don White - Exhibits Chair
BidTek

Geree Streun - Exhibits Co-Chair
GV Software Solutions, Inc.

Howard Mercier - Workshop Chair
Step Technology

Doug Hamilton - Workshop Co-Chair
Anderson Consulting

CONFERENCE PLANNING COMMITTEE

Rick Biehl
Data-Oriented Quality Solutions

Greg Borchers
Sharp Laboratories

Dan Campo
DSC Communications

Pam Case
Case Consulting Services

Kingsum Chow
Intel Corp.

Carolee Cosgrove
McCain Foods Ltd.

Jayash Dalal
Lucent Technologies

Taz Daughtrey
Software Quality Professional

Cindy Ellis
Advanced Sterilization Products

Galina Golant
Cypress Semiconductor

Bhushan Gupta

Dick Hamlet
Portland State University

Warren Harrison
Portland State University

Dan Hoffman
University of Victoria

Mark Johnson
OrCAD Corp.

Rachel Jordon
Cypress Semiconductor

Bill Junk
University of Idaho

Randy King

Claire Lohr
Lohr Systems

Sue McGrath
SAS Institute Inc.

Patricia McQuaid
Cal Poly State University

Fred Mowle
Purdue University

Lynn Moyers
Phoenix Technologies

Rusty Perkins
Lockheed Martin Corp.

John Pustaver

Eric Schnellman
Boeing

Gayla Suddarth
Tanning Technology

Craig Thomas
Sequent Corp.

Miguel Ulloa
Intel

Lynne Warren
Motorola

Scott Whitmire
Advance Systems Research

THE LEARNING ORGANIZATION

Keynote by Tom DeMarco
The Atlantic Systems Guild Inc.

Learning is to Mind
as
Healing is to Body

- What it means to be a learning organization
- History of organizational learning 1950-1997
- Who learns in a learning organization?
- The key organizational change to facilitate learning

2

MANAGING THE SUPPLY CHAIN

K.2

3

WHAT IT MEANS TO BE A LEARNING ORGANIZATION

First an example:

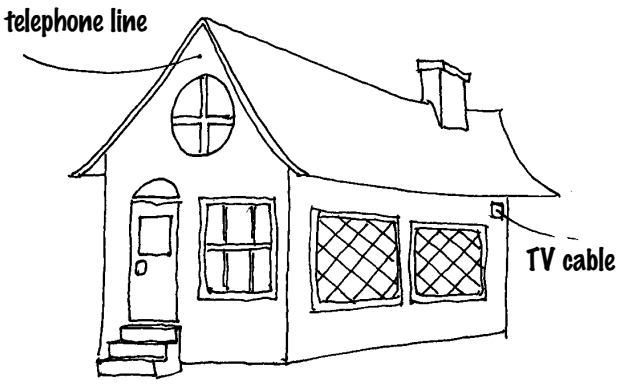
> The on-line bookstore Amazon.com, moves its warehouse to Memphis Tennessee (adjacent to the Federal Express runway).

Observation: virtually all the currently celebrated organizational "learnings" have to do with alternate ways of managing the supply chain.

K.3

4

CONNECTIONS TO YOUR HOUSE



Anything else?

K.4

5

CHARACTERISTICS OF A LEARNING ORGANIZATON

Organizations become known for their learning when they:

- ☆ get out of their boxes
- ☆ rethink the rules
- ☆ redefine themselves

but still manage to . . .

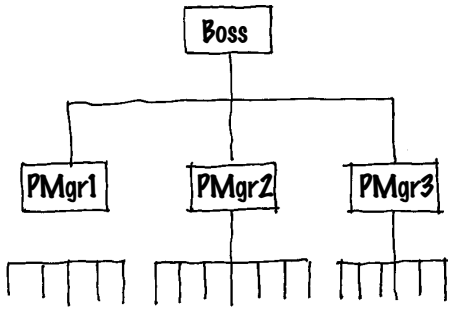
- ☆ stick to the knitting

K.5

6

HISTORY OF ORGANIZATIONAL

From this:



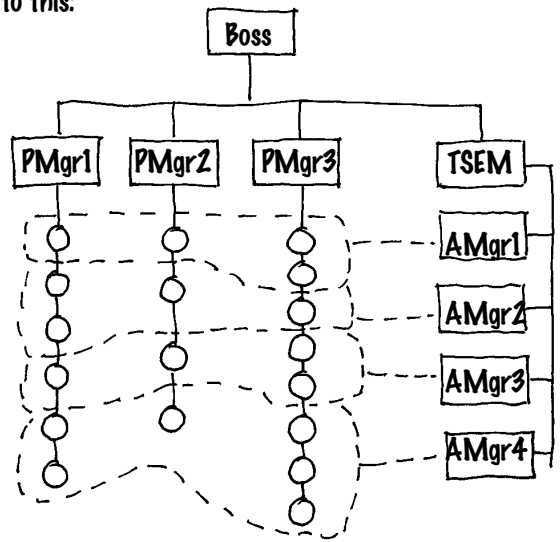
Project-focused organization

K.6

7

LEARNING: 1950 TO THE PRESENT

to this:

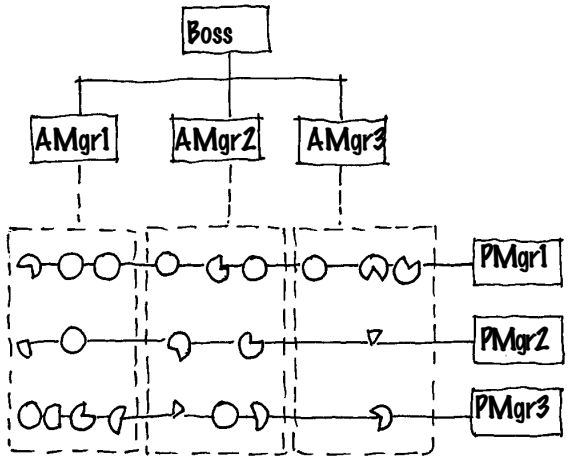


Matrix Management

K.7

8

HISTORY OF ORGANIZATIONAL

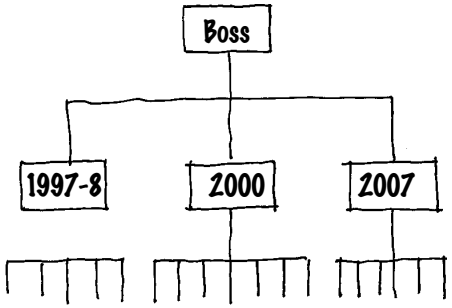


Super Matrix Management

K.8

9

LEARNING (CONTINUED)

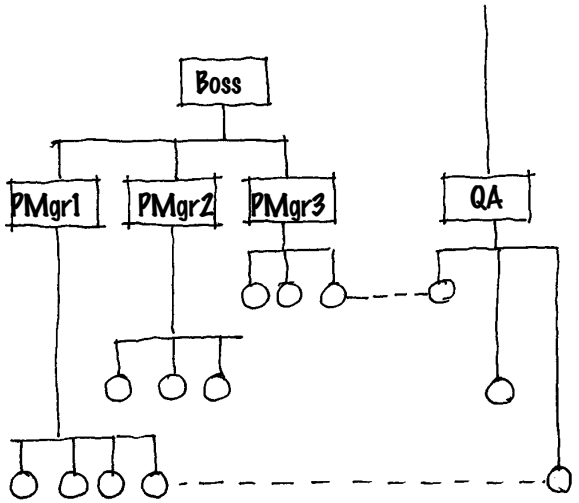


Stratified

K.9

10

HISTORY OF ORGANIZATIONAL

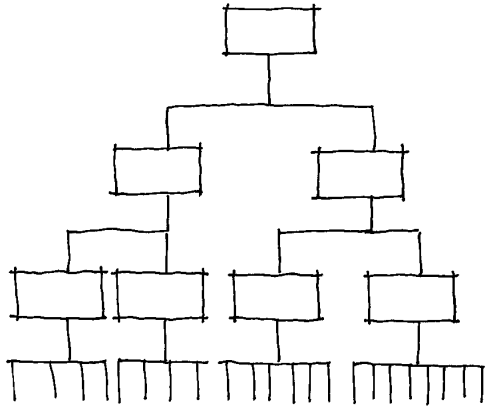


Project Structure plus Q.A.

K.10

11

LEARNING (CONTINUED)

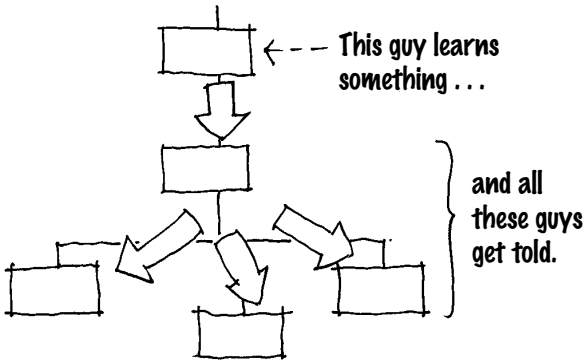


Management Team?

K.11

12

IS THIS HOW ORGANIZATIONAL LEARNING REALLY WORKS?



K.12

13

BASIC HYGIENE:

To make organizational learning possible at all, you have to:

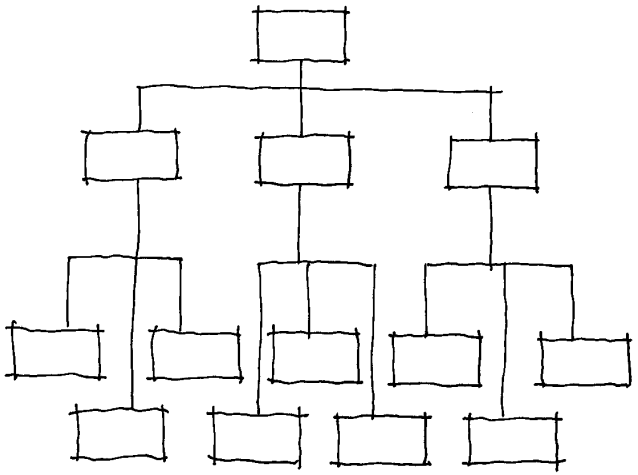
1. keep your people
2. "drive fear out of the organization"*
3. relax control

* W. Edwards Deming

K.13

14

CAN YOU SPOT THE LEARNING CENTER?



K.14

15

WHERE IN THE ORGANIZATION DOES LEARNING TAKE PLACE?

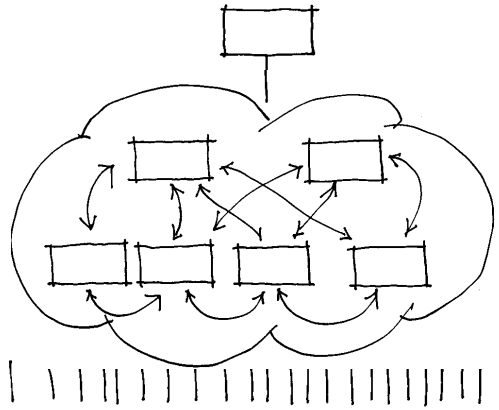
Forget the lines of hierarchy and concentrate on the space between them.

Now . . . tell me about your "management team."

K.15

16

MANAGEMENT TEAM



K.16

17

HOW'S THE OUTLOOK?

Not great.

People in the key learning positions are:

- isolated
- embattled
- frightened
- overworked

or gone . . . .

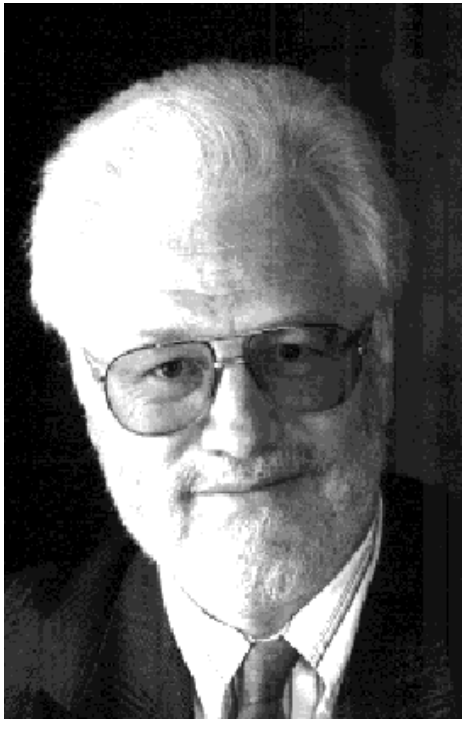
K.17

NOTES:

View Slides

Process Out Quality In

- By Tom Gilb
  - Gilb@acm.org
  - Www.result-planning.com
- 14 Oct 1998
- PNSQC
- Portland, Oregon



CRITICAL CHAIN

Doing Development Faster with Quality

Richard E. Zultner, *Jonah*

ZULTNER & COMPANY

12 Wallingford Drive, Princeton, NJ 08540 USA
phone: +1 (609) 452-0216 fax: +1 (609) 452-2643
richard@zultner.com http://www.zultner.com

Abstract

One of the most common complaints in software development is, "Our projects take too long." Project managers facing aggressive deadlines often see Quality as a hindrance—as something to jettison if at all possible. Can Quality actually reduce project schedules (not indirectly, but) directly—and up front? Yes! By adopting the paradigm shift of Critical Chain, your project schedule can be reduced 15-25% without cutting features, increasing risk, or adding resources. But you must plan in a more sophisticated way, and manage your resources differently…

Keywords: Theory of Constraints (TOC), Critical Chain Project Management, Schedule Reduction, Quality Function Deployment (QFD), Project QFD, Value Identification & Delivery, Statistical Process Control (SPC) for Project Management, Statistical Project Control, Risk Measurement, Risk Reduction

Author

Richard E. Zultner, *CQA, CQE, CSQE, PMP, Jonah*, is an international consultant, educator, author, quality coach, and paradigm guide. His primary focus is applying improvement methods, such as QFD, TOC, and SPC, to software development. Richard trains managers and technical professionals, consults with project teams, and counsels executives in shifting their paradigms. A student of Dr. W. Edwards Deming from 1986-1993, Richard holds a Master's in Management from the J.L. Kellogg Graduate School of Management at Northwestern University, and has professional certifications in Quality, Project Management, Software Engineering, and Theory of Constraints. He is currently working on his Doctorate in Software Quality. He is a 1998 recipient of the Akao Prize (for his work in Quality Function Deployment) –one of twelve people in the world so honored.

A ***Jonah***, which refers to not to a minor 8th-century BC Hebrew prophet, or a book of the Old Testament, but to someone certified in the Theory of Constraints. (Also the name of the consultant-coach in Dr. Goldratt's first book, *The Goal*.) Certification is achieved not by being swallowed by "a great fish" (Jonah 1:17) and then being "vomited out … upon the dry land" (Jonah 2:10), but from extensive training at the Averham Y. Goldratt Institute (http://www.goldratt.com). However, Jonahs *are* supposed to be able to coach "more than a hundred and twenty thousand persons who do not know their right hand from their left" (Jonah 4:11).

Version: 1.60 Revised 15 Aug 98

Introduction

Do your projects take too long? Do you want to reduce your project schedules by 15-25% without cutting features, increasing risk, or adding resources? Are you willing to plan your projects in a more sophisticated way, and manage your resources differently to achieve such schedule reduction?

Over the past five years, a new approach to project management has become best practice: Critical Chain Project Management. Developed by Dr. Eliyahu M. Goldratt, this application of the Theory of Constraints to Project Management produces projects that are 15-25% shorter, without cutting features, increasing risk, or adding resources. So no sacrifice of Quality is required in order to achieve shorter schedules. Or, some of the schedule savings from Critical Chain can be invested in improving quality. No longer is, "I'd like to do that <Quality Method>, but we don't have time on this project," an acceptable excuse for not doing Quality. But before we look at some of the details of Critical Chain, let's make sure we aren't sacrificing the satisfaction of our customers for speed.

Assure Value with Quality Function Deployment (QFD)

Quality Function Deployment is a Quality method developed over the past thirty years by Drs. Yoji Akao and Shigeru Mizuno [Akao 1990, Mizuno & Akao 1994]. QFD, one of the key quality systems of Total Quality Management (TQM), was built to assure value to customers [Zultner 1993]. QFD applies to the software product, the software development process, and the management of software projects [Zultner 1998a]. In recent years, Project QFD has become an increasingly important part of software project management because it provides the tools and techniques to identify and deliver *value* [Zultner 1997].

Value comes First

In software development, the "prime directive" is to build software that customers want. *When* you can deliver it, *how much* it costs, *how* reliable it is—all these are secondary to the basic question, "Will the customers *want* it?" If you cannot develop software that customers want, it doesn't matter how fast, how cheap, or how well it performs—you *failed*. In addition, you cannot compensate for building the wrong software by finishing development early, or at a lower budget, or with great morale on the development team. Your software development project still failed.

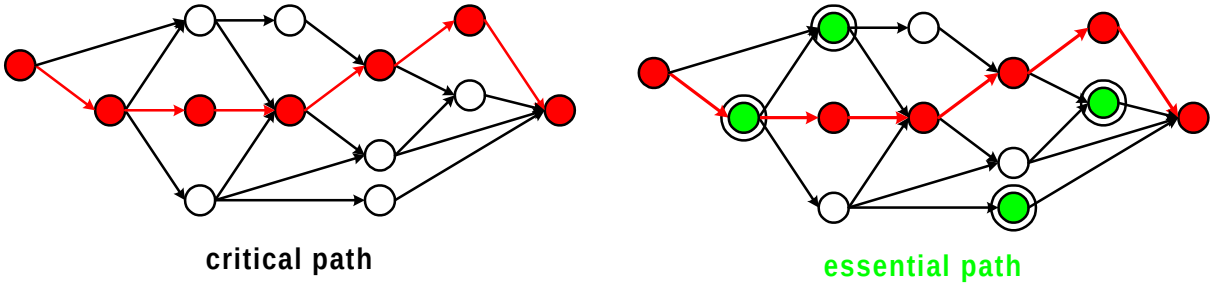


Figure 1. Critical vs. Essential Paths. Critical Path activities are **critical** to the project *schedule*. But even more important are the **essential** activities that provide *value* to customers. The activities that add the most value define the Essential Path. Cutting an essential activity, by moving resources to an activity that is merely critical, is a common mistake in managing software development projects. The Maximum Value table of Project QFD identifies the Essential Path activities.

Yet, in a competitive situation, it is not enough to just build software that your customers *want*. You must build software that your customers want *more* than the offerings of competitors. And enough customers must want your soft-

ware, that you can deliver the **business case** that justified your software project in the first place to the organization. Any project that cannot satisfy its business case is a business failure.

In order to have a good chance of delivering your business case; you must satisfy enough customers. To do this, you must deliver enough value in your software to enough customers—more than your closest competitors. To do that, you need methods to identify and deliver value—with *assurance*.

Identifying Value Fast

QFD is the Quality system for assuring customer satisfaction through the efficient delivery of value to customers. Schedule is important, but value to customers is more important in software development. The presence of value is also more important than the absence of defects. So what is the fastest way to identify and deliver value?

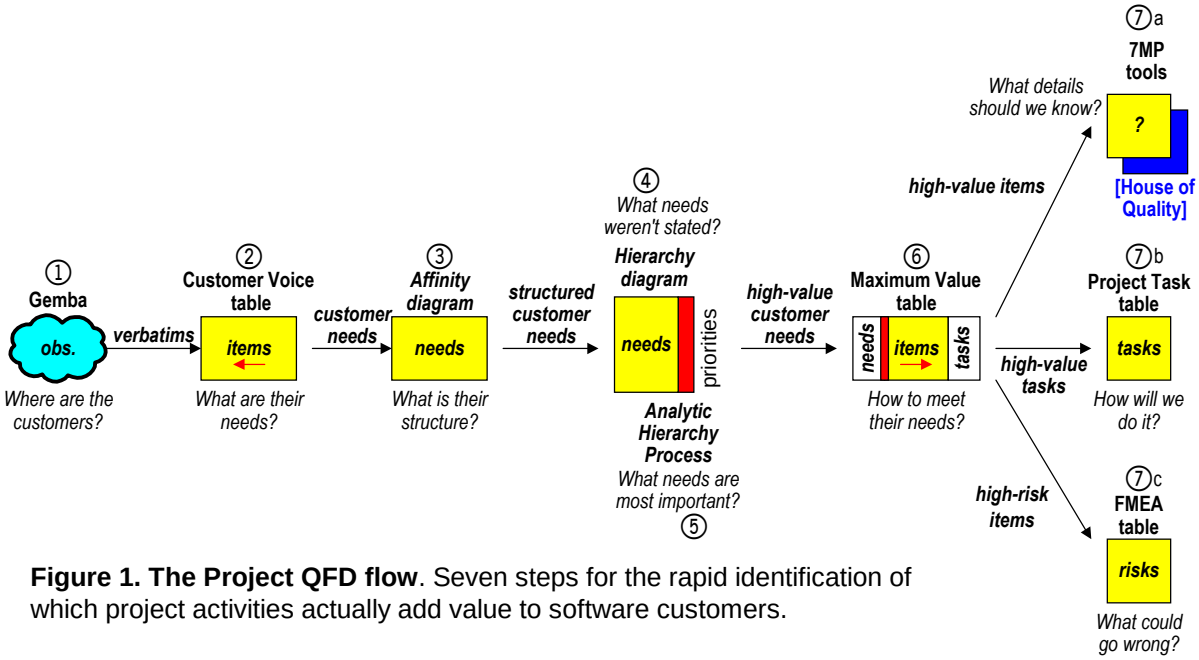


Figure 1. The Project QFD flow. Seven steps for the rapid identification of which project activities actually add value to software customers.

Project QFD is an "essential minimum method" to identify high-value needs, product characteristics, and project tasks, quickly. There are seven basic steps [Zultner 1995]:

① ***Where are the customers?*** Project QFD begins by focusing on the most important customers, and by direct observation of their needs—by going to the *gemba*. This supplements traditional interviewing, and offers a way to obtain in-context details needed to discover unstated requirements.

② ***What are their needs?*** Once focused on the most important customers, we must analyze their statements—their *verbatims*. The Customer Voice table (CVT) is used to understand and analyze what customers say, and why. The result is a list of refined customer needs (only)—identifying what 'value' is for this customer, with this software, on this project. These are *not* product features, but customers' needs.

③ ***What is their structure?*** Once the customers' needs are clear, their structure must be determined. The KJ method is used *with customers* to produce an Affinity Diagram (AD) that shows how *customers* think about *their* needs—the skeleton structure of their requirements.

④ ***What needs weren't stated?*** Using the skeleton structure, a Hierarchy Diagram (HD) is produced, and analyzed for unstated, but structurally implied, requirements. These are confirmed with customers. Now we understand the wants and needs of the customers, and their structure.

⑤ ***What needs are most important?*** The customer needs hierarchy is then quantified and prioritized with customers—top-down, with the Analytic Hierarchy Process (AHP) [Zultner 1992a]. This allows us to identify most of the most important customer needs *first*. Now we have a small number of high-value needs.

⑥ ***How to meet their needs?*** The "vital few" most important needs are then analyzed in depth in the Maximum Value table (MVT). Here, every important, difficult, or risky project activity that is required to deliver each of the high-value needs is identified. The result is the clear identification of those essential project tasks where the high-value needs are delivered—the *maximum value-adding activities* of the development project. This is where the project manager directs "best efforts" to be applied.

⑦ ***What details should we know? How will we do it? What could go wrong?*** Once essential tasks are identified, there will be details that require more scrutiny. The seven Management & Planning (7MP) tools, the tool set of QFD, are used for this purpose. **The House of Quality** matrix is *one instance* of a matrix being used to explore in detail the interaction between (just) two columns (out of many) on the Maximum Value table. For new high-value tasks, the **Project Task table** (PTT) is used to define such tasks in more detail. To identify possible failure modes of high-value tasks, the **Failure Mode Effect Analysis** (FMEA) table is used. This table is not only useful to consider what might go wrong, but also to plan what countermeasures could be employed to prevent, detect, and minimize the project failure modes of high-value development tasks. FMEA is an important tool for risk management: the identification and reduction of project-task-related risk during development.

Project QFD can assure that value is identified and delivered to customers [Zultner 1996]. The Maximum Value table identifies for the project manager the essential tasks where the greatest leverage points for applying Quality Methods are: "where can we apply limited resources to produce the most value for our customers?" (We can also see where the most important points are for defects to be prevented or minimized.)

Identifying value, and focusing our best efforts on high-value tasks, is essential for successful software development [Zultner 1992b]. But sometimes, developing the software quickly is also critical to satisfying our customers. And with Project QFD to identify and deliver value, we can assure that features essential to customer satisfaction are not sacrificed for speed. So with Quality assured, how can we do development faster?

Reduce Schedule with Theory of Constraints (TOC)

The Theory of Constraints, as developed by Dr. Eliyahu M. Goldratt, is the application of the methods of the hard sciences to human systems [McMullen 1998]. At its core, it uses two concepts of the hard sciences that are quite different from their ordinary use: 'complexity', and 'problem'.

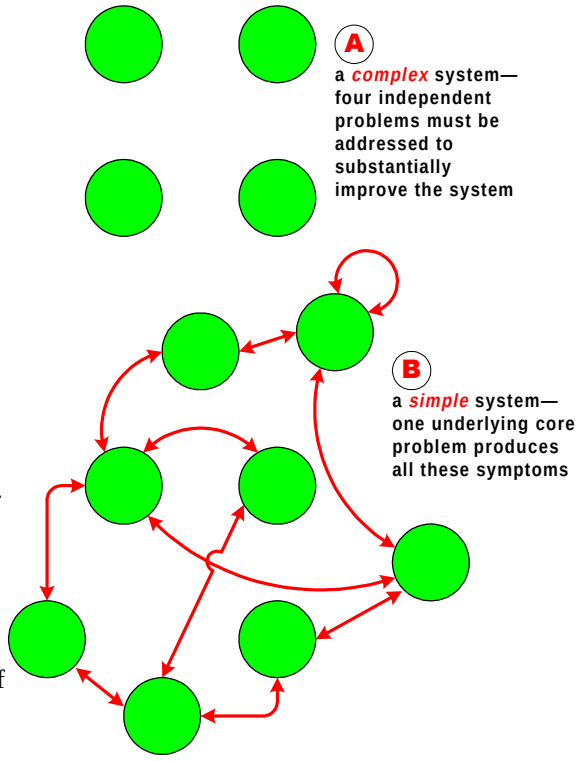
'Complexity' in ordinary language refers to the ***amount of data*** required to describe a system. In the figure below, System 'B' requires many words to describe, so we'd ordinarily say it's "complex."

But 'complexity' in the hard sciences refers to ***degrees of freedom***—the number of points you'd need to interact with, to impact the whole system. System 'A' is "complex", as you must interact with four elements to impact the whole system.

In the hard sciences, a '**problem**' is viewed as a conflict between two necessary conditions. Ordinarily, we seek to "solve" problems by compromise—trading off both conditions to get the "best" result we can. Sometimes this is even called "optimization," as if to suggest that it is not possible to do better. In Theory of Constraints, we "dis-solve" problems by breaking the conflict with a win-win "no compromise" solution. By not compromising, it is possible to get big gains quickly—if you are willing to shift your paradigm…

In order to solve the problem with system A, it is necessary to develop and implement four solutions. Anything less cannot impact the entire system. This will be difficult.

To solve the problem with system B, we will work to discover the ***one core problem*** that causes the numerous symptoms we see. We will uncover the underlying ***core conflict***, and break it. So we only have to develop and implement ***one comprehensive solution***—which will impact the entire system. This is faster and easier.



In analyzing the problems traditionally found in managing software projects, can Theory of Constraints analysis find a single solution that can dramatically reduce elapsed time? Yes. Critical Chain project management achieves significant schedule reduction—even on large projects [Goldratt 1997].

Case Studies

Does the Theory of Constraints solution to the core problem of traditional project management problems, work on real projects? Big projects? Even huge projects?

Harris Semiconductor, Project Raptor was an investment of $250M —the largest capital investment in Harris Corporation history—in a state-of-the-art 8" wafer fabrication plant in Mountaintop, Pennsylvania USA. Management wanted the plant completed in 27 months. (The world record for completion of a wafer fab at the time was 29 months, and the world record for production ramp-up after completion was 16 months—at a different plant from the one holding the completion record.)

The project team came up with a Critical Chain schedule of 18 months. Management instructed the team to "adjust" their schedule to 23 months, as 18 months was just "too ambitious."

The plant was complete in 13 months, with a 4% budget overrun. Usually construction projects of this type require at least 30 months. Ramp-up to full production took 21 additional days. Usually 46-54 months is required to go from groundbreaking to full production. Harris did it in 14 months.

With the plant up and running, Critical Chain was then applied to custom chip development: the total elapsed time was reduced from 8 months to 6 weeks.

Harris is now deploying the Critical Chain approach company-wide [Levinson 1998].

Israeli Aircraft Industry, Maintenance Division. In November of 1995, the Maintenance Division of Israeli Aircraft was in trouble. In 1995 they attracted only 14 aircraft to maintain, had only $12M of orders for 1996, and needed an additional $80M in revenue to break-even—or face massive lay-offs. But as a government-owned firm, and 1996 being an election year, no lay-offs were permitted. So they tried Critical Chain. It worked.

By 1997 they attracted 38 planes, with the same people and facilities as in 1995. Continental Airlines started sending all their widebody jets to IAI for maintenance, even though they don't fly to Israel as a destination. They willingly paid 30% above the going rate for maintenance, because they got their planes back in half the usual time. One-day earlier turn-around saves the cost of renting a replacement plane for a day: $60K. The Maintenance Division is now the biggest profit center in the company. In 1998, business is so good they broke-even for the year in April.

Software Development. As software development projects are typically unique, numbers as to "how much faster" Critical Chain projects are finished are hard to relate to. There are several software development organizations that have completed projects in 15-25% less time than initially planned by using the Critical Chain approach. One mid-sized software company experienced their first-ever ***early*** large software development project. Another reported that for the first time, large amounts of overtime were not required to meet the deadline. Experience to date indicates that software projects experience even greater gains from Critical Chain than hardware projects [Goldratt 1998b].

Shorter Development Schedules

So how are these organizations doing development projects with shorter schedules—without cutting features, increasing risk, or adding resources? They are managing *project variation* in a more sophisticated way, and ***not*** doing "projects as usual" practices that invisibly waste large amounts of time on software projects.

Project Managers of software projects will protest that they, and their teams, are working hard every day to get their software developed as soon as possible. Yet huge amounts of time are wasted by poor practices—such as multi-tasking project resources.

Stop Multi-Tasking. Critical Chain analysis has found that multi-tasking is the single most devastating practice in terms of project schedules [Goldratt 1998a]. Nothing delays projects like multi-tasking. Multi-tasking is so bad it can overwhelm almost any positive project practice. With traditional project management, multi-tasking is… tolerated. With Critical Chain, multi-tasking is *the greatest sin a project manager can commit*. For an example of the tremendous losses due to multi-tasking, please see the Appendix on "Multi-Tasking: What Evil do we do?"

Avoiding bad practices, like multitasking of project resources, is necessary, but we must also manage variation in a more sophisticated, and visible, way. This is the basis for the "guarantee" that Critical Chain makes: to reduce your software project by 15-25% without cutting features, increasing risk, or adding resources.

Manage Variation. Critical Chain focuses on the oldest concept in the field of Quality: variation. The traditional Critical Path Method (CPM) ignores variation—assuming that good and bad luck will "average out" over the duration of the project. The Program Evaluation and Review Technique (PERT) does consider variation, and allows for the calculation of the probability of making your end date on time—but offers no means to *manage* the variation. Critical Chain not only has a more efficient way to *plan* for variation, resulting in shorter schedules, but also the means to *manage* variation—so we have a better metric for project progress, and schedule assurance.

Measuring project progress, and the use of phase-end stage-gates, is done differently with Critical Chain—because of a superior means of managing variation [Zultner 1998b].

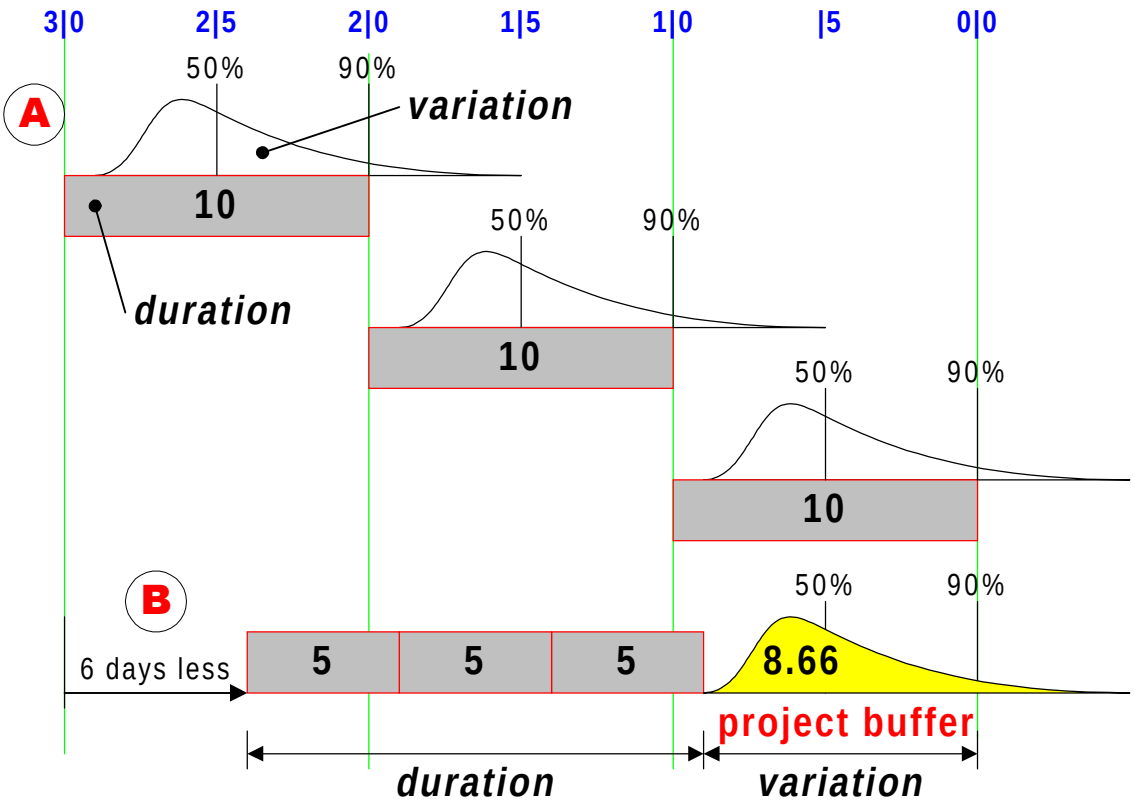


Figure 2. Managing Task Variation Efficiently. In Critical Chain Project Management, variation (risk) is not managed at the *task* level, but at the *project* level—where equivalent protection requires less time (because it much more efficient). The same "leverage" used by insurance companies.

Traditionally, variation (risk) is managed at the task level. The estimate to complete a development task is not 50% likely, but 90% likely. This means the estimate includes protection from variation—**safety**. In the example (A) above, three tasks are estimated to take ten days each (with 90% confidence). The project will take thirty days (and experienced project managers will add additional safety onto that!).

If instead, as in (B) above, the safety is collected in a pool for the entire project—a **project buffer**—a much smaller amount of time can provide equivalent protection. Here a project buffer of 8.66 days provides the same safety as 15 days at the task level. So by managing variation more efficiently, small projects can be done in 15% less time, and large projects in 25% less time.

Calculating Buffer Size. The minimum required buffer size is calculated as follows:

1. On the resource-constrained critical path, for each task, take the difference between the 50% likely task duration estimate and the 90% as two standard deviations of task variability.
2. Divide each by 2 to get the task standard deviations. Square them to get the task variances. Add the task variances to get the project variance.
3. Take the square root to get the project standard deviation, and multiply by 2 to get the 90% likely project buffer. (A beta distribution can be used to estimate other likelihood durations.)

A more conservative rule-of-thumb may be used instead: consider half of all critical path task durations as safety, and set half of that total as the project buffer. So the buffer will be 25-33% of the total project duration.

In the Critical Chain approach, we will protect our projects against risks by establishing the appropriate buffers. Such buffers assure that we can make our deadlines despite "Murphy" (risk) being encountered on our projects. Project buffers, feeding buffers, resource buffers, drum buffers, and other buffers, absorb variability efficiently so our projects can reliably finish on-time, in less time [Zultner 1998b].

There are many interesting details of both the problems with traditional project management, and elements of the single comprehensive solution of Critical Chain for software project management. In practice, the entire "how to" details of Critical Chain for software projects can be learned in two days [Zultner 1998c]. Critical Chain is in many respects a *simpler* approach to project management for software projects than traditional project management.

Implementing Critical Chain for Software Projects

The technical details of Critical Chain for software projects are a straightforward application of an understanding of variation to project planning. The challenge comes in implementing it on a project, and throughout a software development organization. Why is Critical Chain easy to understand, but challenging to implement?

Warning! Paradigm Shift Required! Critical Chain requires a new way of thinking about planning and managing projects. The number of things done differently justify Critical Chain as a true paradigm shift [Zultner 1998c]:

Critical Path Project Management	**Critical Chain Project Management**
The project finish is a date we think we can hit (and then we work like hell to make it).	The project finish is planned with a chosen level of likelihood, and assured with buffers.
The critical path determines the start and end of the project—and the path may change during the project.	The critical path determines the end of the project (after a project buffer is added to it), but the start is often determined by a non-critical activity. The path does not change.
Variation is implicit, and assumed to "average out" over the length of the project.	Variation is explicitly planned and managed throughout the project with buffers.
Variation is managed at the task level (component view).	Variation is managed at the project level (system view).
To keep the project on schedule, we must keep each task on schedule according to the calendar.	To keep the project on schedule, we manage our buffers, which allow us to absorb variation efficiently.
Task starts and finishes are carefully tracked. Schedule "slippage" is important and is closely monitored.	Buffer status is carefully tracked. When any task starts or finishes relative to the calendar is not important.
People are evaluated in terms of whether their tasks are late relative to their committed calendar date for task completion.	Half of all tasks are expected to take longer than planned, and the buffers absorb such variation.
Fixed-date "stage gate" reviews are scheduled to evaluate project progress to date.	Floating "stage gate" reviews are triggered by phase completion, and buffer status is reviewed for project completion likelihood.
The amount of slack that non-critical paths have is not important, and not tracked.	Non-critical paths must have a sufficient "feeding buffers" to protect the critical path.
Making progress on every project, during every reporting period, is important, so resources are multi-tasked.	Multi-tasking of resources is devastating, and is avoided at all costs, including delaying the start of projects.

Table 1. Traditional vs. Critical Chain Project Management. Ten key differences make Critical Chain project management a *paradigm shift* requiring changes in mindset and behavior by the entire project team—and ultimately, the entire software development organization.

It is not easy to persuade software developers to do things very differently from what they have been doing for so long, even though your logic is impeccable, the need is great, and the benefits are big. Even examples from other "benchmark" software organizations are not sufficient to get people to change their paradigms.

Software developers raise objections to new ideas in a set pattern: the Layers of Resistance. The implementation of Critical Chain for software projects must be carefully structured to move people through all five layers successfully and efficiently. Only then will implementation be successful, and sustainable.

Layers of Resistance. Here is a summary of the layers [Goldratt 1998b]. For each layer, Critical Chain contains specific tools and techniques used to efficiently work through the resistance, and achieve buy-in to the solution.

- **Layer 1:** "That's not our real problem"
- **Layer 2:** "That's not the right direction to look for a for a solution"
- **Layer 3:** "That's not a solution"
- **Layer 4:** "There are negative effects with that solution"
- **Layer 5:** "There are obstacles to implementing that solution"
- **Layer 6:** Unverbalized fear

The Layers of Resistance represent defense-in-depth against bad ideas—so they won't harm the organization. "No resistance" to new ideas is not good, for people, or for organizations. Good ideas will (eventually) make it through the resistance—but we can accelerate the process (from years to weeks).

Implementation Program. Implementing Critical Chain for software projects requires overcoming each Layer of Resistance in sequence. A "typical" implementation of Critical Chain on a project requires a total of five to ten days of training and facilitation [Zultner 1998c]. Only two days of training is needed to teach the team "how to" do Critical Chain planning—the rest of the time is spent working through the Layers of Resistance with the team and their stakeholders. ***All key players*** must be intellectually and emotionally comfortable with the approach and its implications *before* the Critical Chain schedule is put into action.

To date, the Critical Chain approach has accumulated a most impressive track record in dozens of organizations [Newbold 1998]. This is due primarily to a very careful and through implementation process. An understanding of the layers of resistance is an essential part of successful implementation of the Critical Chain for software projects.

A more sophisticated approach to planning and managing software development projects, Critical Chain has consistently reduced project schedules by 15-25% without cutting features, increasing risk, or adding resources to development projects. Critical Chain is a paradigm shift for project managers, and software development organizations. Some people, and organizations, are already pioneering this paradigm shift. Would you like to join them? (Or will you let your competitors go first…?)

The application of Theory of Constraints to project management gives us a new way to plan and manage software development projects: Critical Chain. In a similar fashion, Theory of Constraints can also be applied to software maintenance: Drum-Buffer-Rope [Goldratt 1992, 1994]. In multi-project software development environments, both Critical Chain and Drum-Buffer-Rope approaches are applicable [Goldratt 1998b, 1998a].

Once Project QFD is used to assure the value of our software to our customers, and Critical Chain is used to reduce our project schedule, are there any further gains to be made on our software projects?

Reduce Variation with Statistical Process Control (SPC)

Critical Chain makes variation visible on our software development projects, and gives us the means to plan for it, and absorb it when it occurs. But we can do more. We can actively *reduce* the variation of our project tasks—which will reduce the riskiness of our projects, and allow us to use smaller buffers for protection.

Variation = Risk

All real-world processes have "noise"—that is, they exhibit variation. Such variation can be *measured* using the oldest and most basic tool of quality—the **control chart** [Wheeler 1993, 1992, and 1995]. When examined in this way, all software development tasks are processes, and can determined to be either in-control (stable) or out-of-control (unstable). For software, the individual (x) and moving range (mR) charts are best [Zultner 1994]. Unstable development tasks are inherently risky. SPC methods, such as the **Quality Improvement (QI) Story** [Kume 1985] can be used to *stabilize* out-of-control development processes, and further, to *reduce* their variation—and often reduce their duration as well. How would this work on actual software development projects?

Software Case Study

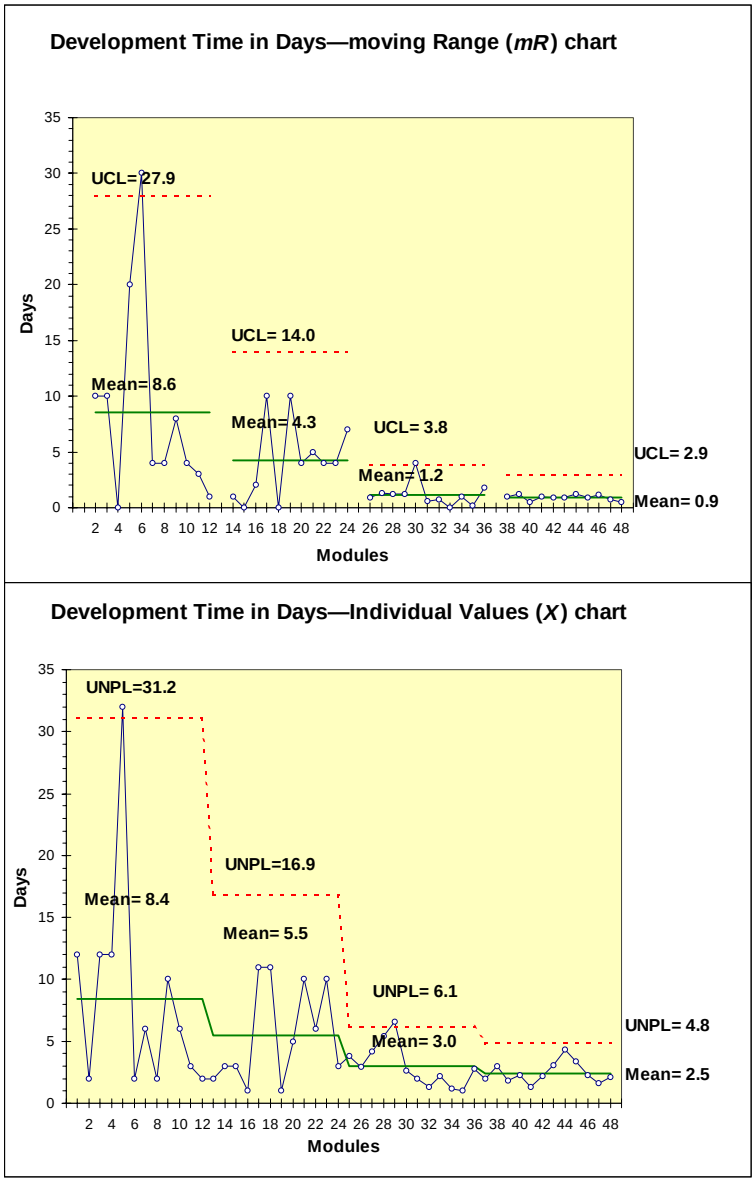
A software development project team had reached the activity of module development. For them, this included module design, coding, and unit test [Zultner 1994].

As seen in the first section of the control charts, the variation (shown on the moving range chart) was averaging 8.6 days, and the duration (shown on the individual chart) was averaging 8.4 days. Further, the work was out-of-control. The project manager saw this on the control charts, and led their team in applying the QI Story to try and get the work under control, and then reduce the variation (risk).

In the second section, the team was able to remove the special cause, and stabilize the work with an average variability of 4.3 days, and an average duration of 5.5 days. Now the real improvement could begin.

In the third section, the results of the QI Story can be seen. The team made five specific changes to their module development process, and as a result, the average variability dropped to 1.2 days, and the average duration dropped to 3.0 days.

This was a significant improvement, but as is common, such changes put the process



out of control. When the standardization step of the QI Story was applied, the process stabilized at an average variability of 0.9 days, and an average duration of 2.5 days. Standardization is an often-overlooked step in software process improvement.

The project team performed these improvements themselves while developing their first 48 modules, and the gains were maintained over the next (and last) 52 modules thanks to the QI Story standardization methods. The team was able to significantly reduce the risk of a module development (as measured on the ***mR*** chart) and decrease the possibility of missing the deadline. With less risk, this activity now requires less safety in the project buffer, reducing the size of the minimum required buffer, and thus shortening the project further. Their improvements in risk reduction also reduced the average module development duration (as measured by the ***x*** chart), directly reducing the total elapsed time of this activity, and thus reducing the overall project schedule. Finally, the team, the only resources for improvement the project manager had, made these improvements *while doing their development work.*

In this way, SPC can be used to *measure* variability, to *protect* the Critical Chain project plan with methods to *recover* from unforeseen problems during the project, and to *reduce* and *stabilize* variability—without the use of more resources or impacting the project schedule.

When SPC is applied in the context of project management to improve how project tasks are done—to reduce risk (and thereby often reducing task duration)—this is SPC for Project Management: **Statistical Project Control**. For many years project managers of software projects have complained about the riskiness of their endeavors. Now they have a way to measure risk, and reduce it.

Conclusion

To do software development projects faster with Quality, we must assure that we *satisfy* our customers. **Project QFD** can identify where we can add the most value to our customers, so we can concentrate our limited resources at those points to generate maximum value in our software. This assures Quality.

To do software development projects faster with Quality, we must plan projects *better*. **Critical Chain** Project Management can give us project schedules 15-25% shorter by more efficiently managing protection from risk—if we are willing to change our paradigm…and stop multi-tasking resources. This reduces schedules significantly.

To do software development projects faster with Quality, we must develop with less *risk*. **Statistical Project Control**, using the **Quality Improvement Story**, **control charts**, and **standardization**, can safely reduce the variability, and often the duration, of high-value or high-risk project tasks. This reduces risk.

Collectively, these methods offer us a synergistic approach to doing development faster and better in substantially less time *without* cutting features, increasing risk, or adding resources.

References

Akao, Yoji, Ed. 1990 [1988]. *Quality Function Deployment: Integrating Customer Requirements into Product Design*. Foreword by Bob King. Cambridge, MA: Productivity Press. ISBN 0-915299-41-0.

Goldratt, Eliyahu M., and Jeff Cox. 1992. *The Goal: A Process of Ongoing Improvement*. 2nd rev. ed. Croton-on-Hudson, NY: North River Press. ISBN 0-88427-061-0.

Goldratt, Eliyahu M. 1994. *It's Not Luck*. Croton-on-Hudson, NY: North River Press. ISBN 0-88427-115-3.

Goldratt, Eliyahu M. 1997. *Critical Chain*. Great Barrington, MA: North River Press. ISBN 0-88427-153-6.

Goldratt, Eliyahu M. 1998a. *Project Management the TOC Way*. Great Barrington, MA: North River Press. ISBN 0-88427-157-9.

Goldratt, Eliyahu M. 1998b. *Reinventing Project Management* Seminar. Held in Harrogate, United Kingdom on 20-21 May 1998. Harrogate, UK: Harrogate Management Centre.

Kume, Hitoshi. 1985. *Statistical Methods for Quality Improvement*. Translated by John Loftus. Tokyo: The Association for Overseas Technical Scholarship. ISBN 4-906224-34-2.

Levinson, William A., Ed. 1998. *Leading the Way to Competitive Excellence: The Harris Mountaintop Case Study*. Milwaukee, WI: ASQ Quality Press. ISBN 0-87389-376X.

Newbold, Robert C. 1998. *Project Management in the Fast Lane: Applying the Theory of Constraints*. Foreword by Thomas B. McMullen, Jr. Boca Raton, FL: St. Lucie Press. ISBN 1-57444-195-7.

McMullen, Thomas B., Jr. 1998. *Introduction to the Theory of Constraints (TOC) Management System*. Boca Raton, FL: St. Lucie Press. ISBN 1-57444-066-7.

Mizuno, Shigeru, and Yoji Akao, Ed. 1994 [1978]. *Quality Function Deployment: The Customer-Driven Approach to Quality Planning and Deployment*. Rev. ed. Tokyo: Asian Productivity Organization. ISBN 92-833-1122-1.

Wheeler, Donald J. 1993. *Understanding Variation: The Key to Managing Chaos*. Knoxville, TN: SPC Press. ISBN 0-945320-35-3.

Wheeler, Donald J., and David S. Chambers. 1992. *Understanding Statistical Process Control*. 2nd ed. Foreword by W. Edwards Deming. Knoxville, TN: SPC Press. ISBN 0-945320-13-2.

Wheeler, Donald J. 1995. *Advanced Topics in Statistical Process Control*. Knoxville, TN: SPC Press. ISBN 0-945320-45-0.

Zultner, Richard E. 1992a. Quality Function Deployment (QFD) for Software. *Total Quality Management for Software*. New York: Van Nostrand Reinhold. Edited by G. Gordon Schulmeyer, and James I. McManus. 297-319. ISBN 0-442-00794-9.

Zultner, Richard E. 1992b. Software QFD: Satisfying Customers. *American Programmer* 5 (February): 12-22.

Zultner, Richard E. 1993. TQM for Technical Teams. *Communications of the ACM* 10 (October): 79-91.

Zultner, Richard E. 1994. Software SPC: What do our Metrics Mean? In *4th International Conference on Software Quality Proceedings*. Held in Washington, DC on 3-5 October 1994, Washington, DC: ASQ Software Division.

Zultner, Richard E. 1995. Blitz QFD: Better, Faster, and Cheaper Forms of QFD. *American Programmer* 8 (October): 24-36.

Zultner, Richard E. 1996. Blitz QFD for Software: A Next Generation Approach for Delivering Value. *In 7th International Conference on Software Quality Tutorials*. Held in Ottawa, Ontario on 28-30 October 1996, Milwaukee, WI: ASQ Software Division.

Zultner, Richard E. 1997. Project QFD: Blitz QFD for Project Managers. In *Transactions from the 9th Symposium on QFD*. Held in Novi, MI on 9-11 June 1997, Ann Arbor, MI: QFD Institute.

Zultner, Richard E. 1998a. QFD and Designing Software. *The QFD Handbook*. New York: John Wiley. 163-184. ISBN 0-471-17381-9. Also Blitz QFD on pages 316-323.

Zultner, Richard E. 1998b. QFD Schedule Deployment: Doing Development Faster with QFD. *In Transactions from the 10th Symposium on QFD*. Held in Novi, MI on 14-17 June 1998, Ann Arbor, MI: QFD Institute.

Zultner, Richard E. 1998c. *Critical Chain for Software Projects* Seminar Materials. Princeton, NJ: ZULTNER & COMPANY.

Multi-Tasking

What evil do we do?

Richard E. Zultner, *Jonah*

ZULTNER & COMPANY
Princeton, NJ USA

Richard Zultner is a certified *Jonah* in Theory of Constraints (granted by the Avraham Y. Goldratt Institute—established by Dr. Eli Goldratt for the advancement of the Theory of Constraints.)

Richard works with software and high-tech organizations as a "Paradigm Guide" to pioneer paradigm-shift methods, such as Critical Chain.

Critical Chain is a more sophisticated way to plan and manage projects. By explicitly managing variation (risk), you can reduce your project schedule by 15-25% with no increase in risk or resources—*if* you are willing to change the way you plan and manage your project…

With Critical Chain, we gain a clear view of risk, and the practice of multi-tasking can truly be seen for the evil it is.

A very readable account, in business novel format, is: Goldratt, Eliyahu M. 1997. *Critical Chain*. Great Barrington, MA: North River Press. ISBN 0-88427-153-6.

Note: This method is also known as "Statistical Project Control" and "QFD Schedule Reduction."

Presentations of this material at your site are available.

Multi-tasking

The practice of assigning one person concurrently to two or more tasks

Resources will be full utilized!

Is this an effective way to manage projects?

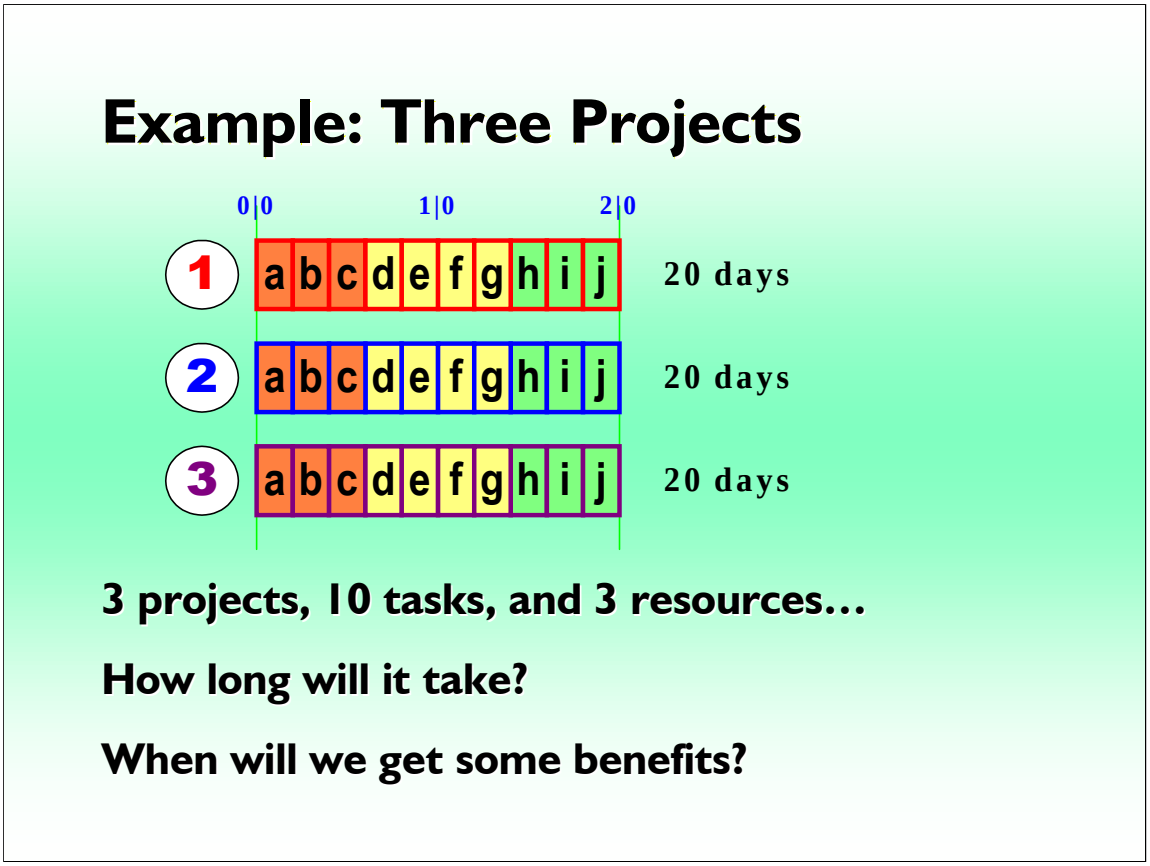
- **Is there an alternative?**

Multitasking is also known as fractional head-count.

This will maximize the utilization of the resources. This is an effective way to use resources, isn't it?

This a good way to manage projects, isn't it?

(This is how we always do it—what alternative is there? Surely this can't be wrong…or worse…)



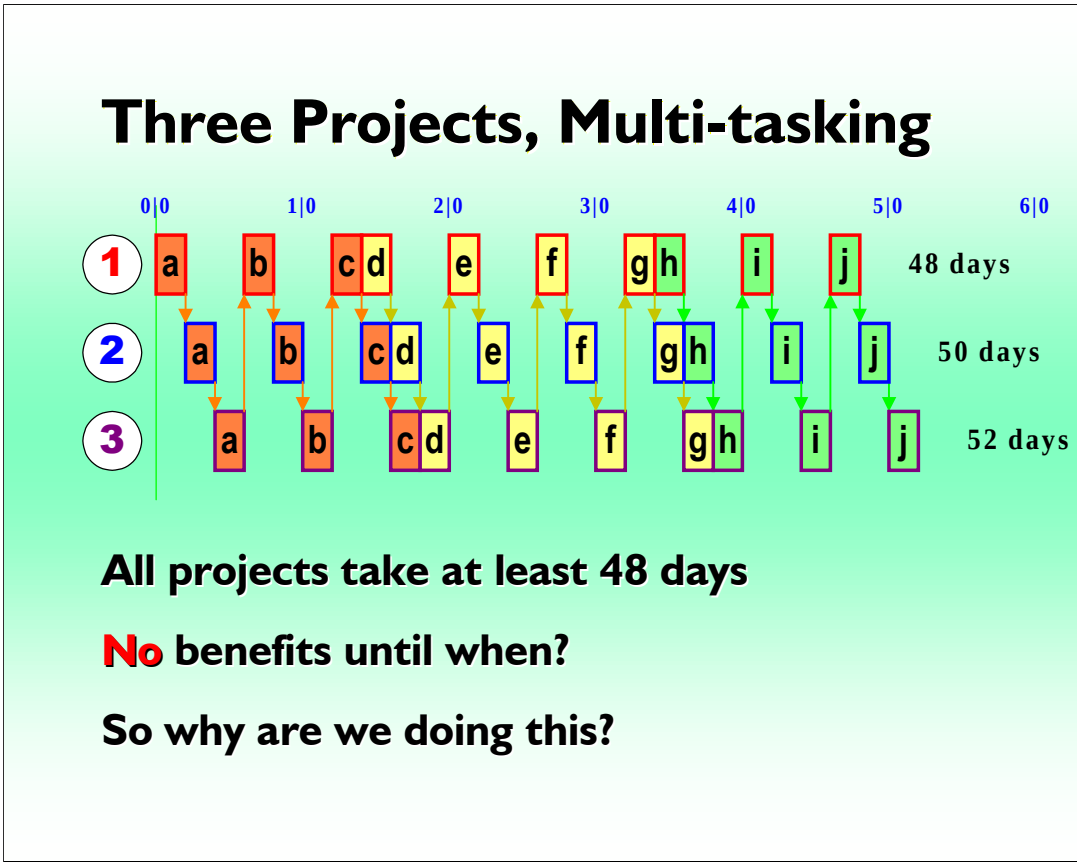
Three Projects, Dedicated Resources. What's wrong with multitasking? Let's see.

Above, are three identical projects (1, 2, and 3), each with ten two-day tasks (a through j). There are three types of resources required:
the **first resource** for Analysis tasks **a-c**,
the **second resource** for Design tasks **d-g,**
and the **third resource** for Development tasks **h-j**.

If we had three of *each* type of resource available, then we could dedicate one of each type, to each project. Then no project would be dependent on another. ***All*** projects would be done in 20 days. ***All*** the benefits from all the projects would arrive in 20 days.

But we have only **one** of each resource—how should we proceed?

(If three project managers each see one-half a person in their resource plans, then 1/2 +1/2 + 1/2 = 1 person will not happen. Tasks will take longer than if a dedicated resource performed them.)



Multi-task all resources. Project Managers for projects 1, 2, and 3 conscientiously insist that some progress be made on their projects by every resource, every week. So every resource switches after every task to another project, so they will have some "progress" to report every week. The toll on the resources from such "project switching" is demanding.

The result is maximum resource utilization. All resources are working all the time. (So what?) All projects are making progress. (Are they?)

All projects take at least 48 days.

No benefits arrive before 48 days. (That's a long time to wait.)

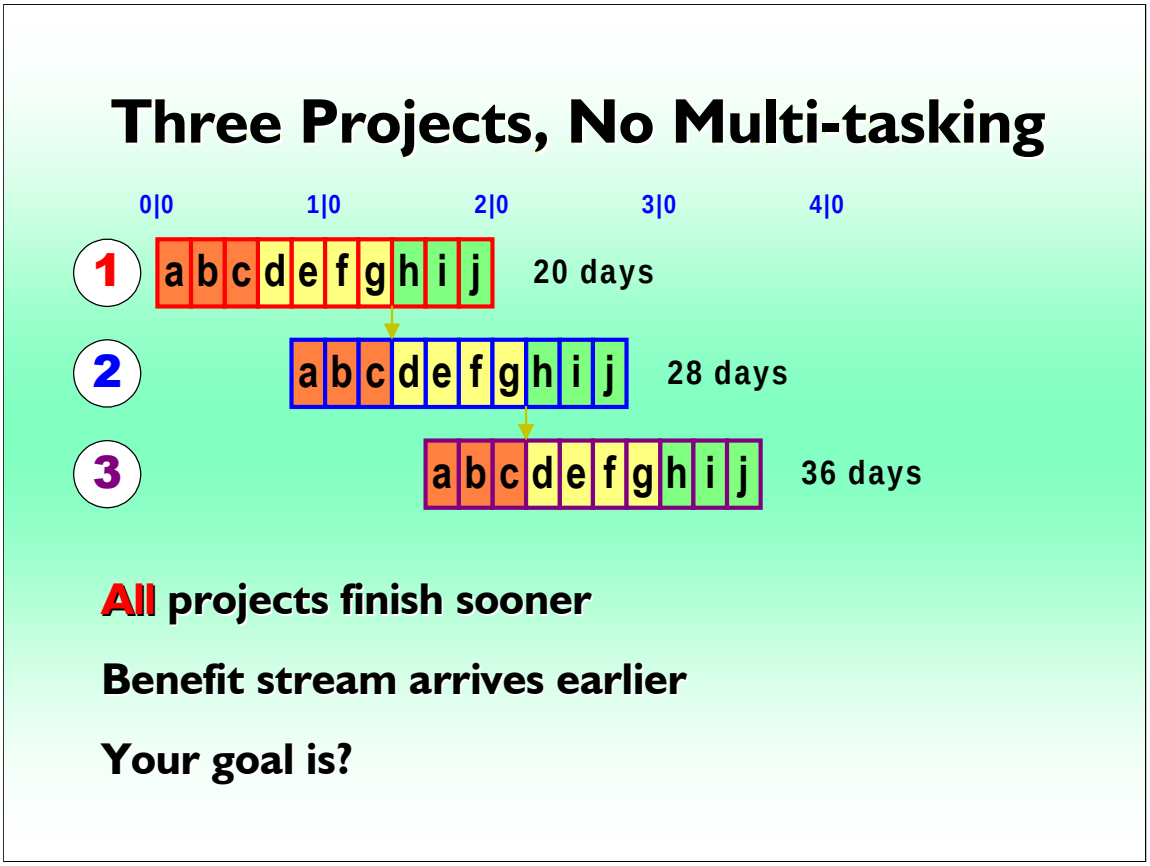
Is maximum resource utilization efficient? (No!)

What is our goal?

In a competitive environment, with tight market windows, this organization is not competitive, despite heroic efforts on every project. Resources are burned out, and turn-over is high.

Everyone loses.

There must be a better way...



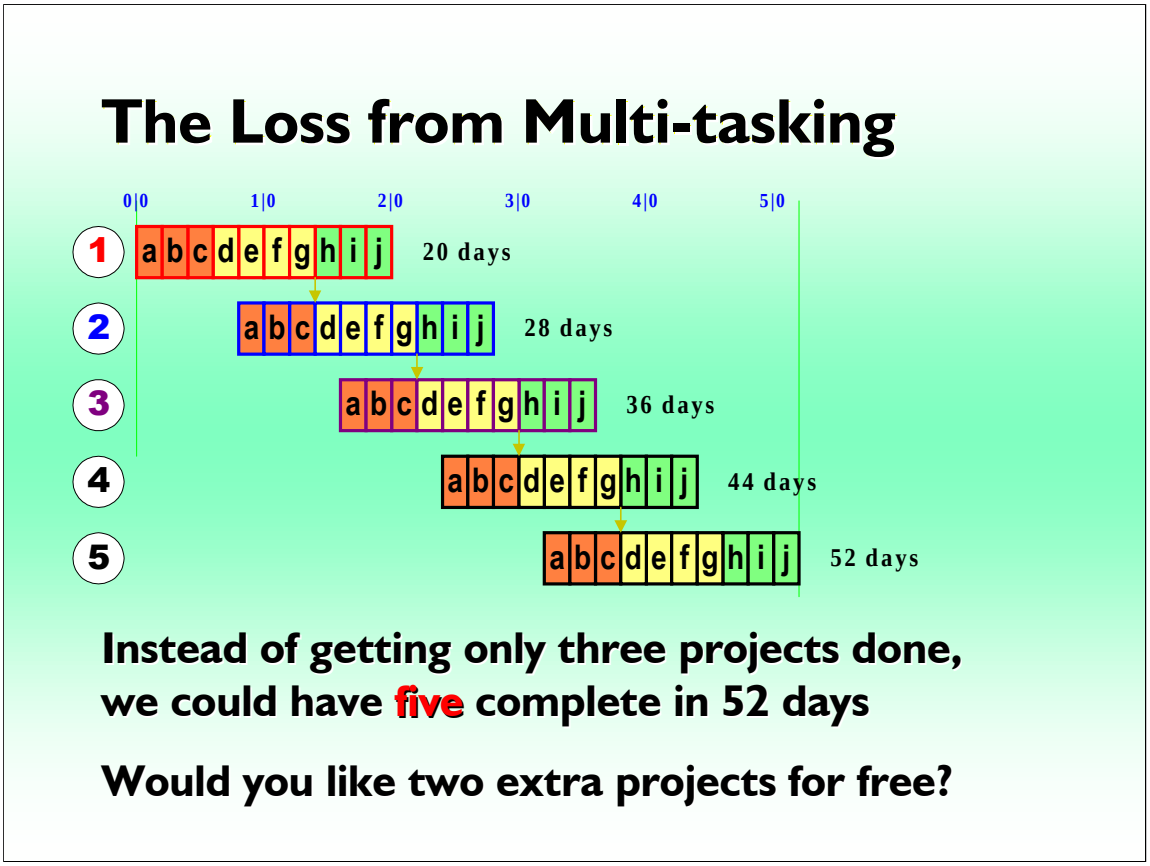
No resources are multi-tasked. By having each resource complete all tasks on one project before going on to another project, we can avoid multitasking. Every task is finished in the time estimated for a dedicated resource to complete it. No resources must switch from one project to another, and back—and then try to pick up where they left off. No resource are overloaded. This is how resources prefer to work.

All projects finish sooner. Even the third project, which waits 16 days to start, finishes earlier than with multi-tasking. ***All projects are better off***.

The "efficiency" of multi-tasking is a myth. Just because a resource is utilized, does not mean it is productive (or genuinely efficient). Here we have lower resource utilization. Only the second resource is working all the time. The first and third resources have a break of two days between each project. Yet we achieve higher throughput. Benefits arrive in 20 days, 28 days, and 36 days, a significantly sooner delivery.

Everyone wins.

Do you want high resource utilization, or shorter projects?



What is the cost of multi-tasking? By not multitasking, we can accomplish more, even though we have lower resource utilization. Resource utilization and productivity are not identical.

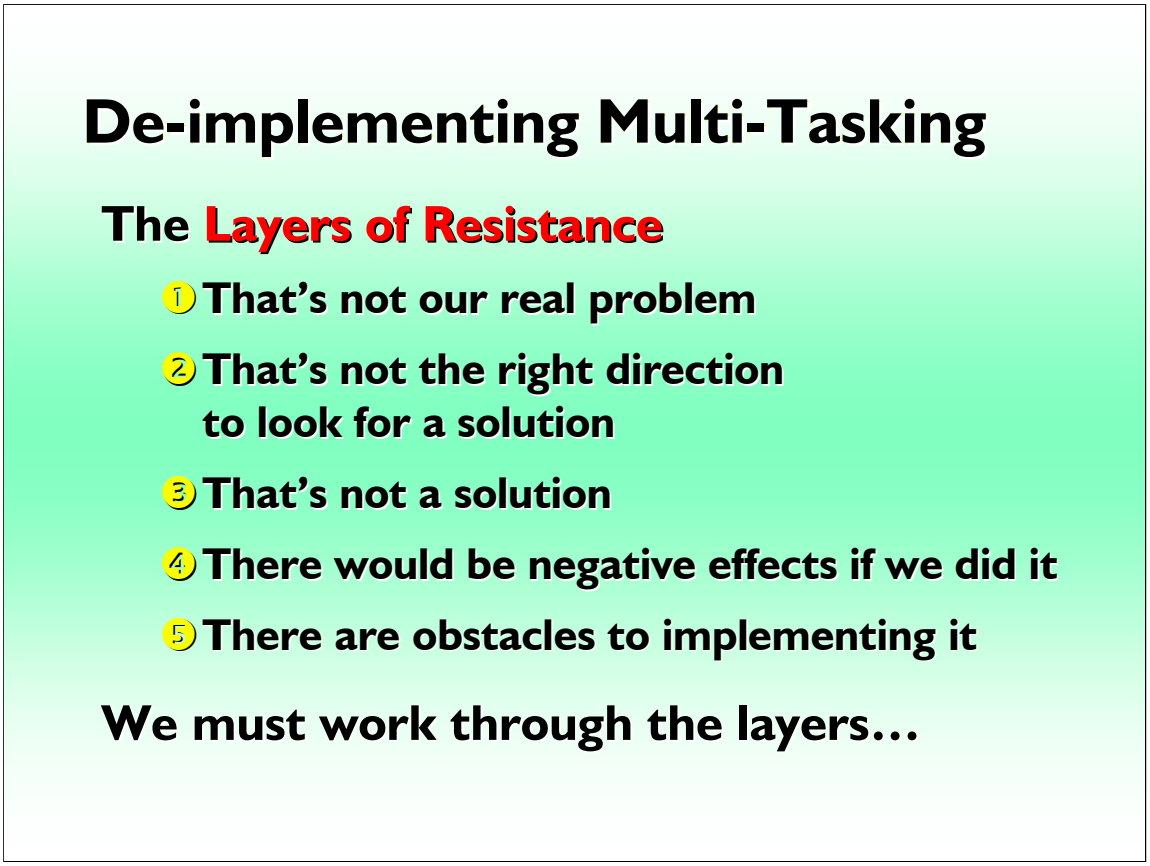
More projects complete in the same time. With multitasking, we can do three projects in 52 days. Without multitasking, we could complete five projects in 52 days—**at no additional cost**. Which do you think is more productive?

We get two-thirds more throughput by not multi-tasking. We complete the three planned projects in two-thirds the time by not multi-tasking. And resources get a break between projects.

Everyone wins.

Do you want high resource utilization, or more projects?

Multi-tasking is devastating to project performance.

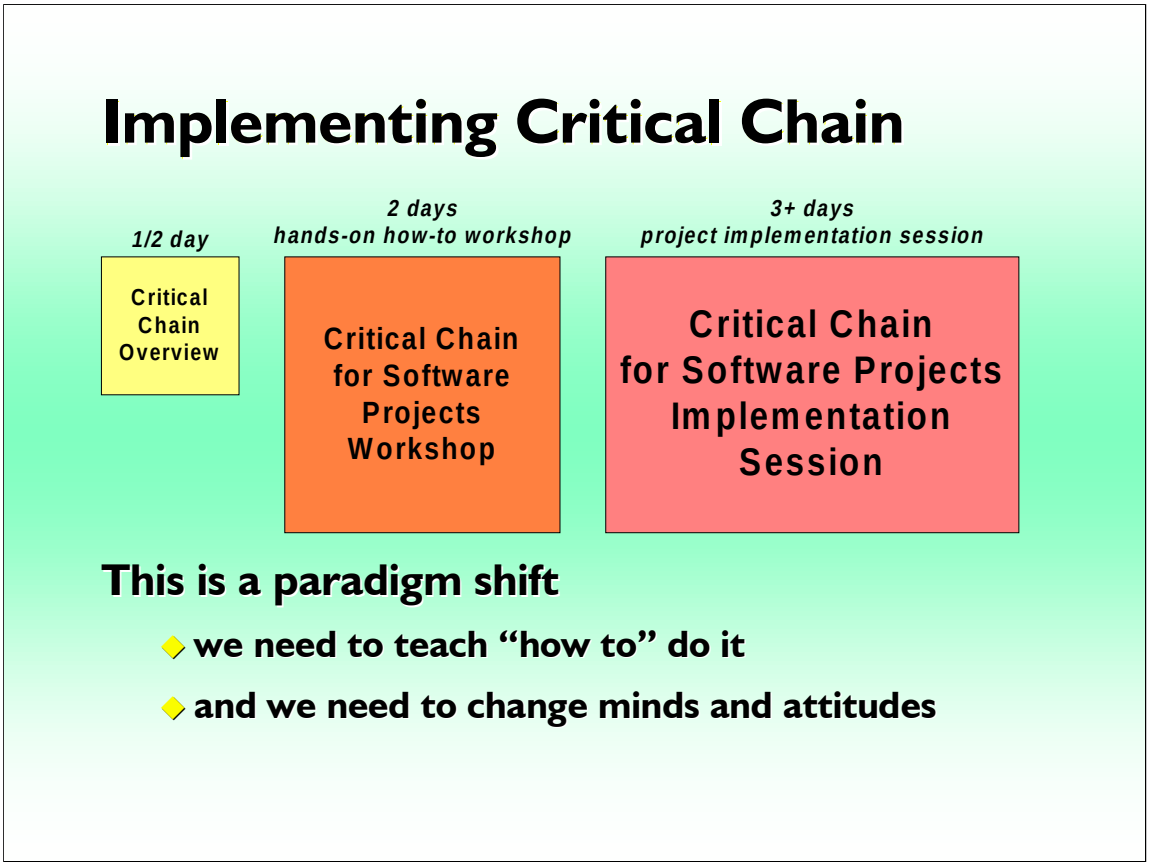


De-implementing Multi-tasking is not a technical problem—it is simpler to plan *not* to multi-task. But this requires a ***paradigm shift***—a change in thought and behavior. For any significant change in managing projects, there will be resistance. To work through the resistance effectively, we must understand the ***layers of resistance***.

Remember, the layers of resistance represent defense-in-depth against bad ideas—so they won't harm the organization. "No resistance" to new ideas is not good, for people, or organizations. Good ideas will (eventually) make it through the resistance—but we can accelerate the process. A Paradigm Guide can help you work through the resistance efficiently.

The **Critical Chain Implementation Program** includes a process for implementing the approach by overcoming the layers of resistance.

Critical Chain is a paradigm shift for project managers, and development organizations. Some people, and organizations, are already pioneering this paradigm shift. Would you like to join them?



The Critical Chain Implementation Program, especially with the de-implementation of multi-tasking, requires learning "how to" plan projects with the critical chain approach. And how to change to the new paradigm.

A half-day **Overview** is available to explain in more detail what problems result from multi-tasking and other traditional sins of project management.

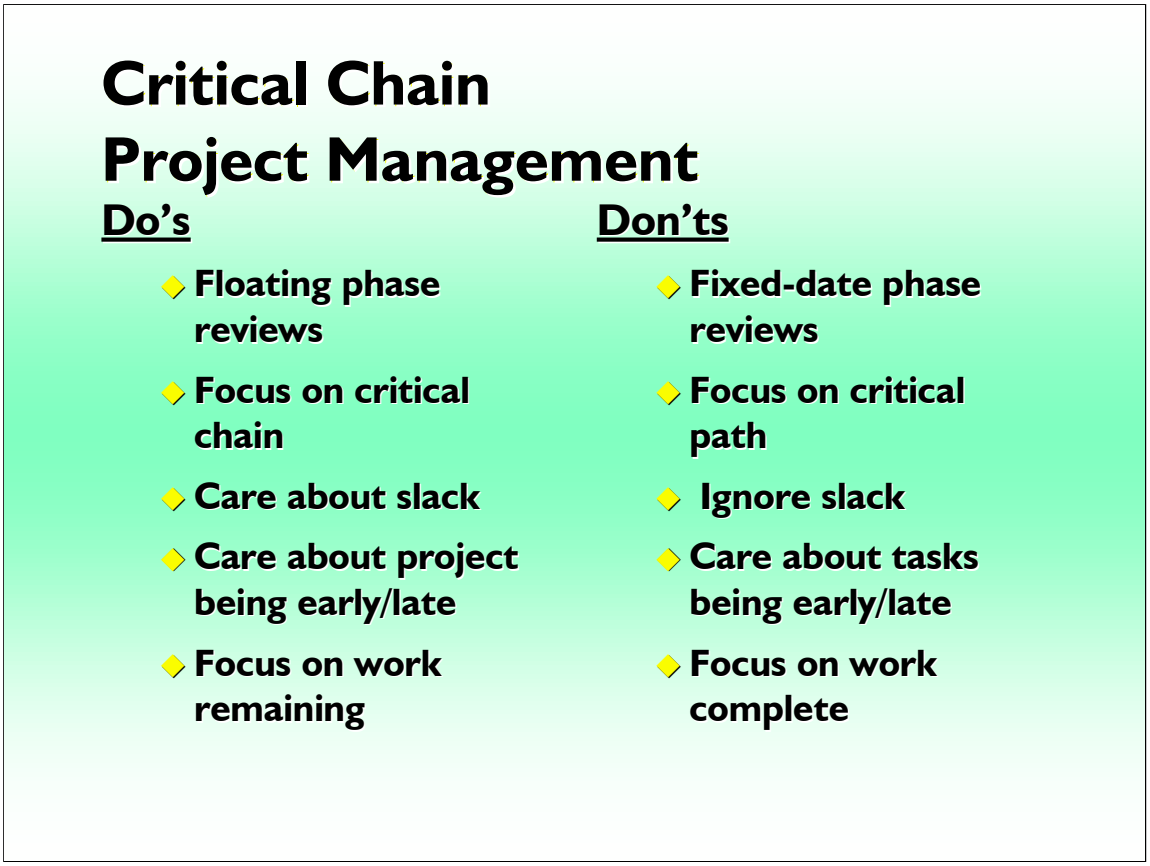
The two-day **Workshop** includes a variety of hands-on exercises. Participants gain not only an intellectual understanding of the critical chain approach, but a "gut-level" understanding. This is the prerequisite to behavior change.

To actually change, requires working through the layers of resistance project-by-project. The project team and all key project players work through the issues and obstacles of replanning their project with critical chain methods in an **Implementation Session**.

The three days (or more, if required) must be spent to get everyone fully "on board" for the change in plans, and in behavior. Follow-up sessions to deal with any difficulties that arise are included in the implementation program.

To date, critical chain has an excellent implementation record. That's due to the thoroughness of this **Implementation Program**. Can we help you too?

—Richard Zultner +1 (609) 452-0216 richard@zultner.com



Benefits of Critical Chain Project Management. Besides multi-tasking, Critical Chain offers a number of insights for managing your projects better.

The primary benefit is a **reduction of overall project duration of 15-25%** *without* increasing risk, adding resources, or cutting deliverables. This is accomplished by explicitly identifying, planning, and managing schedule variation (risk) in a more sophisticated (and more efficient) way.

This leads to a number of non-traditional results.

❑ The slack time of non-critical path activities turns out to be important, and many projects don't have enough slack for their non-critical path activities.

❑ Project starts may be driven by the start of non-critical path activities (the end is still determined by critical path activities, but not the start).

❑ Many projects that feel rushed and lacking in time to do their work often have more time than they need—but they are wasting significant amounts of time—and it is invisible to them. (Such as the time wasted by multi-tasking!)

❑ A simple way to check on project status, and the status of multiple projects in a program.

And more…

Critical Chain Success Story

Harris Corporation

- **Microelectronics plant, Mountaintop, PA**
  - **Goal: operational in 27 months**
- **Critical Chain Schedule: 18 months**
  - **Industry norm: 28-30 months**
- **Results: 14 months, 4% over budget, at full production in 21 days**
  - **Industry norm: 46-54 months total**

Case Study: Harris Corporation's Mountaintop, PA microelectronics plant was a $250M investment—-the largest in Harris Corporation history.

Management wanted the plant operational in 27 months.

The project team came up with a critical chain schedule of 18 months. Management instructed the team to "adjust" the schedule to 23 months, as 18 months was just "too ambitious."

The project was finished in 13 months, with a 4% budget overrun. Usually construction projects of this type require 28-30 months.

The plant was fully operational 21 days after construction was complete. Usually 46-54 months is required to go from groundbreaking to full production. Harris did it in 14 months.

Harris is now deploying the critical chain approach company-wide.

Avoiding Litigation: Reflections of an Expert Witness

by Brian Lawrence

A Disclaimer

I am not a lawyer, and no part of this paper should be interpreted as legal advice. Do not act or rely upon law-related information in this paper without seeking the advice of an attorney.

Introduction

It is January, 1998. I am serving as an expert witness in a breach-of-contract suit between two companies; one who wanted some software, and the other who was hired to produce it. My job is to analyze the design to determine the quality of the delivered software system. While looking over the various artifacts produced during this software project all I can do is shake my head and think "What a waste." Millions of dollars down the drain. If these guys had taken some simple precautions, perhaps they wouldn't be in court now. As it stands, one may be judged the winner and the other the loser, but in reality they both lose. The only question is "which one loses more?"

Around three years ago both parties agreed to build a software system to manage the client's factory floor. The software supplier was a specialist in that kind of software. It seemed like an easy job, with good value for both parties. But the system was never completed, missed many deadlines, and was way over budget. Then the client canceled the contract and now they're in court, with suits and counter-suits for breach-of-contract and fraud.

Trust me—this is not a place you want to go. It can mean lost fortunes, irretrievably damaged reputations, or even bankruptcy. Even if you win it can cost you millions of dollars. The litigation process is painful and distracting. As with many afflictions, the best cure is prevention. Don't allow yourself to get into this position in the first place.

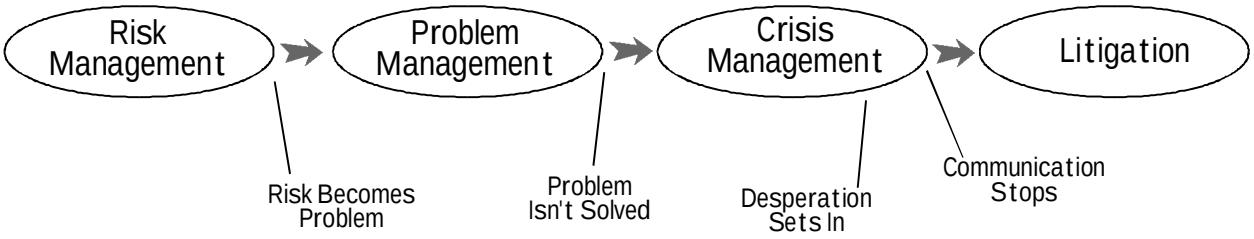
None of the ideas I'm going to suggest for avoiding litigation are new, although you may wish view them in a new light. But one thing really is new. More and more frequently, parties involved in producing software, both clients and suppliers, are finding their way into the courts. With the litigation surrounding Y2K, it's looking like the sky's the limit! Whether you like it or not, your chances of landing in court on the wrong side of a lawsuit are increasing and will probably continue to increase. If you're in a software development firm that works for hire, or an R&D group that works with a marketing group, or an MIS shop that services a corporation, the possibility of litigation is a risk you'd better start thinking about.

My intent is to discuss reasoning about practices you can adopt to help prevent litigation, not to go into any depth about the practices themselves. You might choose to adopt these same practices for other reasons. I'm limiting the discussion to reasoning about avoiding litigation.

What Leads to Litigation?

So how do events lead to such a significant breakdown that people feel their only recourse is to file suit? It's not so much the blown schedules and budgets—it's "unfair surprises."[1] Most people, including the competent experts, expect software projects to take longer and cost more than originally planned. That's not the problem. What really upsets people is when a supplier doesn't tell them about the delays, poor quality, or cost overruns until it's too late to do much about it. Lawsuits are filed when people feel like they have no other choice—and they're feeling angry, frustrated, and helpless. The principal culprit is poor communication.

Here's one model of the events leading up to a lawsuit being filed:



There are always risks in every software project. For each risk, one of two things happens. Either the risk never materializes, in which case you count your blessings, or it turns from a risk into a problem. If you can't solve the problem, it becomes a crisis. At this point it's not unusual for communication to become more difficult. You have bad news, and it's not much fun delivering it. If things break down badly enough, then communication between contracting parties breaks down altogether. This is the time when participants start thinking about looking to the courts for relief.

The Satir Interaction Model

Jerry Weinberg devotes an entire volume of his classic treatise on software management, *Quality Software Management, Vol. 2, First Order Measurement* to the topic of communication. In his book, Jerry uses the Satir Interaction Model, where a basic communication transaction is broken down into four distinct steps, for the point of view of a message recipient:

1. **Intake**: we process information coming in from the world. This step is much less passive than we often believe. We exercise a great many choices during intake.
2. **Meaning**: We put the sensory intake into some context, which gives it meaning.
3. **Significance**: We sort out the intake into more or less important, depending on our interpretation of meaning and its relevance to us. Some things we pay attention to and the rest we ignore.
4. **Response**: In some instances of intake we produce a response. This response sometimes has an effect on the outside world.

---

[1] I learned about the concept of "unfair surprise" from Tim Lister in his fine seminar on risk management.

One crucial aspect of communication that we often overlook is that just because you sent a message, that doesn't mean it was received. And what was received may not have been the message you sent. Even if you did successfully send your message, it may not have the same significance to the recipient. For successful communication we have to be reasonably certain the our messages have been sent and received successfully, and the import of the message must be appreciated.

Important Communication Topics

It's absolutely critical to communicate successfully about:

- the overall value of the project,
- a clear and specific definition of what problem it is you're setting out to solve,
- your strategy for solving it, including your approach to design and your plans,
- status of your progress,
- status of the fitness of your product.

The Value of the Project. One of our industries worst practices is undervaluing a system's worth and consequently underfunding its development. One state agency recently estimated a computer system would cost $50 million. It really would have cost at least $250 million. Even a casual examination of the system's return value over its lifetime easily demonstrated that it was well worth developing, but expectations were set that it would cost one fifth of what it would actually cost. That system was doomed to failure before development even began. The state paid its money—and got nothing. Not surprisingly, it ended up in court. Failures like these are happening in every state government, and worse ones in the federal government. Both suppliers and clients need to get clear agreement on the value of the proposed system very early in the design process.

What Problem are You Solving? It seems unlikely, but it's surprising how frequently clients and providers don't share the same understanding about what problem they're jointly setting out to solve. This is a sure fire path to a major disagreement, especially if both parties don't find out the other's point of view until the system is put into place. Gaining complete agreement of system requirements is the first and best defense against project failure.

Product Design and Plans. As a system designer, you should assume that if your client doesn't see your designs and plans during your development project, they will if things get bad enough that you end up in court. And they will use them against you. Plans are just about always more optimistic than reality and they're written in black and white. You are by far better off sharing and gaining agreement about the content of definition and planning documents with everyone involved, than keeping them confidential. When things don't go as planned, at least it's not a surprise what the plans were.

Project Status. "We will deliver in three weeks." I saw that in a manager's weekly status report fifteen weeks in a row. Then they stopped bothering to write up weekly status reports. This does not look good in an expert report. It looks a lot like incompetence. It's important to maintain both the content as well as the pattern of status reporting. Write accurate, concise status reports consistently and regularly.

Product Status. I've actually seen a memo from a CEO to a project manager explicitly instructing the manager to conceal the existence of defects which he had previously been reporting to his customer. The rationale was that they were going to fix them anyway, and the information was eroding the customer's confidence. It's important to understand that patterns of behavior and common usage of documents or other artifacts are usually interpreted as a part of the contract, whether or not anything is actually said about it in contract itself. In this example, the routine reporting of defects becomes a contractual obligation once the pattern of reporting is established. Once you start a certain practice, be very careful about changing it. Good status reports convey both the remaining estimated work as well as the evaluated quality level of the product, and the quality goals. That way customers can independently evaluate whether the estimates are reasonable, and negotiate about them if their sense is different. No surprises.

Some Troubles

There are a whole slew of ways to run afoul of the courts when developing software. Some examples are Breach of Contract, Fraud, Negligence, Deceptive or Unfair Trade Practices, Unfair Competition, Liability, and Negligent Misrepresentation. There are many others[2]. In contracted software projects, there are two that you are the most likely to encounter:

1. **Breach of contract**: Somebody violated the agreed upon terms of a contract. This usually involves late or non-delivery of a software product, or failure to conform to a product's requirements. It's important to recognize that even in the absence of a written contract, there is an implied contract simply by shaking hands and saying you'll do a job.
2. **Fraud**: Knowingly making false statements for the purposes of unfair or unlawful gain. For example telling a customer a product is nearing readiness to release when it isn't, or even that a product exists when it doesn't. A close cousin is *Negligent Misrepresentation*, where statements are not known to be false, but the teller *should* have known if they'd done reasonable due diligence.

Breach of contract is more common and easier to prove than fraud, and liability is usually limited to the value of the contract. Fraud, while hard to prove, can involve punitive damages above and beyond ordinary damages.

From The Developer's Point of View

If you're a software supplier then you're in the business of producing software for somebody else. It could be developing software for a specific customer, or for the mass market, or for some other organization inside your company, such as an MIS group. If you are working on software for some other company, you are at a higher risk of encountering litigation than in-house groups. If you're in an MIS group inside a company, you probably won't face a lawsuit over your software failures. Instead, you'll face being "outsourced." I don't know which is worse…

In either context, these ideas can help keep you out of trouble.

---

[2] See Herb Krasner's article "Looking Over the Legal Edge at Unsuccessful Software Projects," for more legal reasons.

Bidding on a Contract.

A contract is an agreement between two parties about the value and scope of a piece of work and for the terms of delivery. It's important to recognize that a contract has lost much of its value the moment it is brought out and waved about when there's a disagreement over the terms documented in it. The whole point of writing a contract is to get everything clear up front. Contracts are most valuable as a communication tool. They can also be very effective at preventing litigation if the terms have been met or devastating as evidence if they haven't.

There's way more to writing contracts than I can cover here, but here are some ideas you might want to consider:

- determine a good solution's value to the client, and scope the project in proportion to the potential value of a solution,
- reference other documents, especially project plans and requirements as official attachments to the contract,
- point out known risks,
- explicitly discuss every step of the software life cycle, including milestone "check out" points and the delivery and installation of your product.

If your client undervalues the solution, and won't agree to a reasonable budget and schedule, don't take the job, no matter how tempting it might be.

Defining and Documenting Requirements.

Requirements specifications are a developer's best friend—or worst enemy… The key is to document requirements which are clear and realistic, as opposed to somebody's blue sky notion about what they might want. If you've produced a product which doesn't meet the stated requirements, that's really easy to demonstrate in front of a jury. On the other hand, if you choose not to document requirements, then you're setting yourself up for disagreement later on. And experts will opine that your lack of documented requirements demonstrates your incompetence.

What you really want to do is to document requirements, then implement a product that meets those requirements. If you've gotten the client to sign off on those requirements then whether or not they like your software, you've been responsible for holding up your end of the deal, and the grounds for a breach of contract lawsuit are baseless.

Don't assume that just writing a requirements specification on your own is going to be sufficient to mitigate the risk of disagreement about the design problem. It's much better to collaborate with your end users or customers to produce a specification. That way everyone's understanding is much better. Remember that good communication is the whole point of the exercise.

Planning and Risk Analysis.

A risk is just a problem that hasn't come about yet. Risk management is looking for the signs of potential problems, and planning a reaction to them in the event they actually occur. There is a very simple principle which we so often tend to forget; it's easier to deal with problems if

you're thought about them and planned for them in advance rather than in the thick of battle after they come about.

Software is a risky business. Some people don't like to think about risk and believe that contemplating potential problems is "thinking negatively." If you're one of these people, I have a simple suggestion. Don't stay in software management. You don't belong there.

If you aren't doing some kind of risk management in conjunction with your planning, learn how. Failing to analyze and mitigate risk is the most direct route to litigation. Check out the risk program at the SEI or the June 1998 issue of the *Cutter IT Journal* for articles and references.

Test Planning and Testing.

Knowing the quality status of your product is a pre-condition of informing your customers about it. Remember the notion of "unfair surprise?" Don't surprise either yourself or your customer about the quality of your product. Test to the degree that you're comfortable with the level of knowledge you've acquired according to the market you're in. Obviously mission- or life-critical applications need more testing than games or any other less-critical program. If you end up in a lawsuit, you're testing practices will be examined in excruciating detail. Make sure your test plans are reasonable for the amount of risk surrounding your product. Follow the plans and keep records of the results. For more on testing, read *Testing Computer Software.*

Reporting Status.

Some of us hate reporting status, but it has to be done. You need to leave an audit trail of the progress of the project. More important, stakeholders need to know what is going on. Minimize the possibility of surprise. It may be tempting to hide bad news, but don't do it. You're better off losing a contract than getting sued over one.

From The Client's Point of View

"Here's the check, now go build it. Let us know when you're done." As a buyer of software systems, a client has a responsibility to participate in the design process. Designers need access to users in order to evaluate that they are building the right thing. You can't just send the coders off to some corner and ignore them. Clients often don't know very much about designing software, but they usually know a great deal about their business and whether a particular solution in the form of software system is what they wanted. Final system delivery is not the time to find out if a solution is acceptable. If that nasty scenario actually does happen, and you take the designers to court, experts will say "you didn't hold up your end of the job. Go away. Case dismissed." They'd be right.

Validating Requirements

You have a responsibility to go beyond just asking for the job to be done. If your supplier is to have any hope of success, you must participate in clearly and exactly defining what you want done. The producers are unwise to proceed if you don't sign off on the requirements. A closely-examined, fully-understood, explicitly-negotiated and signed-off requirements specification is a mandatory attachment to any software contract. Don't go on without it. A good requirements specification is your best defense against project failure—and later in court.

Evaluating Proposals

Software designers frequently have the fear that "if we tell them what it will really cost, they won't give us the job." Software systems are often much more valuable than they first appear. Especially when examined over the full lifetime of the system. Software development projects should be budgeted based on the potential value that the software system will deliver. If it's a big system on which your business depends, expect to spend a lot. Short changing the development budget is one of the best ways to ensure total project failure. Be reasonable. Projects have to have reasonable and fair value for both sides.

Doing Due Diligence

What makes you think your supplier can do the job? One of the most common circumstances of software project failure is when a group of designers sets out to build something which is bigger or more complex than anything they've built before. Software design processes do not scale up. What works for small projects does not work for large projects. Designers must choose approaches which are appropriate for the job. It is the client's responsibility to evaluate their supplier's capability to do the job. Ask for a design strategy as a part of the contract. Check references. If the developers have never done a job like yours before, assume that the project has a high risk of failure and do something about it. If you don't, then you'll get what you deserve.

Understanding Risk Distribution

Some clients think that once you've hired a group of designers, all the risk is on their shoulders. Not so. Risk is always distributed across both parties. Rather than ignore risk, manage it. If you don't know how, learn. Make sure you gain an understanding of how the designers perceive the risk. If they don't pay attention to risk, don't work with them. They're not qualified to develop software.

Project Status and Reporting

You need to know what's going on in your software project. Demand regular status reports. Good status reports convey progress of tasks relative to a life cycle model, current assessed quality level, evidence of why that quality assessment is believable, evaluation of current staffing, prioritized perception of risks, and review status of project artifacts. There may be other items you will want reported as well.

What It Looks Like To An Expert Witness.

Expert witnesses are sought out based on their qualifications, including writing articles and books, and years of experience in the software industry. There's a natural tendency for experts to be drawn from the more formal side of software development. Unlike all other witnesses, expert witnesses are asked for their opinions about what happened. All other witness are allowed to testify only about facts as they see them. When asked for an opinion about what is reasonable, experts are more likely to side with more formal practices rather than less formal ones. Of course, they must make their case just like anyone else, but remember that they're going to tend toward the more formal side because that's where many expert witnesses come

from. So if you didn't write any specifications and don't have any plans expect the experts to come down on you like a load of bricks.

It's normal during the discovery and preparation phases of a lawsuit for an expert to see the entire collection of artifacts produced during a software project. The quantity can be staggering. Project status reports, all types of specifications, the product itself, the source code in various stages of development, the build system, every bug report, hand written notes from meetings, letters of intent, contracts and licensing agreements, memoranda, and logs of support and other calls are some examples. If you think you can hide poor practices, think again. They nearly always show through. Good experts will point out the lack of crucial documents as well as the "smoking guns" lurking in the huge pile of paper.

There's a tendency for some to claim that their software project didn't turn out well because that's the nature of software projects, that they're disorganized, late, and over budget, and that's just the way it is. "Everybody knows that."

Don't believe it. Don't try it. Experts will convincingly debunk that myth to the court. If you tried something high risk and used the appropriate methods, all that will be documented in the artifacts. You may not have gotten a positive outcome in your project, but it was know that there was some high probability of that particular outcome. If you were just sloppy and didn't use appropriate methods, that will show in the artifacts as well. The argument that software projects are inherently uncertain doesn't hold water.

Some Things To Avoid.

There are many things you should consider doing to help ensure a successful project. Here are some things you should never do. They are litigation bombshells.

Documenting for the sake of form. Somebody says you should have a requirements specification or a test plan, so you make one up just to say you do, but never actually use it. These documents will almost certainly be used against you in a lawsuit. What are you going to say? "I was only kidding!" If you do that you build up the notion that if you cannot be trusted about your written specifications, then when can you be trusted?

Write specifications and plans that you fully intend to implement and follow. Use them as communication tools, and work to make them effective. Choose only ones that are helpful to the project. If an approach doesn't work out, place a statement to that effect in the last revision of the document and in your status reports. Keep working documents up to date. You may not think it's much fun, but those documents will be crucial evidence if a lawsuit is filed, and you want them to testify a well as possible for you. Experts will take them at face value unless there's some compelling evidence suggesting that they've been superseded by something else.

Lying. I know this sounds obvious, but never lie! Especially not in writing. Sometimes it's tempting to suppress bad news, in the hopes that by the time the other side finds out, the bad news has gone away. Don't be tempted. Always deliver the true status at the time it's known. I'll say it again. Never lie.

Playing the estimate game. "They won't do the project if we tell them how much it's really going to cost." As an industry, we have a strong tendency to underbid jobs, then "go back to the well" for more resources when we run out and the job's not yet done. There is this notion that we won't win the job if tell the truth. In cases like that, you're better off not getting the job. Jobs like that are set up for failure before they even start.

If it isn't documented, we can't be held accountable. You may get the idea that to be safe, you just don't leave behind any smoking guns. You might be surprised about how many documents are created during a software project. The other side will ask to see them all and most likely get them. They'll tell the story all by themselves. Documents that aren't there but should have been can speak as loudly as those that are there.

As an aside, never destroy or suppress evidence rather than give it up to the other side. For most litigation, individuals are not personally liable for their company's legal problems. If you destroy or hide evidence, and the other side finds out, you will be held personally liable, and probably go to jail. Judges don't have any sense of humor where withholding evidence is concerned.

Conclusion

My recommendations for practices aren't anything new or original in the software engineering literature, but lots of software organizations still don't do them. Up until now, you may have ignored the software pundits' advice about them. If you haven't adopted such practices, you'd better start considering doing so. It's sad to say, but the growing tendency toward litigation is providing stronger motivation where desire for better quality or predictability hasn't. If you get nailed for fraud, you can lose more than you spent on development costs together with all your profits. Punitive damages can be pretty nasty.

Until recently, there's not been any even remotely agreed-to standard for software professionalism. That's changing. New efforts are moving forward to certify software professionals. For example, the Texas Board Professional Engineers has just established a standard for Software Engineering Discipline in June of this year, and the ASQ has its qualification exam for Software Quality Engineer. In some ways, the courts have always been a forum where important professional practices have been established. That's happening now in the software field. As experts get qualified and courts make decisions, better and better ideas are forming about what constitutes reasonable practices.

The best way to avoid litigation is to deliver a successful product, where your customer is happy, and there's no reason for disagreement. That aside, your best hope to avoid litigation is to choose a software design process which is appropriate for the level of risk and potential value of the solution to the design problem. If you have a reasonable design approach and can demonstrate proper care and attention in your design and communication approaches, there's little basis for any lawsuit.

Finally, I hope I never see you on either side of a lawsuit.

References

DeMarco, Tom, and Tim Lister, "Both Sides Always Lose: The Litigation of Software Intensive Contracts", *Cutter IT Journal*, Vol. 11, Number 4, April 1998

DeMarco, Tom, and Tim Lister, *Risk Management for Software: Course Notes*, The Atlantic Systems Guild, Version 1.4, February 1998

Kaner, Cem, Jack Falk, and Hung Nguyen, *Testing Computer Software* , New York: International Thompson Press, 1993

Krasner, Herb, "Looking Over the Legal Edge at Unsuccessful Software Projects", *Cutter IT Journal*, Vol. 11, Number 4, April 1998

Weinberg, Jerry, *Quality Software Management, Vol. 2: First Order Measurement*, New York: Dorset House, 1993

Risk Management (whole issue topic), *Cutter IT Journal*, Vol. 11, Number 6, June 1998

The Texas Board Professional Engineers, http://www.main.org/peboard/sofupdt.htm

The American Society for Quality – Certified Software Quality Engineer, http://www.asq.org/standcert/certification/csqe1.html

Unnamed court cases in which I have served as an expert witness, whose identity and content are under non-disclosure.

A COST ESTIMATION BASED APPROACH TO QUANTIFY SOFTWARE RISK EVALUATION RESULTS

Peter Hantos, Ph.D.

Xerox Corporation

701 South Aviation Blvd., MS: ESAE-375

El Segundo CA 90245

Phone: (310) 333 - 9038

Internet: peter.hantos@usa.xerox.com

ABSTRACT

Risk management is one of the most critical and most difficult aspects of software project management. There is a vast literature documenting approaches and tools that address risk assessment and mitigation. In this presentation, we introduce "hard" and "soft" classifications of the approaches, that are based on either the mathematical rigor describing the development of the model/tool or the mathematical rigor expected from the user during the use of the model/tool. We then compare benefits and liabilities of the various approaches. The purpose of this analysis is to provide a foundation for a simple, practical approach to risk analysis, which combines the identified benefits, without suffering from the known liabilities. The solution we are presenting is a combination of the SEI/SRE (Software Engineering Institute/Software Risk Evaluation) method, and COCOMO (Constructive Cost Model) II, the popular software cost estimation model and tool.

KEYWORDS

COCOMO II, Software Project Management, Software Risk Management, SEI

BIOGRAPHY

Peter Hantos is Principal Scientist of the Xerox Corporate Software Engineering Center, where currently his main responsibility is the definition of the corporate software development process standard. His tasks include coaching and mentoring software process improvement teams across Xerox; identifying and sharing best practices, methods and tools; establishing repositories and asset libraries of guidelines, examples, and templates; and facilitating the acquisition/development of software training. Finally, he is also working on the internalization and institutionalization of systematic software risk management approaches for the corporation, and as part of that work he developed the Software Technology Readiness process. Previously at Xerox, he managed a software engineering organization that provided methodology, tools and infrastructure support for members of the Corporate Research and Technology Division located in El Segundo, California. Dr. Hantos is a Senior Member of the IEEE (Institute of Electrical and Electronics Engineers), and Member, ACM (Association for Computing Machinery).

1. INTRODUCTION

Software risk management, if practiced properly, is a set of continuous activities, used to identify, analyze, plan, track and ultimately control the risks, and is conducted in the context of everyday project management. The first reaction of a project planner may be to avoid risks all together, but relying strictly on avoidance as a risk mitigation technique is usually not adequate. Project success is primarily dependent on the ability to manage the delicate balance of opportunities and risks. Unfortunately when all risk goes away, so does opportunity. Since risks ultimately manifest themselves as incremental, unexpected cost elements, risk management can also be viewed as a way to dynamically handle the cost/benefit analysis of the project. While the techniques for risk identification are usually handled separately from software cost estimation, the cost aspects of risks can be used as a communication vehicle during risk prioritization. This leads to our main proposal, i.e., making the connection between an established risk assessment tool, SEI/SRE and an industry-wide accepted software cost estimation tool, COCOMO.

But is there any connection to software quality? The answer is yes. Beside balancing opportunities and risks, project managers are also constantly balancing the elements of the ***Q****uality* - ***C****ost* - ***D****elivery* (***QCD***) triangle. Contrary to the popular misconception, quality is not free; consequently, in the shadow of cost-overruns or schedule delays, quality must compete directly for the available project resources and funds.

2. RISK MANAGEMENT

Based on Boehm's work [1], the risk management issues can be classified as follows:

RISK ASSESSMENT		RISK CONTROL	
Risk Identification		**Risk Management Planning**	
☞	Checklists		Buying Information
	Decision Driver Analysis		Risk Avoidance
	Assumption Analysis		Risk Transfer
	Decomposition		Risk Reduction
Risk Analysis			Risk Element Planning
	Performance Models		Risk Plan Integration
☞	Cost Models	**Risk Resolution**	
	Network Analysis		Prototypes, Simulations
	Decision Analysis		Benchmarks
	Quality Factor Analysis		Staffing
Risk Prioritization			Analysis
	Risk Exposure	**Risk Monitoring**	
	Risk Leverage		Milestone Tracking
	Compound Risk Reduction		Top-10 Tracking
			Risk Reassessment
			Corrective Action

In this paper we focus on the connection between software risk identification and cost-model based risk analysis. (The Taxonomy Based Questionnaire, which will be discussed in detail later, is basically a checklist.) Please note that this approach permits the determination of cost ramifications of risks in the software development domain only. Other, very much quantifiable business risks, such as loss of market opportunity, loss of sales, and so on, can be determined from the software development data, but cannot be automatically computed. Similarly, the tools can provide quantification of the risks, but the overall prioritization and resolution has to be done in the full context of project management.

3. RISK TAXONOMIES

Generally defined, software risk taxonomy* provides a basis for the systematic organization and analysis of risks in a software project. The title, Risk Taxonomies, is intentionally plural, because in addition to describing the importance of a specialized risk taxonomy, we also want to note a level of -- in our opinion, undesired -- proliferation of software risk taxonomies.

Here is a brief reference to articles where the "overt" or "covert" development** of taxonomies play a role:

- SEI's report lays the foundations of the development of the SEI taxonomy, and also discusses the basic concepts of risk taxonomies [2].
- In Garvey's presentation, the risk elements are described in risk templates, and the taxonomy is implemented via web-based links [3].
- Moynihan chose the approach of eliciting constructs from experienced managers to determine how they assess risk, after deciding that the taxonomies published in the literature were not adequate [4].
- Barki et al. conducted a wide review of the literature and determined 35 variables that were used as taxonomy for risk assessment [5].
- Madachy developed an extensive rule based system (identifying 600 risk conditions), where the rules were structured around COCOMO cost factors, reflecting an intensive analysis of the potential internal relationships between the cost drivers [6].
- Känsälä built his tool around 15 risk items he identified as critical, after filtering the data received from 14 selected companies [7].
- Conrow et al. documented experiences on large projects at TRW, and defined a taxonomy consisting of 17 software risk issues [8].
- We, at Xerox reviewed and used the SEI-developed taxonomy [9] in five major projects, and found that while it did not provide a complete coverage for all situations, by combining it with their Taxonomy-based Questionnaire (TBQ) will be applicable for most of our major software projects [10].

We found that all authors decided that the introduction of new risk categories or the creation of a whole new taxonomy was needed. In our opinion, this is not always justified. During the pilot of the SRE (***S***oftware ***R***isk ***E***valuation) method at Xerox, the criticism of the SEI taxonomy was in two areas. In large software projects, respondents complained that the terms and language of the TBQ sometimes did not map to local terminology (for example, the classification of contractor relationships). Second, respondents from firmware development projects stated that in their work the distinction between hardware and software was somewhat blurred, and their risk issues were not always adequately covered.

Our view is that the application of any risk taxonomy always requires a certain level of customization before use, and the quest for the "perfect" taxonomy -- consequently the "perfect" risk management tool -- is fruitless. Also, in the case of actual, computer-based tools, the taxonomy always ends up "hard-coded" into the tool. Instead of further specialization, the approach should be exactly the opposite, i.e., we should step back and find a framework that is applicable for a large class of projects, with the understanding that some level of customization will take place. As we stated earlier, the SEI taxonomy satisfied this requirement.

---

* From Webster's dictionary: tax-on-o-my \tak-'sän-ə-me\ n 1: the study of general principles in scientific classification.

** "Covert" taxonomy means that the risk sources and their structure are not visible, and they are embedded in the tool. In case of "overt" taxonomy there is an explicit reference to the description of the risk hierarchy.

4. THE SEI SOFTWARE RISK EVALUATION METHOD

The scope of the SEI/SRE method is the identification, analysis, and preliminary action planning for mitigation. The software risk taxonomy (See Appendix) provides a consistent framework for risk management. It is organized on the basis of three major software risk classes: Product Engineering, Development Environment, and Program Constraints. At the next level, risk elements of the classes are identified, which are further decomposed into risk attributes. SEI also developed a tool called TBQ (***T***axonomy-***B***ased ***Q***uestionnaire) to carry out structured interviews by an independent assessment team. A sample segment of the TBQ follows (The numbers refer to the originals in the full documentation).

Class: A. Product Engineering
Element: 2. Design & Implementation
Attribute: d. Performance

Starter question: [22] Are there any problems with performance?

Cues:
- ☛ Throughput
- ☛ Scheduling asynchronous 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?

. . .

During the interviews the risks are identified and recorded. At the end of the interviews, the impact and likelihood of occurrence is identified. At the completion of all interviews, a consolidated report of risks is presented. In Figure 1, an example is shown to demonstrate how the risk issues are recorded during an SRE session. The relevance of the shaded rows will be explained in Chapter 7, where this risk report is used to demonstrate our process).

Risk Class and Element from TBQ	Issues and Concerns Recorded during the SRE sessions	Risk Magnitude Rating
Program Constraints/ Resources	Current plan is schedule driven	7.5
Program Constraints/ Resources	Bottom-up plans do not support the schedule	7.5
Development Environment/ Management Process	Management is not ready to reconcile the differences between the engineering plan and the business plan	7.5
Program Constraints/ Resources	Top-level plan is unrealistic, and it is not based on past track-record and experience	7.5
Development Environment/ Management Process	Inability in estimating effort due to the lack of experience with the new technology	7.1
Development Environment/ Development Process	**Feedback from implementers to architects takes too long, and there is no closure on certain issues**	6.3
Development Environment/ Development System	**Capacity of the development system (network bandwidth and speed of compilation) is a concern, and impacting schedule**	6.1
Program Constraints/ Resources	Lack of confidence in the current plans	5.6
Development Environment/ Development System	**Lack of availability of adequate number of software licenses**	4.3
…	**…**	**…**

Figure 1. A sample record of risk issues during an SRE

5. COST ESTIMATION WITH COCOMO II

Xerox is aggressively pursuing the application of COCOMO for software cost estimation, and participates in the USC/CSE (***U***niversity of ***S***outhern ***C***alifornia/***C***enter for ***S***oftware ***E***ngineering) Industrial Affiliates Program. At this time 27 industrial affiliates provided data or input to enhance and fine-tune the COCOMO model. Here, we provide a conceptual introduction to COCOMO only. For up-to-date details, please see the appropriate COCOMO II materials on the USC website (http://sunset.usc.edu) or [11] in hard copy format. (Please note that since publication of that article, the model was renamed COCOMO II from COCOMO 2.0). The COCOMO II model uses 163 data points from the affiliates' databases, and it is the enhanced version of the earlier COCOMO 81, which was originally developed using only 64 data points from TRW. Beside refining the model, USC also provides MS/Windows, Sun and Java versions of the tool, based on the current version of the model. Due to the model's popularity, a number of industrial tool-vendors incorporated the COCOMO II model into their software cost estimation tool offerings.

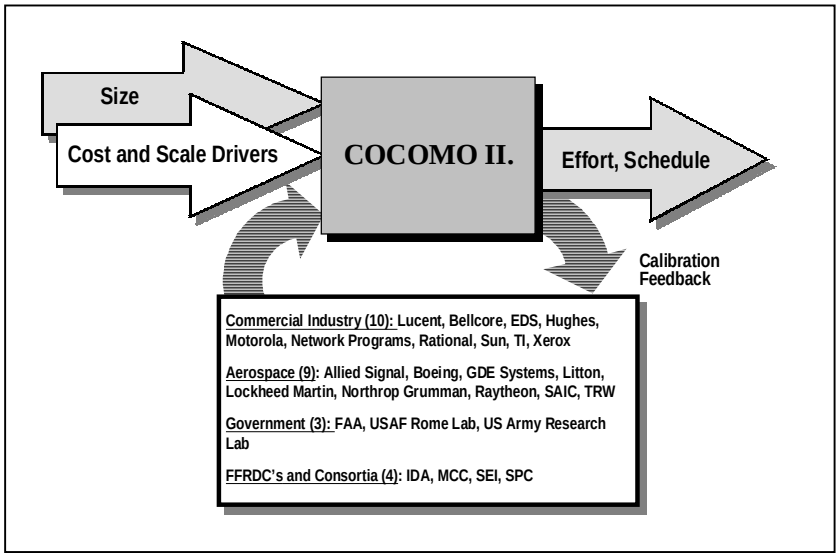


Figure 2. COCOMO II Software Cost Estimation Model

COCOMO II Cost Drivers

ACAP	Analyst Capability
AEXP	Applications Experience
CPLX	Product Complexity
DATA	Database Size
DOCU	Documentation to match lifecycle needs
LTEX	Language and Tool Experience
PCAP	Programmer Capability
PCON	Personnel continuity
PEXP	Platform Experience
PVOL	Platform Volatility
RELY	Required Software Reliability
RUSE	Required Reusability
SCED	Required Development Schedule
SITE	Multi-site operation
STOR	Main Storage Constraint
TIME	Execution Time Constraint
TOOL	Use of Software Tools

COCOMO II Scaling Drivers

PREC	Precedentedness
FLEX	Development Flexibility
RESL	Architecture/Risk Resolution
TEAM	Team Cohesion
PMAT	Process Maturity

Figure 3. COCOMO II Cost and Scale Drivers

During the estimation process, the estimator determines the value of scaling constants and cost drivers, using the supplied rating tables, and enters the value in the appropriate screen of the tool. An example based on the COCOMO Model Definition Manual for the cost driver rating guideline follows:

Cost Driver: **SCED** (Required Development Schedule)

Actual Rating	Entry for the tool's screen	Very Low	Low	**Nominal**	High	Very High
Rating Guidelines	Relationship to nominal	75%	85%	**100%**	130%	160%

This cost driver belongs to the group of cost drivers that characterize the *project* to be estimated, and it measures the schedule constraint imposed on the project team. The ratings are defined in terms of percentage of schedule stretch-out or compression with respect to a nominal schedule for a project requiring a given amount of effort. Compressed or accelerated schedules tend to produce more effort in the later phases of development because more issues are left to be determined and resolved.

6. "HARD" AND "SOFT" APPROACHES

As indicated previously, we classify the different risk analysis approaches based on the mathematical rigor required. The first example of a "hard" approach is [6], where the author created the risk taxonomy outright around the COCOMO cost drivers. This is an impressive system, using knowledge-engineering methods to refine their risk quantification scheme. The second example is [7], where the author used Madachy's basic approach, but instead of working around a particular cost estimation tool, he derived his own risk database using risk questionnaires and historical project data. Experimental integration of this risk front-end (Called RiskMethod) was carried out with three different cost estimation tools. The underlying principle in both cases was using regression analysis to determine the model's internal coefficients. Both approaches required extensive calibration to achieve acceptable results. Finally, in both cases the author's initial objective was not only to assess and prioritize but also to quantify software risks.

The first example of a "soft" approach is [8]. In this TRW approach, the authors defined a list of specialized risk issues (it could be viewed as a one-level risk taxonomy) and this taxonomy was used by internal Risk Review Boards (RRB) to assess risks via intensive monthly sessions with key representatives of the functional and support areas. No particular efforts were made to quantify the impact of identified risks. The second example is [9], where a more complex taxonomy was used by a combined external-internal assessment team to assess risks via one single assessment, where for several days the assessment team formally interviewed a cross-section of the development organization. The interviewees were not necessarily key representatives, and they represented a "vertical" sample of the people in the development organization. Non-attribution is a key guiding principle during the sessions, and the names of the persons raising concerns are kept confidential. The common theme in both cases was that interviews and guided discussions were used, and no provisions were made for risk quantification.

We had primary experience with [9]. The SEI approach had two advantages: (1) The taxonomy was broad and detailed enough to carry out very efficient interviews, and (2) the non-attributional approach was very useful in uncovering risks that were well-known and obvious to the developer community, but due to lack of trust or broken communication was not acknowledged by management. It became obvious that the ability to quantify would be helpful in prioritizing and presenting the risks to the Decision Authority.

These experiences led us to recognize that it would be useful to combine the best of both worlds: keep the SEI method as a risk front-end and use COCOMO for quantification. The benefit is that we maintain flexibility and easy customization at the front-end, while using an already calibrated COCOMO model to quantify the results. We did not perceive the benefit of an expert-system approach, because it introduced a complicated, and in our opinion unnecessary, calibration and learning process.

7. USING COCOMO II TO QUANTIFY SOFTWARE RISK EVALUATION RESULTS

On a conceptual level, the task can be phrased as an ***m:n*** mapping from the risk taxonomy into the COCOMO scale and cost driver taxonomy. This complexity resulted in the identification of nearly 600 (!) risk conditions in Madachy's tool. While the precise mapping and the identification of all possible combinations are necessary to create a knowledge-management tool, we found that the goal never can be fully accomplished, and customization will have to take place before use anyway.

In the recommended "soft" approach the following process steps need to be followed:

❶ Carry out an SRE, and customize the TBQ by the assessment team before and during the interviews

❷ Filter, consolidate and interpret results, map risks into the COCOMO scale/cost driver taxonomy

❸ Execute COCOMO estimation with ***baselined*** scale and cost factors

❹ Execute COCOMO estimation with ***risk-adjusted*** scale and cost factors

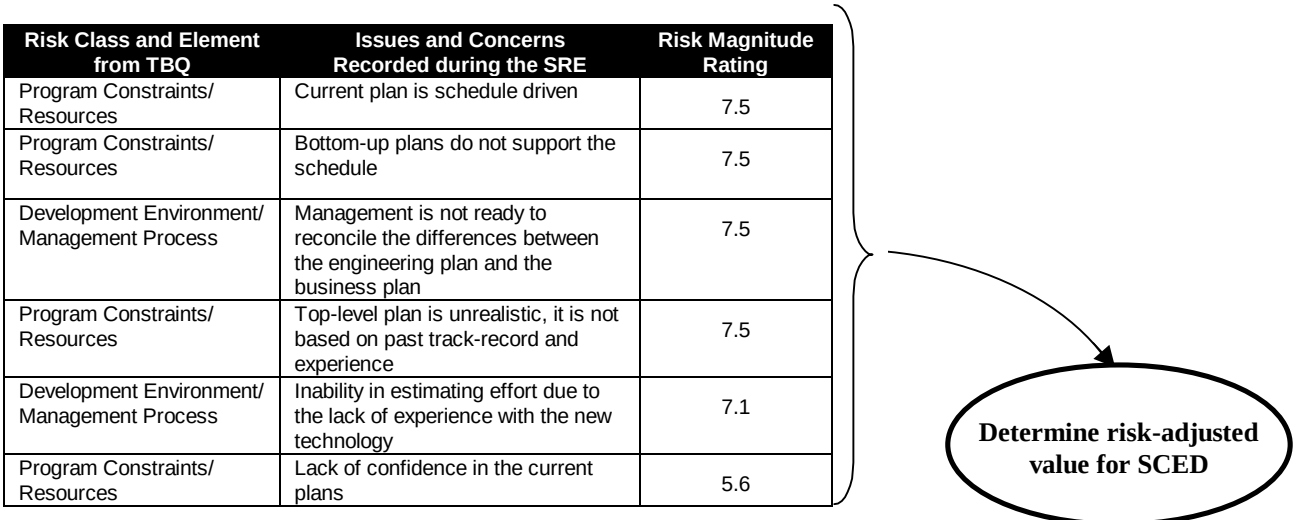
❺ Compare baselined and risk-adjusted cost estimations

We believe that the SRE as a risk identification method yields better, more detailed and relevant risk items than any input processes which are customarily used with the described "hard" risk assessment tools.

Example of the approach showing the risk-adjustment of the SCED cost driver:

On Figure 4, we demonstrate the process using the earlier sample risk report. One worksheet per scale or cost driver needs to be prepared for the assessment team meeting, and distributed to the assessment team members. In this case, during the risk consolidation process, we identified that only the risk items in the non-shaded rows map into the SCED cost driver, so this data was copied to the appropriate worksheet. In the next step, every assessment team member worked out his/her own risk adjustment for every COCOMO II driver, using the ratings in the third column, which were representing the originally determined risk magnitudes in a range of 0 - 9*. Based on the rating, a determination was made by the team for the final risk-adjusted value of the cost driver, basically following the same process that was used for consensus building earlier when the risk magnitude values were finalized.

Risk Class and Element from TBQ	Issues and Concerns Recorded during the SRE	Risk Magnitude Rating
Program Constraints/ Resources	Current plan is schedule driven	7.5
Program Constraints/ Resources	Bottom-up plans do not support the schedule	7.5
Development Environment/ Management Process	Management is not ready to reconcile the differences between the engineering plan and the business plan	7.5
Program Constraints/ Resources	Top-level plan is unrealistic, it is not based on past track-record and experience	7.5
Development Environment/ Management Process	Inability in estimating effort due to the lack of experience with the new technology	7.1
Program Constraints/ Resources	Lack of confidence in the current plans	5.6



Scale or Cost Driver	Baseline value	Risk-adjusted value
SCED	Nominal	***Very Low***

Figure 4. Developing the risk-adjusted value for COCOMO II scale and cost driver

* For details on risk ratings, see the referenced SEI document on conducting an SRE [9].

SUMMARY

Basic concepts of risk management were presented with a focus on the "front-end" issues (risk identification and analysis.) An overview of the most significant literature was given on risk assessment methods and tools with a classification and evaluation based on the differentiation between hard and soft approaches. With a declared (and by now hopefully justified) bias toward light-weight solutions, a simple method was introduced that is based on the combinative use of the SEI/SRE approach as the main risk identification vehicle and the COCOMO II tool as the means to quantify software domain related risks. We believe that this is a preferred lightweight approach, because it uses established and already familiar and applied tools. We found that customization and calibration are always needed even using very comprehensive and sophisticated knowledge-engineering based tools. In view of this recognition, we believe that applying the customization effort to the existing tools in a lightweight setup is a more efficient approach than purchasing and implementing new and complex tools.

APPENDIX

Complete SEI Software Risk Taxonomy

A. PRODUCT ENGINEERING	B. DEVELOPMENT ENV.	C. PROGRAM CONSTRAINTS
1. **Requirements**	1. **Development Process**	1. **Resources**
a. Stability	a. Formality	a. Staff
b. Completeness	b. Suitability	b. Budget
c. Clarity	c. Process Control	c. Schedule
d. Validity	d. Familiarity	d. Facilities
e. Feasibility	e. Product Control	
f. Precendented		
g. Scale		
2. **Design and Implementation**	2. **Development System**	2. **Contract**
a. Functionality	a. Capacity	a. Type of contract
b. Difficulty	b. Suitability	b. Restrictions
c. Interfaces	c. Usability	c. Dependencies
d. Performance	d. Familiarity	
e. Testability	e. Reliability	
f. Hardware Constraints	f. System Support	
g. Non-Development Software	g. Deliverability	
3. **Code and Unit Test**	3. **Management Process**	3. **Program Interfaces**
a. Feasibility	a. Planning	a. Customer
b. Testing	b. Project Organization	b. Associate Contractors
c. Coding/Implementation	c. Management Experience	c. Subcontractors
	d. Program Interfaces	d. Prime Contractor
		e. Corporate Management
		f. Vendors
		g. Politics
4. **Integration and Test**	4. **Management Methods**	
a. Environment	a. Monitoring	
b. Product	b. Personnel Management	
c. System	c. Quality Assurance	
	d. Configuration Management	
5. **Engineering Specialties**	5. **Work Environment**	
a. Maintainability	a. Quality Attitude	
b. Reliability	b. Cooperation	
c. Safety	c. Communication	
d. Security	d. Morale	
e. Human Factors		

REFERENCES

[1] Boehm, Barry W. "Software Risk Management." *IEEE Computer Press, 1989.*

[2] *Taxonomy-Based Risk Identification*. Technical Report CMU/SEI-93-TR-16.

[3] Garvey, P.R. "A Risk Management Information System Concept for Systems Engineering Applications." *Leesburg, VA: 28th Annual Department of Defense Cost Analysis Symposium, September 1994*

[4] Moynihan, T. "An Inventory of Personal Constructs for Risk Researchers." *Journal of Information Technology, Vol. 6, No. 4, 1996*

[5] Barki, H., Rivard, S., and Talbot, J. "Toward an Assessment of Software Development Risk." *Journal of Management Information Systems, Vol. 10, No. 2, 1993*

[6] Madachy, R.J. "Heuristic Risk Assessment Using Cost Factors." *IEEE Software, May/June 1997*

[7] Känsälä , K. "Integrating Risk Assessment with Cost Estimation." *IEEE Software, May/June 1997*

[8] Conrow, E.H., and Shisido. P.S. "Implementing Risk Management on Software Intensive Projects." *IEEE Software, May/June 1997*

[9] Sisti, F.J., Joseph, S. *Software Risk Evaluation Method, Version 1.0*. Technical Report CMU/SEI-94-TR-19

[10] Hantos, P. "Experiences with the SEI Risk Evaluation Method." *Portland, Oregon: Pacific Northwest Software Quality Conference, 1996.*

[11] Boehm, Barry W., Clark, B., Horowitz, E., Westland, C., Madachy, R., and Selby, R. "COCOMO 2.0 Software Cost Estimation Model." *American Programmer, July 1996.*

8th International Conference on Software Quality
16th Pacific Northwest Software Quality Conference
October 12-15, 1998

From VICTIM to VICTOR:

The Trials and Tribulations of a Neophyte Software Testing Team

Elizabeth A. Adams, M.Ed.
Quality Assurance Manager
tele: (225) 231-0754
fax: (225) 231-2302
email: badams@lwcc.com

Beth Adams is Quality Assurance Manager at Louisiana Workers' Compensation Corporation. She holds a Master of Education from Louisiana State University and has 25+ years of experience in teaching, training, sales and project management. She is an active member of the American Society for Training and Development and the American Society for Quality. She is an accomplished speaker, who presents locally, regionally and nationally on various topics.

For the past three years Beth has assisted in creating, developing and managing a Software Testing Team for her current employer. The following is a case study which relates to the agony of defeat, the thrill of victory and the ongoing journey experienced by a newly created Software Testing Team as it navigates the political and technical challenges of effectively managing "testing" throughout the software development life cycle.

Corporate Background

Louisiana Workers' Compensation Corporation (LWCC) opened its doors for business in October, 1992. It was legislated into being to help solve the workers' compensation insurance crisis in the state of Louisiana. Workers' compensation fraud had carriers literally fleeing the state and there remained no viable alternatives for employers, required by federal law, to cover workers who were injured on the job.

LWCC purchased a workers' compensation insurance software package and proceeded to install, test, enhance and customize this application in accordance with the needs and regulations as set forth by the state of Louisiana and the current marketplace. For the first six months the software creator and supplier was onsite to assist with these activities.

The first year of operation was a hectic, driven year . . . one in which insureds (employers/customers) with no current workers' compensation coverage were transferred en masse to our corporation for insurance coverage. In addition to the simple insurance workflows of receiving and processing applications and injury reports, we dealt with the more complex issues of rating, experience modifiers, schedule debits/credits, claims management and reserves --- all in a workers' compensation insurance marketplace that had literally gone haywire!

During the first two years of operations, (10/92 - 10/94) the testing efforts on the software application were mostly adhoc, and very departmentalized. There were no formal testing processes, or centralized testing team in place. Modifications and enhancements were designed and executed on a first come, first serve basis, with little effort made to determine the impact on overall corporate business goals. In many regards, chaos was the order of the day! There were simply too many system modifications to be accomplished and not enough resources to successfully implement them. The overwhelming and continuing challenge of the customizations, modifications and enhancements to our primary software application package lacked definition and was fraught with unrealistic expectations of the management team and the employees.

Enter: The Quality Assurance Testing Team

In January, 1995, a Quality Assurance Department was created. A new manager was appointed and seven employees were pulled from various departments to form the new testing team. These employees had been executing the testing function using various methodologies, and with varying levels of support and success, throughout the corporation. While this may have seemed like an extremely well-thought out corporate move, destined for great accomplishments, the reality of the situation was slightly skewed. The new members of this Quality Assurance Testing Team wondered if they had been promoted to a new challenge or demoted to obscurity.

It is common knowledge it's not a "real" department unless there's a vision and/or mission statement. Hence the vision of this newly formed Quality Assurance Team:

We are a professional, technical, quality-driven, software testing team who serves the needs of our customers by delivering defect-free software, in a timely manner, that exceeds customer expectations.

However, for each vision, there looms a reality:

Who died and made our Testing Team . . . the ENEMY ???

Thankfully, the newly appointed QA Manager took the bull by the horns and immediately began a program of image and damage control. The good news was that these employees were truly "power users" who had been working with this software application since Day One of company operations. The bad news was that they were ill-informed regarding the profession of "software testing." The QA Manager contracted with a consulting firm for an onsite workshop. The workshop was designed to provide education in the basics of software testing, an overview of the life cycle of software development, and an outline of the various testing tools, to include the test plan, test scripts, case scenarios, defect/incident reporting (pre and post test), test review and test analysis.

This workshop provided a well-needed foundation for this newly formed Quality Assurance Testing Team. The team members discovered that they had actually been performing and executing many of the "foundations" of acceptable software testing. They merely had to more formally organize, document and track their activities in order to deliver expected results in a more professionally-respected, industry-accepted manner.

With new knowledge under their belts, the team began to tackle the task of implementing the new tools and processes they had learned. Using workflow methodology, the QA Team documented a "Life Cycle Of A Workers' Compensation Insurance Policy" and a "Life Cycle Of A Workers' Compensation Insurance Claim." Based on this information, the Team developed common test scripts and case scenarios for the most frequent processes which continuously recurred throughout these life cycles. As software modules required enhancements or modifications, test scripts and case scenarios were updated and added to the testing resources.

In November, 1995, the QA Manager was promoted to another position within our corporation. I had been serving as Program Development Specialist in the Education & Training Department. Part of my responsibilities included creating and developing software training to better prepare employees for working with this unique software application. This corporate experience provided me with just enough knowledge to be an extremely dangerous candidate for this position. Add to this my naturally optimistic, self-confident attitude and you have a whole lot of plutonium just waiting for a weak spot! Fortunately, our QA Team is responsible for defect-free "insurance" software . . . not a nuclear plant or life-saving medical equipment. Nonetheless, our executive team considered the flawless functioning of our computer-driven insurance system paramount to meeting its corporate goals.

Implementing CAAT: Corporate Attitude Adjustment Therapy

If quality technical testing were all that were required for a successful effort, software testing would be a whole lot easier. But think about how dull life would be without "office politics." Not only must our QA Team maintain its status as the "final barrier" for a defect-free product before release . . . we must do so with aplomb! Considering the battered egos and wilting self-esteem held by most of the QA team members, interpersonal skills training and motivational workshops seemed the order of the day.

The entire team was treated to a two-day Myers-Briggs Workshop. The IS Applications and Operations Departments were also invited, and supervisors and managers were included. As you can imagine, there was some marked resistance to attending one of those "touchy-feely" workshops . . . IS folks are not exactly known for their expressive, outgoing personalities . . . which was the whole necessary point of everyone's participating in this adventure. The workshop leader was superb! The first day was spent administering / taking the test (the Myers-Briggs Instrument), and receiving an overview of the different types of personalities that might emerge. The second day, participants received the results, indicating their specific personality types. Individual and group exercises were conducted to provide practice and observance of the different personality types in various work settings.

We were most pleased with the results of the workshop. We transferred the knowledge and skills learned into our work environment --- most notably, a new respect for each other's differences, and a bevy of tools for interacting with different personality types. The moral of the story for our particular corporate setting: You might consider approaching and interacting with each personality type in accordance with his/her comfort level --- depending on how badly (and quickly) you want results.

This experience was so rewarding, and so freely complimented by the participants, that many other LWCC departments participated in similar workshops . . . which led to the next successful phase of our implementation plan.

Commitment to the Process !

How often have you experienced this scenario: The first you learn of a system modification or enhancement is via the email you receive when it's ready to be tested! The front end of the software development life cycle was as frustrating to our developers as it was to the testing team. System modifications and enhancements were being submitted at random with no thought, rhyme or reason. First come, first serve! Whoever carried the biggest stick beat their way to the front of the line!

As much as you might like to, we really can't blame the executive team. They are often as intimidated by our computer systems as the lowest knowledge worker. Not only is it unfair of us to take advantage of this level of discomfort, but depending on the level of trust (or distrust) established, it can be downright job-threatening. We considered it the responsibility of the Information Systems and Quality Assurance Departments to educate and interact with all levels

of management and employees on a platform on which they could comprehend and appreciate the nature and complexity of the work we perform. We would be required to *earn* their respect!

Thus was born the cross-functional, interdepartmental Software Systems Team. What seemed like a logical answer to a desperate situation proved initially to be complex and cumbersome to execute. Employees were recruited from each department and all levels of management. A flowchart was created to that outlined the myriad steps required for a system modification/enhancement to move from the arena of "...gee, that'd be nice" or "...the system blew up last night" to being released back into production. Education was basic, such as the difference between a "system modification" and a "system enhancement." We gingerly made the point that our responsibility, IS and QA, was to work on whatever task came our way. However, the executive and management staff might want to consider prioritizing their system needs in accordance with the overall corporate goals --- our work sequence could then follow the defined business critical needs. IS and QA contributed to this prioritization process with our estimates of manhours and resources needed to complete each task.

The focus then turned to implementation of the process. Once management prioritized system needs in accordance with business needs, the actual work in the QA arena began. Management was made to understand that in order for the various departments to receive the expected end product, "user" input and participation was heavily needed in most phases of the process. "User" was defined both as the person requesting the system change, and the person(s) required to manipulate the changed end product. The earlier the users' and QA's involvement, the more closely we achieved expected results and timely delivery.

Once a work request was defined, the Education and Training Department was invited to meetings. Their early involvement was further insurance for a prepared workforce when implementation of the change was affected.

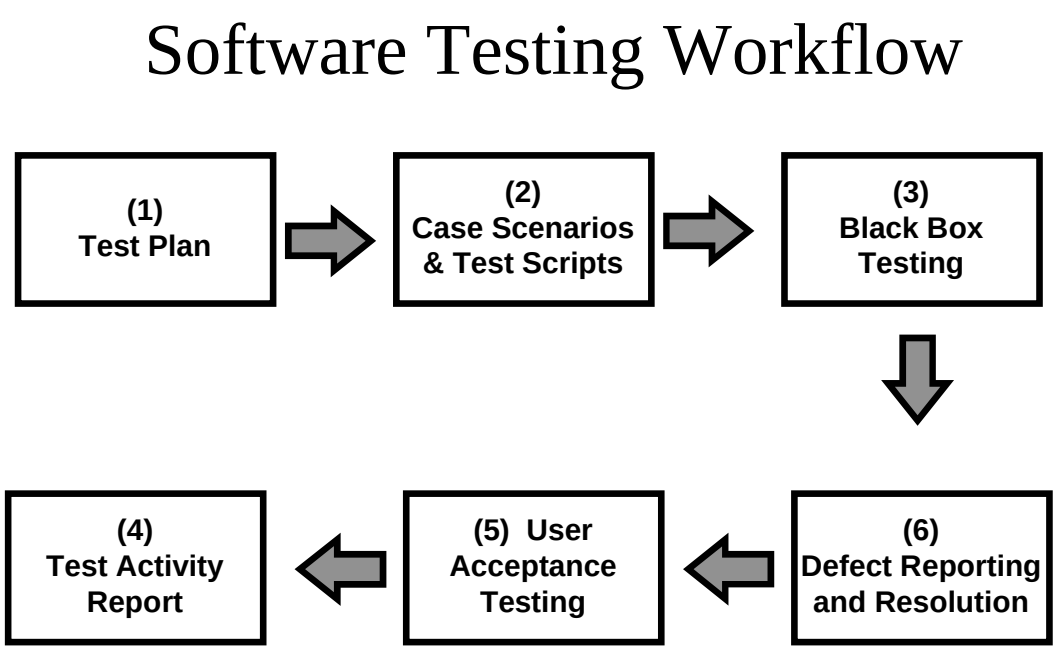
A Brief Overview of the Software Systems Team Process:

1. System change is requested, approved and prioritized.
2. QA creates User Specifications Document and submits to IS developer.
3. Developer designs, codes and obtains peer code review.
4. QA executes preliminary "black box" testing.
5. QA requests Users' assistance with "acceptance testing"
6. Training sessions conducted as needed.
7. System change released into production.

Software Testing Workflow

The QA Team's involvement begins early in the process. In conjunction with the user, we are now responsible for creating the User Specifications Document. These documents range in complexity from a simple paragraph to a multi-page document with screen prints illustrating and detailing the requested changes. As liaisons, the QA team members have learned to effectively translate the needs of the users to the developers. The more experienced the developer becomes with our particular work processes, the easier this process. Often, the developer is called into preliminary meetings to affirm that a requested change hovers within the realm of reality and possibility . . . a time-saving measure. Of course, we all know that developers are much like salespeople . . . and that all things are possible! One must however, continually remind the user (and the manager) of the potential "costs" of those achievements.

QA submits the User Specifications Document to IS and begins preparing the test plan. Should IS have questions regarding clarification of requested changes, both the QA Analyst and the user(s) are in for further discussion. Upon completion of coding changes, IS forwards the module to QA and the real work of software "testing" begins.



1. **Creating a test plan.**
   The complexity of the test plan (like the user specifications document) depends on complexity of the work request (system change). It typically includes an introduction detailing relevant background information and affirming the need for the system modification. It includes testing objectives, processes to be tested, and expected results. Often, the test activity report alone serves as documentation of simpler testing efforts.

2. **Creating case scenarios and test scripts.**
   Case scenarios and test scripts are named and filed according to processes. Our manual system is sweet and simple. Test scripts are continually maintained and updated according to the ongoing system changes --- that part's not so simple!

3. **Black Box testing to resolve initial defects.**
   Once the system changes have been moved to the testing platform, preliminary testing begins. Fortunately, most of our testing team has battlefield experience with the workflow processes, having worked with them since corporate inception. Our team is expected to maintain their knowledge of workflow and software system changes on an ongoing basis.

4. **Incident / defect tracking and resolution.**
   Our system of defect tracking and resolution is currently a manual one. Once again, simplicity is the key. A screen print is made of each defect, which is then reported to the developer who made the coding modifications. Thanks to Myers-Briggs, and our greatly enhanced interpersonal skills, celebrations by "QA Code Busters" remains a private event. We are compelled to respect the personal investment each developer makes as coding modifications are executed. However, QA is continually reminded that our job security is greatly enhanced whenever defects are noted *prior to* rather than *subsequent to* release of the modified product. The original defect report is retained in QA (they have a mysterious way of getting lost in IS) and the developer signs off on a copy once the code correction has been made. All documents are maintained in the work request file.

5. **User Acceptance Testing.**
   We have learned the hard way that USERS are our best comrades in the battlefield of software testing! Involving users in all phases of the software modification process provides them with an exceedingly strong sense of ownership regarding the modified end product. Therefore, very few work requests are ever released back into production without a minimum of "user" inspection. The resulting dialog among employees, when defects are found *after* release, becomes "...OUR problem (Users, QA & IS)" instead of the finger-pointing "...YOUR problem (QA)." Other statements like "...my goodness, how did WE ALL miss that in testing?" can be heard.

6. **Creating the final Test Activity Report.**
   This report finalizes and documents the testing effort; again, it's length depends on the level of effort. All reports do contain a minimum of the following elements:
   6.1. A description of the requested change.
   6.2. The particular objectives of testing.
   6.3. A reference to noted defects (with copies of details in file).
   6.4. A statement of completion (passed, ready for production).
   6.5. An accounting of resources used: employees, peoplehours, etc.
   6.6. Signature of approving parties.

Insidious Schemes versus Professional Strategies

Much like the dreaded practice of "office politics," one must determine the appropriate timing and use of insidious schemes versus professional strategies. Of course, when caught in the act --- one must never admit to stooping so low as to employ anything other than professional measures to prove one's point.

"To test or not to test?" That is the question. Scenario: Only one line of code is changed. (Been there, done that!) The modification is released back into production --- no testing is performed. What happens next, more often than not, is a system problem. You can believe, when asked how these "defects" got by our QA Team, we plead complete ignorance! We apologize profusely, assuring the affected department that QA would certainly have included their users in any testing efforts . . . and we leave IS hanging out to dry. One might consider that a cold response; however "...my most *remembered* lessons were created by my biggest mistakes." Somehow, most of us seem destined to learn the hard way!

Manipulation or cooperation? It is undeniably in everyone's best interest to "build relationships" throughout the corporation, on all levels, at every juncture. This is a positive, proactive method of "playing the game" versus the age-old synonym of "brown-nosing," a self-serving, nonproductive activity. Building relationships is a two-way, win-win method of accomplishing corporate goals and objectives . . . from individual employees, through middle management, up through the executive level. It's a whole lot easier to call in your chits if you were there first to assist others in the line of fire. Our call to users for assistance in the acceptance testing phase is regarded as beneficial rather than as bothersome.

Won't you let me help you? QA is in a unique position, our departmental goals are hardly self-serving. To the contrary, our goals are to assist other departments in attaining their goals . . . most often through successful system modifications which improve their workflow processes. The more QA interacts with managers and employees to attain their goals, the more our contributions are acknowledged! The earlier we become involved in the process, the more likely we are to be acknowledged in the successful completion and attainment of those goals.

Batman & Robin versus The Jokers! My exposure to the software testing industry educated me to the concept of the developer as the *eternal optimist* --- "Of course that coding is perfect / correct/ brilliant! The developer created it, didn't she!" And to the tester as the *eternal pessimist* --- "It's a tough job, but somebody's gotta break it!" As employees desiring job security, we might consider maintaining those perspectives. However, respect for each discipline's frame of reference is imperative. The tester must recognize the inherent personal contribution the developer makes in creating code. The developer must recognize the duty and responsibility of the tester to break that code!

The Winning Plan for Success

Due to the nature of the software development life cycle, the main responsibility of our QA Team - testing - often finds itself at the end of the production line. Our team's reputation at LWCC has been developed and earned over these past years by bringing knowledge of our unique software application and hard-earned testing experience to the modification and enhancement processes. We continually build our reputations as *professional service providers.* Our goal: timely delivery of a quality product. Our internal business partners are constantly (and frustratingly) reminded that QA's early and continual involvement benefits everybody.

Do we have all the answers? Certainly not! Even the Myers-Briggs experience did not eliminate the need for further training in conflict resolution skills and negotiation skills --- which moved us another step forward in civilized corporate interpersonal interactions. It also led to our success in constructing the "two-faced ally." Any software tester worth his/her salt knows that it's no easy matter to keep both the developers and the users happy. While we acknowledge the innate right of the user to make change requests, we contend that developers have a right to be informed of ALL (most) of the requested changes prior to the initiation of code modification. *(We can dream, can't we?)* Meeting deadlines is contingent upon complete user specifications. Any changes require signed addenda to the User Specifications Document!

Respecting and adhering to established workflow procedures helps ensure efficient and effective use of people power and resources. Departing from established procedures often results in confusion, costs overruns, frustration, and less than adequate delivery of the modified product. Our QA Team works to continuously improve our own internal workflow processes. We strive to remain flexible! We work closely with the Education & Training Department to provide the users with current information regarding the continuing changes to the software system and to the workflow processes.

And let's not forget to celebrate our accomplishments! Budget for your celebrations. And celebrate the milestones as well as the project's completion. So often, we become mired in the mundane, stressful and endless tasks that keep us gainfully employed. We forget that public praise and celebration is important and meaningful. Homo Sapiens often respond to the most basic of human needs as defined in Maslow's hierarchy. That's right . . . food! When monetary reward is limited, feed the people! Be it pizza brought into the office, breakfast in our corporate cafeteria, or ice cream snacks at 2:00 p.m. in the afternoon. Our employees respond to almost any level of acknowledged appreciation. And don't forget to include supervisors and management in these celebrations; they are usually footing the bill!

How does any team attain success? Education and respect. We should choose to educate each other to the responsibilities and requirements of our individual disciplines. We should choose to behave more respectfully in executing those responsibilities, particularly when interacting with others. The result: rather than behaving as passive "victims" of our work environments, we learn to behave as proactive "professionals" and "victors" --- adding value to our work lives and contributing to our corporation's bottom line.

Transferring Software Development Best Known Methods between Generational Product Lines

James R. Bindas

Intel Corporation
JFT-101, 2111 NE 25th Avenue

Hillsboro, OR 97124

Phone: (503) 264-8869

james.r.bindas@intel.com

Biography

James R. Bindas is a Software Process Engineer with Intel Corporation in Hillsboro, Oregon. His tasks include working with the software development community to help standardize and improve software development processes. James has been working with Intel for eight years in various software roles ranging from software tester to project leader. Before joining Intel James worked for RCA/GE Solid State and Harris Semiconductor as a Quality Assurance Technician. James holds a Master of Science degree in Computer Science from Steven's Institute of Technology and a Bachelor of Science in Graphic Communications from California University of Pennsylvania.

Abstract

Launching a new product line is a difficult and time-consuming task under the best circumstances. While working on a new project can be refreshing, difficulties arise in allocating sufficient resources to create and tune the new product development processes. Modifying the processes or Best-Known Methods (BKMs) of the past is one technique employed to allocate time and resources more effectively.

Transferring BKMs from an earlier product line into the current one can bring about its own set of challenges which include: defining the BKMs, producing repeatable BKMs to be deployed in varied environments, implementing techniques to log the BKMs, and placing the BKMs in a central location.

This paper discusses one approach to transfer software BKMs between generational product lines. The topics addressed include:

- Defining the project's mission and goals.
- Creating definitions and processes to support the BKM transference.
- Encountering and solving issues while implementing the processes.
- Addressing the human dynamics issues involved.

Introduction

Many words in the English language have multiple definitions with similar meanings. A prime example is the word "quality." The term Best Known Method (BKM) also falls into this category. The difficulty in defining BKM surfaces when attempting to phrase the composition, implementation, and certification. Some persons would use another term, but other phrases cannot rival the perception and awareness of "BKM." This paper addresses how to succeed in the challenging environment of transferring BKMs to another party.

My department is responsible for the design, development, and delivery of Electronic Design Automation (EDA), Computer-Aided Design (CAD) tools to internal customers - the Intel microprocessor design groups. Our department employs over 500 employees at four major sites to support the quickly evolving microprocessor roadmap. The department is divided into two layers of cascading development teams located in Santa Clara, California; Hillsboro, Oregon; and Haifa, Israel. We refer to these groups as generational teams. As one development team is maturing (Team-B), the other group (Team-C) is preparing to deliver solutions for the next generation of EDA tools. With the emphasis on tighter time schedules, pressure is placed on devising Team-C to produce high quality solutions, with minimal start up costs.

In order to adhere to the tight schedule, an approach adopted was to reuse Team-B's software development processes, some of which were considered to be Best Known Methods (BKMs). While the idea is logical, no process "existed" to support the transfer. A common BKM definition and transfer infrastructure was needed between the two generations to support this effort. This project was assigned to our team, which is outside both generational organizational structures and is charted with managing several cross-organizational, tactical improvement projects. Managing the effort was my responsibility.

Preparation

The project began in March 1997 by researching Team-B's software development processes, their current status, their documentation, and their utilization. Team-C followed by identifying which processes they required. A majority of the research had been performed by previous improvement projects; thus the job was a straightforward procedure to gather data, sort the information, and verify it with key generational process personnel. An added bonus was the early identification of key personnel with expert knowledge who were interested in championing the processes.

Forming a Working Group

The best method to gain commitment from a team is to include them early in the process. In late May, early June 1997, key personnel from both generations were invited to begin the project. Obtaining a commitment from team members was not a problem since members had their own personal reasons to support the project. For example, the return on investment (ROI) for Team-B included:

1. An opportunity to document their own processes.
2. A basis for training new hires.
3. A legacy on which to base future processes.
4. An opportunity to introduce improved Team-C processes into Team-B.

For Team-C, the ROI was straightforward:

1. Saves time and effort from developing new and untested processes.
2. Provides a single source for information on Team-B processes.
3. Provides a vehicle to improve upon lessons gained from implementing the process in previous projects.

Establishing a joint working team provided a forum to level expectations on both sides early in the process.

Defining the Working Group

In any good working group, the first item of business is to define the mission, objectives, and definition of success. The team agreed to limit itself to a four-month scope with short-term, well-defined goals because:

1. The group's focus often varied during a long-term project.
2. Management priority sometimes shifted due to uncontrollable forces.
3. Team members tended to leave over time, due to work assignment changes.

While preparing the mission statement, the group discussed whether they should monitor the compliance level

BKM Validation and Certification Team Charter
Rev 2, 6/16/97

- **Mission Statement**
  - **Develop a basic infrastructure to support the transfer of BKMs from Team-B to Team-C.**
- **Objectives**
  - **Create a basic pilot BKM transfer methodology.**
  - **Review and document any lessons learned from the experience.**
- **Definition of Success**
  - **Document methodology of a repeatable BKM transfer.**
  - **Two BKMs or KMs transferred using the above documented methodology by the end of 3Q97 (A total of six by end of 4Q97).**

Figure 1

of each BKM after it was transferred. The group decided against this, because it was a factor outside of their control. However, the team was able to control building an infrastructure that allowed BKMs to transfer between groups, where success could be measured. The final objectives included creating a repeatable process and documenting lessons learned (**Figure 1**).

Defining the Methodology

With a common purpose and goal established, the next step involved defining common terminology. BKM was the first term to be defined with its many meanings and applications. A common reaction was to create a new standard definition. After consulting a resident BKM expert, the team agreed they would have difficulty in settling on a single definition. Since the term BKM is common in people's minds, the introduction of a new term would never be accepted. The best that could be accomplished was to define BKM in the context of the Team-B/Team-C software development environment. Surprisingly, the team quickly and painlessly agreed to the following definition:

A Best Known Method (BKM) is a portable package of methodologies and supporting materials that demonstrate improved results over other known methods to solve specific problems/issues in a given environment. A BKM is composed of the following components:

- *Owner(s)*
  - The expert(s) for this BKM.
- *Description*
  - Brief description of the BKM, including its purpose.
- *Entry/Exit Criteria*
  - What must happen and/or what must be available before the procedures associated with this BKM are started, and what must happen and/or what must be available before this BKM can be considered completed.
- *Procedure*
  - How the BKM is implemented.
- *Supporting Materials*
  - Any tools, templates, scripts, references and training that are needed to provide descriptions and on-line locations.
- *Deployment History*
  - Information on when and where the BKM was deployed.
  - Metrics and/or data from previous projects can be used to measure the BKM's effectiveness or estimate future expectations.
  - Examples of past experiences.
- *Lessons Learned*
  - Documented experiences of previous BKM users.
  - Descriptions discussing why certain paths were chosen in the past.
  - Thoughts about what we might do differently next time?

Note: Until the BKM is "Certified," we will refer to it as a KM or Known Method.

Two key points are included in this definition. The initial point is that a BKM is a portable *package* of not simply methodologies, but supporting materials as well. The package includes the tools and procedure (**Figure 2**, x and y-axis's). The second key point is that the BKM documents experience (**Figure 2**, z-axis) which makes the BKM much more usable.

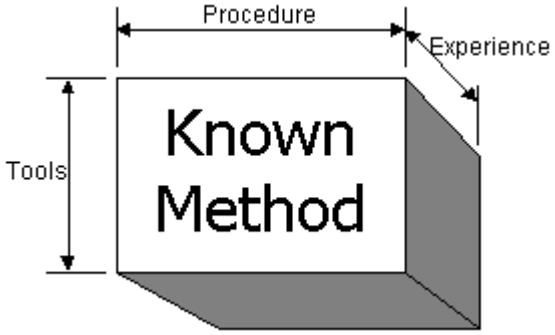


Figure 2

The next step was to create an infrastructure to support the transfer process that resulted in creating **Figure 3**. The process began simply - select the KMs to be transferred, package the KMs, verify, and certify them. Early issues arose, such as:

1. A person, familiar with the process, should package the BKM because they are in a position to ask the right questions, and include the correct components in the package.
2. Team members discussed the difference between validation and certification, suggesting that this could be a duplication of effort. The team replaced validation with the End-User Forum (defined later). Certification would be determined if the KM customer could implement it into their infrastructure.
3. Team members expressed concern about whether the KM could be certified as a BKM. Certain Known Methods are valued for their lessons learned, but not necessarily the best methods to follow. A method was needed to separate KMs from BKMs allowing both types to transfer.
4. A drive to keep the infrastructure simple existed. Too often, good processes failed because of high overhead in maintaining a complex model.

After concerns were addressed, the resulting model was formed including four key points:

1. The model was generic allowing any generation to transfer KMs to another.
2. Expectations were addressed initially by agreeing to a common definition and requirements between Team-B and Team-C, before packaging began.
3. During the packaging stage, an End-User Forum was held between Team-C and Team-B. During this time, actual Team-B End-Users provided real-life feedback about the KM; Team-C had an opportunity to ask insightful, key questions.
4. Some KMs offered valuable lessons learned, but were in such condition that they could not be used in the receiving generation infrastructure. Therefore, the method was not certifiable and was transferred to the receiving generation as a KM. The KM would then be examined and a new methodology will be constructed.
5. Certification is performed by the receiving generation, based on a set of well-defined criteria **(Figure 4**).

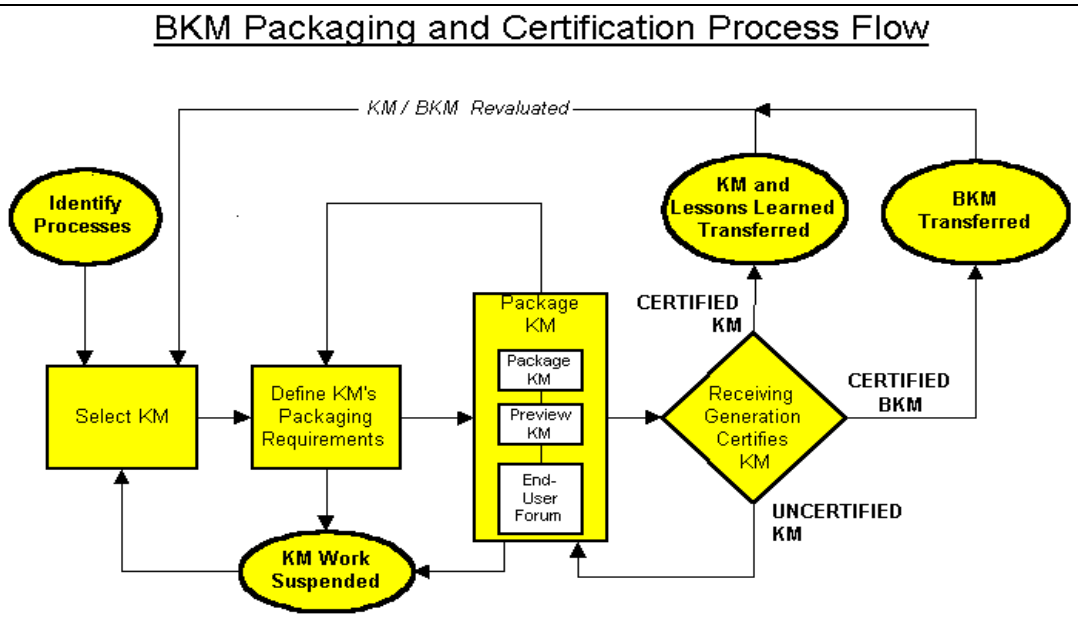


Figure 3

Implementation

One of the greatest challenges of the project was implementing the process after creating the methodology. In order to accomplish this, the team initially agreed to the high-level processes and definitions before starting implementation. This yielded a common framework to build upon, allowing the lower-level detailed processes to be specified as each step of the high-level processes was completed.

Implementation work started in late June 1997 with two KMs -

1. Software Documentation Process.
2. Software Development Life Cycle.

The selection process was kept simple because a high methodological-based approach would not produce a high ROI to justify the overhead. The KMs selected had two common attributes:

1. Team-C required them.
2. The KMs were well documented since the infrastructure was new and untested.

The second phase of the process, Defining KM Packaging Requirements, began early to mid-July. The process itself proved to be enlightening and profitable. The team soon realized that assumptions made by both Team-B and Team-C were not the same. If expectations were not leveled early, they would have led to wasted effort, and conflict within the group. This problem was solved when Team-C realized that Team-B's Life Cycle would not bring as high an ROI as previously expected. Software inspections were substituted for the Life Cycle KM.

The packaging began in early August and ended mid-September. Using the BKM definition (**Figure 4**) as a guide, various BKM components were identified and gathered. The next step was to create an hour-long (15-25 slides) presentation, describing the different components in the BKM definition and to answer questions about the KM. This presentation, or management summary, was presented at the End-User Forum in late September.

Owners were identified and assigned to package the KMs. A common packaging location was needed to act as a KM repository. The main, and perhaps only, common information link across ten time zones was our Intranet, called InfoNet. A web site was designed to be the repository.

BKM Certification Definition		
The process that ensures the BKM is in a state that can be implemented into the receiving generational infrastructure with minimal effort.		
BKM Certification Criteria		
Certified BKM	***Certified KM***	***Uncertified KM***
• Recommended by the packaging team with a minimum of changes. • Packaging is comprehensive. • Completed two cycles - Product Release or Tape Out. • Uniformly accepted and deployed by the sending project.	• Recommended by the packaging team as a set of learned lessons from implementing the method. • Packaging is completed as much as possible. • Not completed two cycles. • Not uniformly accepted or deployed by project.	• Missing or ambiguous KM components. • Out-of-date components that need to be updated.

Figure 4

The End-User Forum planning proved to be non-trivial. The plan was to keep the forum to a small, informal group of less than 15 people. The group was designed to represent a wide selection of vocal people department-wide. Getting participation for the End-User Forum was easy due to pre-selling the forum. The main dilemma was scheduling. Finding an open timeslot for 15 people proved to be a non-trivial task. Finally, after two postponements, the two and one-half hour End-User Forum was held on September 23, 1997.

A week before the forum, the team mailed the formal invitations, agenda, and sample questions alerting the members to the types of questions to be asked. The agenda was simple - present the slides and field questions from the forum.

What was refreshing about the forum was that new ideas and information were exchanged liberally between Team-C and Team-B. Although the discussions often continued on tangents, the participants offered undocumented information that only few had. This information was repackaged into a KM for future use.

The certification process was scheduled to start on October 30, 1997. Plans included splitting KM certification into two half-hour sessions. At the first session, an abridged version of the End-User slides was presented. At the second session, held weeks later, specific questions were raised. At this time, the certification team formally approved the two KMs as a BKM.

Next Series of KM

With the BKM infrastructure in place, emphasis was shifted from creation to KM transfer. Originally, four KMs were to be transferred:

- Build and Release Process.
- Unit Test.
- Flow Test.
- Methodology Development.

Because knowledge of our activity had spread, we received a request for a fifth KM: Project Estimation.

During the KM Definition, both generations discovered the need to package a "Build and Release KM" disappeared which resulted in this KM being placed on hold.

Both "Unit and Flow Test KMs" are currently being packaged and an End-User Forum was held in June 1998. Again, the End-User Forum discussion was lively and informative. Unfortunately, due to schedule conflicts, the KM will not make it on the certification team's agenda until August.

Both generations initially agreed that Methodology Development was a potential BKM. During packaging, team members discovered it would be best to package Methodology Development as a KM. The End-User Forum was held in November 1997. Both generations were eager to share information. A follow-up meeting was agreed to share data, which would not be available until first quarter 1998. The follow-up meeting was held in April 1998, and the KM was presented to the certification team in June 1998. Further discussion was held once again, regarding the KM certification level. In the end, the team determined the KM should be accepted as a "Certified KM," due to the process needing more maturing.

Project Estimation was agreed to by both parties, but was also delayed a few months due to a product release. Project Estimation was eventually packaged and an End-User Forum was held in January 1998. Project Estimation was then certified as a KM, because it was not widely accepted and deployed by Team-B.

Lessons Learned

PACKAGING AND AVAILABLE RESOURCES: Two KMs were committed for packaging, and resources were allocated for the next quarter due to higher priorities. As it turned out, other priorities quickly appeared and supplanted the packaging effort further down on the priority list. This resulted in lost momentum and little effort being allocated for packaging.

SCHEDULING CONFICTS: Three KMs were delayed due to fourth quarter 1997 holidays and again due to product releases. Again, June and July 1998 scheduling furthered delay certification. These conflicts further slowed the momentum almost to a halt.

DEVELOPING CHECKLISTS: This lesson was discovered during KM certification when one KM was nearly denied certification due to confusion regarding the criteria. Looking back, unspoken assumptions, held by some of the certification team members, caused this confusion. During the next round of KM definitions, a checklist was used to unmask any hidden assumptions.

ENCOURAGING FACE-TO-FACE END-USER FORUMS: Because our department is multi-geographical, face-to-face meetings are not necessary for certain people. However, the receiving group needs to be present in person, to fully engage in the exchange of information. Some forums had problems with various attendees being heard or recognized on the telephone conference call. Technical problems prevented one attendee from participating in the forum. When the participants were drawing on the white board, a person on the phone line was trying to follow the conversation, based on verbal clues. These difficulties could hamper the knowledge transfer process, as well as, cause miscommunication between the teams.

KM PACKAGING PARTICIPATION: The Haifa, Israel representative departed Intel, and was never replaced. Because of this, the KM did not become implemented in Haifa

Conclusion

Often overlooked in a vertical organization, is the difficulty in sharing process improvements between teams. Unless action is taken, an improved software process cannot be developed and will end with the current product line, simply to be recreated in the succeeding product line bearing the same startup pains. One response to break the barriers between product generational teams is to ensure improvement upon this procedure (**Figure 5.**)

Was the project a success? Yes, if judged by the pre-determined "Definition of Success." This definition states the team should have a documented methodology enacted and three KMs transferred. Three additional KMs should be transferred in the near future. The project's success can be evaluated by Team C's use of five-out-of-six KMs. One KM, that was placed on-hold, indicated that the process was preventing work from being performed on a non-valued KM. The project's short-term success appears to be growing into a long-term success as time progresses. This success is not measured by active processes, but by people outside of our team asking to use the transfer process for their own purposes.

Quick BKM Transfer Process Overview:

1. Research Team-B's process for their status and utilization.
2. Research Team-C's needs process requirements.
3. Identify both Team-B and Team-C's stakeholders and form a Working Group.
4. Develop the Working Group's
   a. Mission Statement
   b. Objectives
   c. Definition of Success
5. Working Group develops a methodology by defining:
   a. BKM Definition
   b. BKM Certification
   c. BKM Transfer Flow
6. Team-C selects a KM to transfer.
7. Team-B researches the KM to determine feasibility for transfer.
8. Team-B and Team-C meet to discuss KM transfer requirements.
9. Team-B's owner packages the KM.
10. Team-C's receiver previews the KM to ensure meeting requirements.
11. Working Group sets up a centralized and accessible BKM/KM repository.
12. Working Group sells End-User Forum to prospective Forum attendees.
13. Face-to-face End-User Forum is held.
14. Team-B repackages KM with information taken from the End-User Forum.
15. Working Group presents KM to the Certification Board.
16. Certification Board certifies the KM as a BKM

Figure 5

Acknowledgment

The success of this project is due to the entire BKM team. Each member contributed his or her time, energy, insights, experience, and knowledge to this project. Three Intel software engineers should be specifically mentioned -

- Doron Becker
- Greg Hannon
- Tamar Yehoshua.

Anticipating and Mitigating The Professional Challenge to Independent Verification & Validation

James B. Dabney
Intermetrics, Inc.
Houston, TX 77058

James D. Arthur
The Department of Computer Science
Virginia Tech
Blacksburg, VA 24060

Abstract: Independent Verification and Validation (IV&V) faces three classes of challenges: the Technical Challenge, the Management Challenge, and the Professional Challenge. In this paper we focus on the Professional Challenge, and in particular, the four phases that characterize it: Denial, Anger, Cooperation, and Dependence. We believe that to implement an effective IV&V effort, one must understand the relationship among the phases and the critical issues underlying them. For each of the phases we (a) provide a characteristic description, (b) discuss how they affect the IV&V effort, (c) present representative issues and examples, and (d) describe steps to reduce the adverse impact of the three detrimental phases. The examples provided are those we have encountered while serving in an IV&V capacity; "lessons learned" guide our suggestions for addressing phase-specific issues.

Keywords: IV&V, Verification, Validation, Software Engineering

Author biographies:

James B. Dabney received the B.S. in Mechanical Engineering from Va. Tech, the M.S. from University of Houston, Clear Lake, and the Ph.D. from Rice University. He has been involved in IV&V of mission critical systems for nine years, working on Space Shuttle and International Space Station onboard flight software and critical ground-based software projects. His research interests encompass IV&V, process control and optimization, and computer aided engineering. He is employed by Intermetrics, Inc. in Houston, Texas. Dr. Dabney can be contacted at Intermetrics, Inc., 1100 Hercules, Suite 300, Houston TX, 77058. His e-mail address is jdabney@rice.edu; his telephone number is (281) 480-4101.

James D. Arthur received the B.S. and M.A. in mathematics for the University of North Carolina, the M.S. from the University of Houston, and the M.S. and Ph.D. from Purdue University. He has been involved in IV&V research for the last ten years. His research interests encompass IV&V, software quality assessment, user environments, and parallel computation. He is currently an Associate Professor of Computer Science at Virginia Tech in Blacksburg, VA. Dr. Arthur can be contacted at Virginia Tech, The Department of Computer Science, 660 McBryde Hall, Blacksburg VA, 24060. His e-mail address is arthur@vt.edu; his telephone number is (540) 231-7538.

Anticipating and Mitigating The Professional Challenge to Independent Verification & Validation

1.0 Introduction

Independent Verification and Validation (IV&V) has a unique status within the software engineering community because, at any given instance in time, its activities can be viewed as impeding or supporting the software development effort. Because of this perceived duality, attendant challenges faced by the IV&V agent are often multi-dimensional and require a thoughtful, proactive response rather than the conventional, often reactive one. In performing IV&V for a variety of programs at NASA, we have observed that challenges most often take on one of three forms: technical, management, or professional. To date, IV&V literature addressing such challenges has focused primarily on those issues stemming from the technical and management perspectives. In this paper, however, we concentrate on the professional challenge: what it is, and how to deal with it. We will begin by presenting a brief definition of IV&V and the three key participants in the IV&V process. Next, we will provide an introduction to each challenge; the remaining sections expand on the professional challenge.

1.1 The IV&V Process

There are many definitions for software IV&V, but most of them provide similar characterizations (AFSC 88, Arthur 96, Neal 97, Rogers 81). In general, we can say that IV&V is verification and validation (V&V) performed by an agent who is independent of both the developer of the software and the customer paying for the software development. To explain the role of IV&V, we will first define software V&V, and then provide some comments on the concept of independence.

V&V is a process which seeks to ensure that software is correct and suitable for its intended purpose by analyzing and testing key artifacts of the software development process at each stage in the development cycle (AFSC 88, Rogers 81, Wallace "Standards" 89, Wallace "Overview" 89). Among these artifacts are software requirements, design, and test documentation, software prototypes, source code, executable code, and test input and output data. The two components of V&V are defined as follows (AFSC 88):

- *Verification*: The process of ensuring that the output of each phase of the software development process satisfies all requirements of the previous phase, is internally complete and consistent, and can serve as requirements for the next phase. Thus, verification is concerned with correctness and consistency.

- *Validation*: Actions taken to determine whether the final product fulfills its intended purpose. Validation approaches software from a system-level perspective. Thus, validation is concerned with suitability to a purpose.

IV&V is the application of V&V methods and procedures by an agency not affiliated with either the developer of the software or the customer purchasing the software. IV&V is usually mandated and paid for by the customer to help improve the quality and reliability of the software product and to increase the customer's confidence in that product.

A project in which IV&V is employed has three principals: the organization procuring the software system (the customer), the software developer, and IV&V.

- *Customer*. In general, we use the term customer to refer to the company or government agency that is procuring the software. More specifically, however, the term customer is used to identify that person in the purchasing organization who is responsible for technical oversight of the software procurement, and often for the acceptance or rejection of the final product.

- *Developer*. The developer is the company or agency responsible for producing the software system. In general, the developer is responsible for the majority of the testing needed to show that the software satisfies all requirements.

- *IV&V Agent*. An IV&V agent is a company or government agency tasked only with the responsibility for V&V. An IV&V agent is independent of both the developer and customer, but works on behalf of the customer (Rogers 81). The IV&V agent performs analysis in parallel with the V&V performed by the developer; the presence of IV&V relieves the developer of no responsibility for V&V or accountability for correctness and suitability of the software product. Independence is believed to allow the IV&V agent to offer unbiased assessment of the software project and its product (Easterbrook 96).

Each of these three principals has an interest in ensuring that the final product operates correctly and fulfills all other requirements. We have observed, however, that developers frequently perceive the IV&V agent as a rival, leading to an adversarial relationship between the developer and IV&V. It is this adversarial relationship that leads to the professional challenge.

1.2 Overview of the Three Challenges to IV&V

The challenges that face IV&V can be characterized as belonging to three classes: technical, management, or professional. In the next three paragraphs we will briefly define each class of challenge.

The Technical Challenge is to develop and execute a strategy that applies the proper methods and tools of IV&V to derive the greatest benefit. While each IV&V project is different, both IV&V theorists and practitioners agree on much of what is important (AFSC 88, Leveson 91, Leveson 92, Neal 97, Rogers 81, Wallace "Overview" 89, Wallace "Standards" 89). The principal objective in meeting the technical challenge is to locate errors as early as possible in the development process, and subsequently, minimize the effort and cost of removing them (Arthur 96). This objective places most of the IV&V emphasis on early lifecycle activities. While we have certainly made progress in meeting the technical challenge, there is still much room for improvement. In particular, we need a more automated process for and supporting tools to assist in the verification of requirements, design documents, code, and the testing process. Moreover, we also need to address ways to achieve V&V using the rigor of formal methods, but at reasonable costs. Research described in (Cheng 93, Easterbrook 97, Easterbrook 96, Gerhart 94, Rushby 95, Rushby 93) shows that progress is being made on this front and offers hope for better tools and methods.

The Management Challenge can be stated most succinctly as "How do I deal with the paradox that the people most able to perform IV&V are least likely to want to do it?" IV&V is a difficult technical activity that requires talented and experienced analysts. Unfortunately, people who possess the necessary talent usually prefer to apply their expertise to designing and building software systems, rather than to verifying and validating someone else's work. Because this is a problem of perception and personal preference, it does not lend itself to an easy solution. In his work with organizational cultures, insights offered by Westrum (93) are helpful, but also underscore the need for additional research in this area.

The Professional Challenge is often the most difficult to overcome. The basic issue is "How do we deal with the natural tendency of the development community to view IV&V as the enemy?" Past experiences and observations lead us to the conclusion that the professional challenge is composed of four distinct phases. Each phase represents a definitive change in the developer's attitude toward IV&V: (1) Denial; (2) Anger; (3) Cooperation; and (4) Dependence. Every IV&V project has the potential to exhibit characteristics inherent in all four of these phases; to date, every project we've participated in has involved all four. In the following section we present each of the four phases and discuss how one can effectively deal with the problems they impose.

2.0 Phases of the Professional Challenge

2.1 The Denial Phase

Experience suggests that the denial phase is the most difficult one to mitigate. In the denial phase, the developer rejects the need to cooperate with IV&V, and refuses to provide them with the information needed to perform V&V tasks. Initial contacts with the developer often result in a litany of excuses justifying non-cooperation. Some of the most common claims we've heard are provided below.

- "My contract does not require me to cooperate with IV&V."

  This is almost always claimed, even if the contract or statement of work explicitly requires support of IV&V. In this situation, the customer is the only one who can effectively address the problem and force resolution to the impasse. If the customer takes the steps to compel the developer to cooperate with IV&V, mutually beneficial information exchange can be expected. Unfortunately, we have found that this does not always happen. We have worked on IV&V tasks in which the only information given to us was forwarded from the developer through the customer - no direct contact with the developer was permitted. Unfortunately again, this approach severely limits the effectiveness of IV&V, and in particular, adversely impacts IV&V activities in the early lifecycle phases where the more significant benefits can and should be realized.

- "Cooperating with IV&V will consume too much development resources."

  This is generally a rationalization type of argument. The most important step the IV&V agent can take to counter this argument is to ensure that the IV&V effort is well planned and structured so as to minimize any adverse impact on development schedule and resources. It is also helpful to point out that resources for IV&V activities have already been allocated and factored into the development plan.

- "The IV&V contractor doesn't have the expertise to understand our project, and we don't have the budget to train them."

  Again, this claim must be false. If it is true, the person(s) tasked in the IV&V role should not be there in the first place. However, we are also of the opinion that IV&V does not necessarily require an *in-depth* knowledge of the particular project from the outset. We have found that adequate project expertise develops rapidly. More important than project expertise is expertise in the practice of IV&V. In fact, the proper IV&V mind-set is not "how would I do this," or "could I do this better," but rather, "is the developer doing the right thing right." Of course, if the IV&V agent requires no training from the developer, there is no budgetary problem.

- "Our processes (or code, or some other things of value) are proprietary. We can't risk divulging this valuable information to a potential competitor."

  This response is most often related to and offered as a supporting claim for the first argument. If such is true, in actuality or perception, a non-disclosure agreement must be arranged.

The above claims characterize a developer who resists meaningful dialogue with the IV&V agent. We must also be aware, however, that just because we can interact with the developer and exchange information, it does not necessarily follow that IV&V can or will proceed unimpeded. For example, we have also encountered developers who simply refuse to address any issues raised by IV&V. This is particularly true for requirements validation issues. That is, issues in which the IV&V agent claims that the software requirements are not the right requirements -- and in effect, that the developer is building the wrong system. Because requirements validation issues raise concerns as to whether the customer is actually getting what he/she expects, it is the customer who must assume the position of arbiter and make the final judgment call. Experience has shown that if the customer takes the IV&V issues seriously, then the developer will begin to take them seriously, too.

In the above paragraphs we have outlined some of the most often encountered arguments stated during the denial phase. In the following paragraphs we offer several suggestions that we've found effective as the IV&V agent in preempting contentious and obstructive arguments encountered during denial phase:

- *Don't get angry* - expect direction and final adjudication to come from the customer. Also, consider the developer's perspective. Many people view IV&V as the "software police." They often believe that IV&V presents a "no win scenario" where an IV&V success is, by definition, a developer's failure. If the IV&V agent begins to exhibit manifestations of anger, the already contentious situation can (and often does) degrade into interpersonal conflicts that harm the effective interaction he/she wishes to develop.

- *Make absolutely sure that any issue raised is real*. Early in his/her involvement, the IV&V agent will identify issues that turn out to be misunderstandings on his/her part. The IV&V agent must constantly guard against raising errant issues because any such mistakes at this point will only add weight to denial arguments offered by the developer.

- *Don't assume that the customer necessarily supports the IV&V efforts*. We have been involved in long term IV&V projects in which the customer didn't really support the need for IV&V, but "higher headquarters" mandated it. The IV&V agent and organization must earn the customer's respect and support, and retain it. This is only achieved by providing the highest quality service possible, and by making every effort to interact with the customer and developer in a professional manner.

2.2 The Anger Phase

The end of the Denial Phase is often accompanied by a feeling embarrassment by the developer - in effect, it is perceived as a loss of face. The inevitable consequence is anger, directed at IV&V. By anger, we mean shouting, invective, fist pounding, and emotional outbursts. We have experienced adult engineers shouting at us, calling us vulgar names, and pounding on tables -- everything short of physical contact. Our impulse in such a situation is probably normal - flee or fight. However, neither of these responses is appropriate. The key to dealing with this aspect of the anger phase is to understand that this phase really marks the beginning of a positive movement forward. More specifically, it implies that the developer now realizes that IV&V must be taken seriously. An additional fact that is encouraging and should help guide a positive reaction by the IV&V agent is the knowledge that the anger phase is usually short lived.

A second manifestation of the anger phase is an attempt on the part of the developer to get revenge. We have found this to be relatively rare, but it does happen. Revenge can manifest itself in several ways, e.g., sending incorrect information to the IV&V agent, directing the IV&V agent down the wrong path, or even self-destructive behavior. An example of the extremes to which developers will go in this regard was a case in which IV&V raised several issues with a requirements document. The development project manager retaliated by changing the scope of the document from binding requirements to "general guidelines." Such actions effectively nullified the document as a binding agreement between two parties.

2.3 The Cooperation Phase

The anger phase is always followed by the cooperation phase. Cooperation is the desired state of the relationship between IV&V and the developer. In the cooperation phase, the developer and IV&V team interact effectively, enabling IV&V to provide services that benefit all parties (Arthur 96, Neal 97, Wallace "Overview" 89, Wallace "Standards" 89). The goal of the IV&V team should be to enter the cooperation phase as soon as possible, and to stay there for the remainder of the project.

The IV&V team must be aware of and recognize those indicators marking the beginning of the cooperation phase. The team must then work hard to establish good relationships with the developers, but just as important, maintain its own independence. We have found that several practices help to strengthen and maintain the cooperation phase.

- *Keep the IV&V/developer relationship relatively formal by following documented procedures for reporting and resolving issues*. This does not mean that IV&V should not discuss issues informally with the developer. However, failing to follow the formal processes can lead to a loss of accountability and a loss of archival documentation.

- *Always be cognizant of the fact that IV&V is a technical activity, not a political one*. Do not get involved in the inevitable political disputes within and between the developer and customer organizations.

- *IV&V must always perform its own analysis and provide recommendations reflecting an informed assessment of the results*. It is tempting to discuss a potential or suspected problem with an experienced developer, and to obtain advice on how to proceed. This practice is most unwise, as the developer's advice will always be colored by (a) loyalty to the development organization, (b) feelings of ownership of the product, and (c) a desire to avoid trouble. Thus, the developer can be expected to have a tendency to minimize the significance of potential or suspected problems, and discourage further investigation.

- *Avoid training seminars sponsored by the developer*. The relationship between IV&V and the developer should be peer-to-peer. Engaging in training activities offered by the developer often leads to a perceived teacher-student relationship by some members of the development organization. This, in turn, can foster the undesirable perception of IV&V personnel being subordinate to development personnel.

- *Keep both the developer and customer well informed on the status of IV&V issues*. In particular, take time to ensure that the issue is valid, and then report it immediately thereafter.

- *Stay on schedule*. IV&V schedules must be written relative to trigger events, namely delivery of key project artifacts such as requirements documentation, source code, and test results. It is the joint responsibility of the customer and IV&V to ensure that developer project plans stipulate timely delivery of projects artifacts to IV&V as prerequisites for project milestone events, e.g., requirements and design reviews. If project milestones occur without completion of attendant IV&V, the IV&V team will rapidly become irrelevant, and the relationship can return to the denial phase.

2.4 Developer Dependence Phase

Dependence is the fourth stage of the professional challenge and arises when the developer begins to rely on IV&V for guidance or direction during the development process. Dependence is counter-productive because by definition it destroys the independence of IV&V. Dependence can arise from any of several causes, three of which are discussed next.

- The first scenario in which we've observed dependence to occur is when the technical expertise of the IV&V team is significantly greater than that of the developer. The developer recognizes this, and asks the "expert" what to do from time to time. An extreme example of this was one project where individual programmers were contacting an IV&V engineer to approve code before it was compiled: "Would you take a look at this and tell me if I'm doing the right thing?"

  Unfortunately, dependence caused by an imbalance in technical expertise cannot be entirely controlled by the IV&V team. The probability of occurrence can be minimized, however, by stipulating and following procedures that clearly define the relationship of IV&V and the developer.

- A second cause of developer dependence is the natural desire of an engineer (the IV&V analyst) to solve problems, rather than merely identify them. For example, in one instance, an IV&V analyst suspected that an algorithm was susceptible to certain numerical singularities. Before reporting the issue, the analyst implemented and tested the algorithm, and verified that vulnerability to the singularities was indeed present. Now fascinated with the problem, the analyst developed a new algorithm that satisfied the original

requirements, but was not susceptible to the singularities. In documenting the issue, the IV&V analyst provided the new algorithm, which was subsequently incorporated in the product. Since the IV&V analyst developed the algorithm, IV&V, rather than the developer, came to be viewed as the expert on (and in a sense, the owner of) this piece of the project. Consequently, the independence of IV&V with respect to this piece of the product was eliminated.

This second cause of dependence can be avoided through a careful peer review of all IV&V issue reports. In all cases IV&V should ensure that the reported problems are clearly identified and substantiated, but that solutions are not provided. It is sometimes possible to allude to possible solutions, but never provide the solution directly.

- A third cause of dependence is hyper-criticism on the part of an IV&V analyst. In this situation, there is an analyst who can't be pleased. This analyst has the mistaken belief that the quality of IV&V can be measured by counting issue reports. Such a person will write the same number of issue reports for a development item that is nearly perfect as for a development item that has numerous serious flaws. This sort of behavior can rapidly demoralize the developer to the point that all decisions are made by the IV&V analyst.

  It is often tempting to overlook the problem of a hyper-critical IV&V analyst. Because the responsibility of identifying problems and reporting attendant issues is a primary responsibility of IV&V, it is easy to misconstrue issue production as IV&V productivity. Moreover, if IV&V management has had to constantly justify IV&V, a natural tendency exists to promote the analyst to "hero" status, and thereby, further exacerbate an already complicated problem. Nonetheless, prevention of hyper-criticism is a key responsibility of IV&V management. The responsibility of IV&V is not to produce issue reports, but to ensure that the developer is building the right product right. Note that "right" does not necessarily mean best. IV&V is not supposed to make sure that the developer follows the optimal path, but rather, that the product will do what it's supposed to do, and nothing else. We have found that a vigorous IV&V peer review process helps avoid this third cause of developer dependence. It is also important to ensure that management measures of IV&V productivity do not unreasonably emphasize issue report production. Reason provides some excellent insights on avoiding the hyper-criticism problem (Reason 93).

The above paragraphs have outlined common causes of developer dependence and have discussed strategies for avoiding such dependence. Even with such precautions, however, some form of dependence can still evolve. In fact, almost every IV&V project we have worked on has, to some extent, entered this phase. Hence, it is still important to be able to recognize and to have a strategy for breaking that dependence. The following paragraphs outline such strategies.

- In the case of the developer taking excessive advantage of the IV&V analyst expertise, IV&V must meet with the developer and customer to discuss the problem. The discussion should address the importance of independence and lead to a definitive solution outlining how to break the dependence without sacrificing project goals.

- Developer dependence stemming from an overly helpful IV&V is also best treated by open discussion of the problem with the customer and developer. The discussion should stress that it is the developer's role to assume ownership of all elements of the product. To help avoid the situation where the IV&V analyst provides solutions, we suggest that IV&V management consider changing analysts' assignments so IV&V becomes a group, rather than, individual effort.

- Hyper-criticism by an IV&V analyst often results in a breakdown of accountability and contributes to developer demoralization. When either IV&V or development personnel recognize that such behavior is occurring, steps should be taken immediately to end it. The problem should be openly brought to the attention of all parties concerned, i.e., IV&V management, the developer and the customer. IV&V should never try to pretend a mistake was not made -- the developer already knows, and the customer probably knows too. It may be advisable to assign another analyst to the task; it is certainly advisable to provide additional training or

counseling to the hyper-critical analyst. Finally, since this problem is a manifestation of one or more flaws in the IV&V peer review and training processes, those flaws should be identified and corrected.

3.0 Cyclic Nature of the Phases

In a long term IV&V project, we have observed that the sequence of phases in the developer's attitude happens once when the project begins, then is repeated on a small scale from time to time as new people are added to the development team. If a new task is added to the IV&V project, the new task will also go through the four phases (usually on a smaller scale as well). Understanding that these dynamics are in progress, and that they are natural, will help one deal with each phase most effectively.

4.0 Conclusions

The Professional Challenge to IV&V is to effectively deal with the adversarial relationship that arises from the tendency of developers to perceive IV&V as an opponent. The Professional Challenge is a natural part of the culture of a software project, and should be anticipated. The Professional Challenge consists of four distinct phases of the developer's attitude toward IV&V: Denial, Anger, Cooperation, and Dependence. In order to successfully deal with the Professional Challenge, the IV&V agent must (a) recognize that some aspect of the Professional Challenge is always present, and (b) understand the characteristics of each phase. The issues and recommendations presented in this paper reflect of our experiences in performing IV&V and "lessons learned." We are firmly convinced that to implement a successful IV&V program, one must necessarily confront and successfully mitigate those issues underlying the Professional Challenge. Success is defined by an IV&V project remaining primarily in the cooperation phase and producing results having a positive impact on the quality of the project.

References

AFSC AFLC Pamphlet 800-5. *Software Independent Verification and Validation*. Washington, D.C.: Andrews Air Force Base, Air Force Systems Command and Air Force Logistics Command, Department of the Air Force, 1988.

Arthur, James, et al. "Reducing the Mean Time to Remove Faults Through Early Fault Detection: An Experiment in Independent Verification and Validation." *Eighteenth Minnowbrook Workshop on Software Engineering* (1996). Blue Mountain Lake, NY: 28-46.

Cheng, Betty and Brent Auernheimer. "Applying Formal Methods and Object-Oriented Analysis to Existing Flight Software." *Proceedings of the 18th Annual NASA Software Engineering Workshop* (1993). Greenbelt, MD: 274-282.

Easterbrook, Steve, and Jack Callahan. "Formal Methods for V&V of Partial Specifications: An Experience Report." *Proceedings, Third IEEE Symposium on Requirements Engineering* (1997). Annapolis, MD. (http://research.ivv.nasa.gov/docs/techreports/1996/NASA-IVV-96-007.ps)

Easterbrook, Steve, and Jack Callahan. "SCR as an IV&V Tool." *Proceedings, Fifth International Software Cost Reduction Workshop* (1966). Ottawa, Canada. (http://research.ivv.nasa.gov/docs/techreports/1996/NASA-IVV-96-026.ps)

Gerhart, Susan, Craigen, Dan, and Ted Ralston. "Experience with Formal Methods in Critical Systems", *IEEE Software* 11.1 (1994): 21-28.

Leveson, Nancy. "Safety." *Aerospace Software Engineering, A Collection of Concepts*. Eds. C. Anderson and M. Dorfman. Washington, DC: American Institute of Aeronautics and Astronautics, Inc., 1991. 319-335.

Leveson, Nancy. "Software Safety Research." *Second NASA Formal Methods Workshop* (1992). Hampton, VA: NASA Langley Research Center. 175-198.

Neal, Ralph, et al. "A Case Study of IV&V Cost Effectiveness." *Software Engineering Process Group Conference '97* (1997). San Jose, CA. (http://research.ivv.nasa.gov/docs/techreports/1997/NASA-IVV-97-007.ps)

Reason, James. "The Identification of Latent Organizational Failures in Complex Systems." *Verification and Validation of Complex Systems: Human Factors Issues.* Eds. Wise, J., Hopkin, V., and Paul Stager. Berlin, Germany : Springer-Verlag, 1993. 223-237.

Rogers, Richard. "Planning for Independent Verification and Validation." *Third Computers in Aerospace Conference* (1981). San Diego, CA: 15-23.

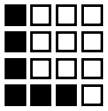
Rushby, John. *Formal Methods and Digital Systems Validation for Airborne Systems*. NASA Contractor Report 4551 (1993). Hampton, VA.

Rushby, John. *Formal Methods and Their Role in Digital Systems Validation for Airborne Systems*. NASA Contractor Report 4673 (1995). Hampton, VA.

Wallace, Dolores and Roger Fujii. *Software Verification and Validation: Its Role in Computer Assurance and Its Relationship with Software Project Management Standards* (1989). Gaithersburg, MD: National Institute of Standards and Technology.

Wallace, Dolores and Roger Fujii. "Software Verification and Validation: An Overview." *IEEE Software* 6.3 (1989): 10-17.

Westrum, Robert, “Cultures with Requisite Imagination”, *Verification and Validation of Complex Systems: Human Factors Issues.* Eds. Wise, J., Hopkin, V., and Paul Stager. Berlin, Germany : Springer-Verlag, 1993. 401-416.



Selecting and Defining Metrics by Example: Two Successful Training Paradigms

October 1998

Claire Lohr
Lohr Systems
P.O. Box 2998
Reston, VA 20195
(703) 391-9007
email lohrsys@erols.com

Topics

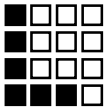
- Background
- The problem
- Examples of training solutions
  - Westfall's 12 step cookbook
  - Analysis by vignette
- Student feedback
- Concluding remarks
- References

Background

- The field of metrics has matured
- There are robust definitions, models, and methods
- There are published results of process improvements fueled by successful metrics programs
- Metrics are NOT universally implemented

The Problem

- Successful examples are plentiful, but practitioners don’t know how to implement in their own paradigm
- The problem is COMMUNICATION
  - Practitioners need help with assimilation of concepts
  - Practitioners need help translating methods into their own paradigm



Solution #1: Westfall's 12 Step Cookbook

- Reduce metrics selection and design to a comprehensive, easy to follow "cookbook"
- The students do each step as they learn it (with themselves as the customer for the metric)
- The instructor does an example first

Solution #1: Westfall's 12 Step Cookbook

Step #1: Identify a customer for the metric

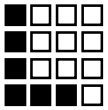
Step #2: Choose a goal (personal or professional)

Step #3: Ask one or more questions which will let you know if you have reached your goal

Step #4: Choose metric(s) which will answer the question(s)

Step #5: Write the metrics objective statement

"To (understand/evaluate/control/predict) the (attribute of the entity) in order to (goal)".



Solution #1: Westfall's 12 Step Cookbook

Step #6: Write a clear definition of the metric

Step #7: Define the model

Step #8: Establish counting criteria

Step #9: Decide what's good

Step #10: Choose how to report

Step #11: Add additional qualifiers

Step #12: Beat it

Student Feedback

(courses containing the 12 steps)

ASQ Course	# of Students	Metrics Section	All Sections
BSQ	79	1.5 / 5.0	1.5-2.25
SQE	173	1.65/5.0	1.65-2.3

Note: 1 is the highest possible score and 5 is the lowest

Solution #2:
Analysis of Evolving Vignettes

- A combination of real and fabricated data
- Only partial information is in the beginning
- More information is added in successive increments
- The data unfolds in a manner similar to evolving information on real projects
    - No history for the first time
    - Lots of history for the Nth time
    - Better results with additional data

Solution #2: Analysis of Evolving Vignettes

Vignette Example #1: Effort estimation

The task: estimate total effort for a new project

1st: Just given a Work Break Down structure (WBS) with task descriptions for the new project

2nd: Add a real set of actual effort #s for one actual project

3rd: Add more real effort #s for multiple real projects

4th: Add descriptions of the differences between the all of the projects

The results converge!!!

Solution #2: Analysis of Evolving Vignettes

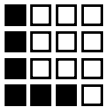
Vignette Example #2: Project Tracking

The task: evaluate status (to identify problem areas and recommend improvements)

1st: Just given six months of planned vs. actual peer reviews of requirements

2nd: Add the overall project plan

3rd: Add similar reports for previous projects



Solution #2: Analysis of Evolving Vignettes

Vignette Example #3: Peer Reviews

The task: evaluate status (to identify problem areas and recommend improvements)

1st: Just given graphs of overall product quality which show that the introduction of Peer Reviews has not increased the product quality

2nd: Add peer review examination rate, preparation rates, size of work product, and hours per inspection meeting

3rd: Add defects found by test, percentage of code inspected, and number of defects found in new code

Student Feedback

- Use of real corporate data is of great interest
- Students are encouraged - it WILL get better after the beginning
- Seeing the results of other students is very valuable

Concluding Remarks

- The gap between metrics theory and practice can be closed
- “Hands-on” training is effective
- Simplification into a “cookbook” is helpful
- Analysis of unfolding examples is helpful

References

[1] Basili, V. R., Weiss, D. M., *A Methodology for Collecting Valid Software Engineering Data*, IEEE Transactions in Software Engineering, November, 1984

[2] Westfall, Linda, *Software Metrics to Meet Your Information Needs*, ASQ 49th Annual Quality Congress Transactions, 1995

[3] Lohr, Claire; Westfall, Linda; Smith, Michael; *Building Software Quality Skill*s, ASQ, 1996

[4] Lohr, Claire; Westfall, Linda; Smith Michael; *Software Quality Engineering*, ASQ, 1995

LOCKHEED MARTIN

Experiences Implementing Software Measurement

Beth Layman
Senior Consultant
beth.a.layman@lmco.com

Lockheed Martin Mission Systems
Consultative Engineering
7700 Technology Drive
Melbourne, FL 32904
(407) 984-2561 voice • (407) 951-2601 fax

Sharon Rohde
Senior Consultant
sharon.l.rohde@lmco.com

Abstract

This paper discusses practical experiences after two years of implementing a proven, software measurement process using quantitative data to manage software projects. As a principal author of the process known as ***Practical Software Measurement: A Foundation for Objective Project Management (PSM)***, Beth Layman has helped commercial businesses, government agencies, and Lockheed Martin organizations implement the *PSM* guidance. Tips for getting started, project measurement roadblocks to avoid, and advice for institutionalizing project measurement are discussed.

Primary Author's Biographical Sketch

Beth A. Layman is a Senior Consultant in Lockheed Martin's Consultative Engineering group (part of Lockheed Martin Mission Systems), and has over 19 years of experience in the computer industry. She has a software development background with specialization in quality assurance, measurement, and process improvement methods, and has project and SQA management experience in both information systems and product software environments. At Lockheed Martin, she is responsible for providing software measurement, process improvement, and quality consulting to customers both outside and within the organization.

Prior to joining Lockheed Martin, Beth operated her own consulting business and served as Research Director and Senior Consultant for the Quality Assurance Institute. She has lead research efforts in areas such as requirements definition, software testing, and function point analysis. She has also developed and taught training courses, performed organizational assessments, chaired conferences, and produced quality-related products and tools.

Beth has a degree in business from Capital University, is a Certified Quality Analyst (CQA), and served as a senior examiner for Florida's Sterling Quality Award program. She is a principal contributor to *Practical Software Measurement: A Foundation for Objective Project Management (PSM).* Beth is an associate editor of *Software Quality Professional*, ASQ Software Division's new journal, slated for publication in October 1998.

Introduction

A software project manager's worst nightmare is having his or her project canceled. Unfortunately, studies have shown that as many as 1 in 10 software projects are canceled, often due to excessive cost or schedule overruns, unmanageable scope creep, or unmet technical objectives. Many software organizations measure schedule delays in years, not months or weeks! We, as an industry, are aware of the many contributors to this "software project crisis" – unrealistic estimates, poor planning, poor risk management, lack of information to support decision making, and so on. What can we do about it?

There is a growing awareness in the software industry that measurement plays an important role in solving these problems. Measurement, when integrated into the overall project management process, provides the information necessary to identify and manage the issues inherent in software projects. Measurement at the project-level can be used to objectively validate estimates and plans, track progress, and even anticipate potential problems such as schedule slippage and cost overruns. The goal of project-level measurement is to provide project managers with sufficient insight into the project to support decision-making and positively influence project outcomes.

Lockheed Martin was the prime contractor for the **Practical Software Measurement** program, sponsored by the U.S. Department of Defense. ***Practical Software Measurement: A Foundation for Objective Project Management (PSM)*** is one of the program's primary products; it is a guidebook which presents a systematic measurement approach and explains techniques for using measurement to make project decisions in time to affect the outcome of a software project. *PSM* is unique in that it was developed by a working group of measurement experts from both government and industry, and has received widespread endorsements throughout the international software measurement community.

Our Consultative Engineering group has been working to transition the *PSM* guidance to software-intensive commercial, government, and Lockheed Martin projects. This paper first provides an overview of the *PSM* guidance, then describes the lessons we have learned through our experiences implementing *PSM* during the last two years. These lessons should be useful to anyone implementing *PSM* or any other project measurement approach.

PSM Overview

PSM is a guidebook designed to help the software project manager: 1) identify those issues and objectives that are important to their project's success; 2) implement a measurement program focused on those issues; and 3) gain objective insight into those issues throughout the project's life cycle. *PSM* represents a practical, easy-to-use set of "best practices" for software measurement. Because *PSM* presents a flexible measurement process, versus a fixed set of software measures, it can be applied to virtually any software project. Information on how to obtain a complete copy of the *PSM* guidance is provided at the end of this paper.

PSM characterizes the key elements or *principles* of a successful measurement program, then describes a comprehensive measurement *process* based on those principles. The process consists of three major activities, as shown in Figure 1. The first activity describes how to **tailor** the measurement program to address project-specific issues, risks, and objectives. The second activity describes a systematic process for **applying,** or using, measurement to gain insight into the project's issues, and to aid in decision-making. The third activity, **implementing** measurement, explains how to get measurement into practice within an organization. In addition to the process guidance, *PSM* includes detailed selection and specification guidelines for proven software measures, sample indicators, measurement case studies from real-life software projects, and guidance for putting measurement on contract.

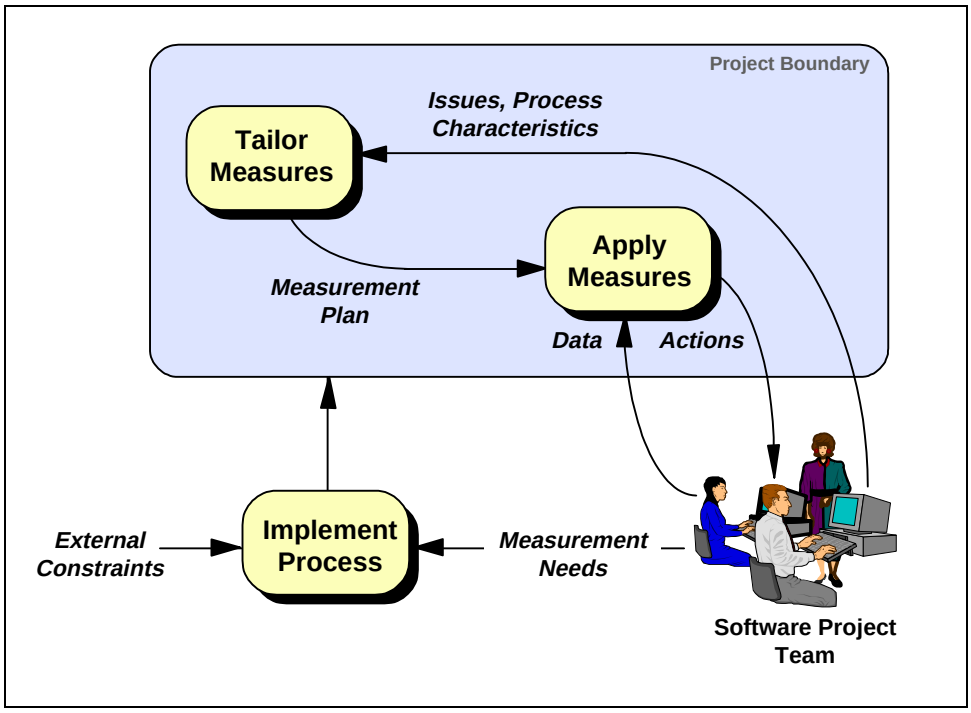


Figure 1. Software Measurement Activities

Developing a Measurement Plan

During the tailoring phase, measurement requirements for the project are identified. *PSM's* issue-driven approach stipulates that the **project's unique, specific issues and objectives drive the identification of measurement requirements.** This is because the purpose of measurement is to first and foremost help the project achieve its objectives, identify and track risks, satisfy constraints, and recognize problems early. *PSM* defines the following common types of project issues:

- Schedule and Progress
- Resources and Cost
- Growth and Stability
- Product Quality
- Development Performance
- Technical Adequacy

PSM emphasizes identifying project issues at the start of a project and then using the measurement process to provide insight to those issues. While some issues are common to most projects, each project typically has some unique issues. Examples of program-specific issues might be lack of available Object-Oriented (OO) expertise/resources or concerns about the implementation of a particular software package. Figure 2 illustrates how program specific issues can be mapped to PSM common issues using the tailoring guidance to help identify useful measures and apply them.

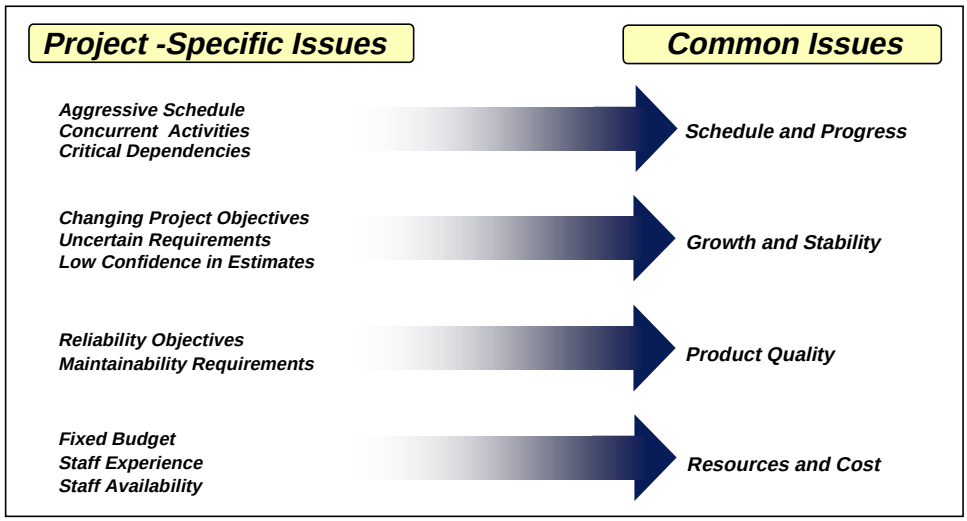


Figure 2. Example of the Mapping of Program Specific Issues to PSM Common Issues

Also, the priority of issues usually varies from project to project. Moreover, it is important to note that many of these issues are inter-related. For example, while an incremental development approach may help uncover or clarify requirements, it may also lead to more schedule slippage. Lack of available OO expertise may not only contribute to additional costs and schedule delays, but may also jeopardize software quality. The relationships between issues must be considered when prioritizing issues.

Once project-specific issues are identified and prioritized, measures can be selected that will provide insight into those issues. A measure is a quantification of an attribute of a software process or product. A variety of measures may be needed to provide insight into a single issue. Measurement selection will be driven by a number of factors including:

- The cost to collect the measure
- The availability of the measurement data
- The timeliness, accuracy, and validity of the measure
- The measures "fit" given relevant project/organizational characteristics

For example, if requirements growth and stability is an issue, then a functional size measure will be needed to track it. The appropriate measure will depend on the nature of the project. Application domain and organizational history/experience will influence the choice of a functional size measure. IT/MIS organizations may already use function points. However, contract software development organizations with a history of tight requirements management techniques may be more comfortable using a requirements counting schema. An important consideration at this stage of the tailoring process is whether the measures selected can realistically be integrated into the project's day-to-day operating procedures.

PSM provides a list of approximately 40 candidate measures. The measures identified in *PSM* are measures that have been used successfully by members of *PSM's* technical working group, which is composed of measurement practitioners from more than 40 different software producing organizations. While this list of measures is by no means complete, it represents a starting place for identifying and specifying measurement requirements. For each measure identified, helpful information is included in the guidance such as: data items and useful attributes to collect; recommended unit of measure, collection level, reporting level; applicability to various domains; sample indicators; etc.

The measurement plan documents the measurement requirements for the project, starting with the project's issues and ending with a complete specification of each measure selected. It does not have to be a lengthy document, but should capture the following:

- Issues
- Measures
  - Data elements to be collected
  - Data definitions
  - Data sources/tool
  - Data Collection Level
  - Data Collection Frequency
  - Access Mechanisms
- Aggregation strategy (how low-level data will be summarized)
- Frequency of analysis and reporting
- Reporting roles and responsibilities

Applying and Using Measurement

After a project gets underway, measurement data is regularly collected according to the measurement plan. Measurement tools, databases, and spreadsheets are often used to collect, store and process the raw measurement data. Once collected, raw *data* is turned into *information*. As data is aggregated, compared, and analyzed within the context of recent project events, information emerges which can be used to help manage the project. *PSM* advocates the use of a flexible and dynamic analysis process that promotes the use of measurement as primarily an "investigative" activity. The key building blocks of this activity are **indicators**. An indicator is a measure or combination of measures that provides insight into a project issue. Indicators often compare actual project performance data to a plan or baseline, and are often portrayed graphically.

One of the unique aspects of the PSM guidance is the amount of detail provided regarding the measurement analysis process. *PSM* 3.1 describes three types of analysis that are performed on data: Estimation (the development of targets based on historical data and project assumptions); Feasibility Analysis (the analysis of the feasibility of initial and subsequent project plans which use the estimates as a basis); and Performance Analysis (the analysis of actual performance compared to project plans). *PSM* also describes a four-step process that can be applied whenever data is analyzed. The four steps are:

1. Identify the problem
2. Assess problem impact
3. Project possible outcomes
4. Evaluate alternatives

Finally, *PSM* stresses the importance of understanding the relationships between project issues and the measurement information. *PSM* prescribes the use of an analysis model (see Figure 3) which shows how some indicators can serve as leading indicators for a particular issue, because they provide insight that contributes to the emergence of the issue. The plusses and minuses show if an increase in the contributor results in an increase (+) or decrease (-) in the resulting issue. For example, an increase in functional size (due to requirements growth) can result in an increase in product size, and an increase in product size can result in an increase in the effort required to complete the project. Therefore, requirements growth could be viewed as a leading indicator of effort overruns; this means that this relationship and the possible resulting outcomes should be considered during the analysis process.

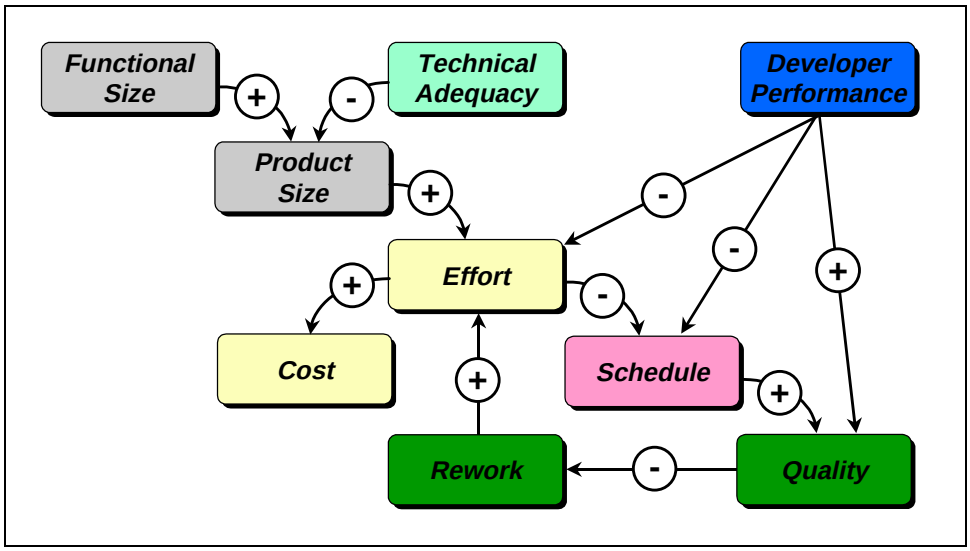


Figure 3. PSM's Analysis Model

The last step in applying measurement on a project is to actually use the insight gained from measurement analysis to make decisions. This involves communicating the results of the analysis (i.e., current problems, impacts, outcomes, and alternatives) to the decision makers and taking action. PSM provides guidance on how to clearly communicate results and how to track the results of actions taken.

Because new issues and problems can emerge at any time throughout a software project, *PSM* advocates that the measurement process implemented be flexible and responsive to change as the project evolves. This means revisiting the tailoring activities and modifying measurement plans as needed throughout the project life cycle. The issues, measures, and analysis techniques must be changed over time to best meet the project's information needs.

Lessons Learned Implementing *PSM*

We have provided training, conducted measurement planning workshops, and assisted with full-scale implementation of *PSM* on a number of projects – large and small – in both development and maintenance groups. The PSM process has received a very favorable response from the project teams we have worked with because the focus is on meeting their project's information needs versus meeting some requirement to provide "outsiders" with project data.

However, we have also encountered a number of difficulties also; these potential problems must be understood and resolved before the process can be successfully deployed. Our experiences described in this section represent common or reoccurring implementation issues in the organizations we have consulted with.

Lesson #1: Mind the Gap.

(Translation: There is often a disconnect between the measures currently collected and the issues "real projects" face.)

One way we help organizations implement *PSM* is through 1-2 day facilitated workshops. Using *PSM's* issue-driven approach, we typically lead project teams through an issue identification process early in the workshop. Here, we use brainstorming and project team synergy to identify existing project issues and constraints, and potential issues/risks that may affect future phases. Next, we identify the type of information that would provide insight into the highest priority issues. Only then do we look at data currently collected and measures/metrics

requirements, if any. A disconnect between what's currently being measured and what information is needed to help the project address their issues often becomes apparent at this stage.

Many of the organizations we consult with already have "measurement initiatives" underway. Typically, a measurement group has been established, and that group has developed a required list of metrics that all projects are required to produce on a regularly scheduled basis. Project staff often tell us that the required metrics are of no value to them. Also, because their project's process doesn't support, as a natural by-product, the collection of the data or the analysis (which should be an integral part of the project management and decision-making process), they often just "meet" the requirements without concern for data validity or accuracy (i.e., fudged numbers).

Often, *PSM's* flexible-at-the-project-level, issue-driven approach is totally counter to existing measurement practices. This usually becomes apparent during the workshop – just as projects are starting to see that measurement might be of value to them to manage their projects and make real-time decisions, the measurement group begins raising objections about losing control of the measurement process. Also, a real concern is losing the ability to capture common measures and compare status and performance across projects. While this seems like a big stumbling block, in reality it usually is not. The concerns are usually overcome with one or more of the following:

- **Recognize that organizational reporting requirements are simply a subset of the project measurement process and learn to make better use of what is required on the project**. If properly implemented, the measurement data collected can often be used in a variety of ways to gain insight into a variety of different issues. We try to get everyone to realize that projects can provide a static set of graphs for organizational use while, at the same time, dynamically analyzing the same raw data and generating other graphs to get insight into their "real-time" issues.

- **Encourage the measurement group to look at the issues that are driving the organizational reporting requirements and streamline the requirements** based on the highest priority organizational information needs. Sometimes, closer examination reveals that the "required measures" are either not **really** being used to make organizational decisions or drive future estimates/plans, or the return on investment of collecting certain information is questionable. Sometimes, a more aggregated view of the data is all that is really needed (e.g., planned and actual work effort by phase to use in future estimates versus detailed effort reporting by person, time period, etc.). Other times, it becomes clear that different measures for different types of projects are more appropriate (i.e., Function Point sizing for new development versus sizing based on change requests for maintenance).

- **Differentiate the information needed to make organizational decisions from information needed to provide senior management oversight into key projects.** If senior management wants oversight into key projects, this can usually be accomplished with regular project briefings where project-specific measures are presented. In fact, this approach is superior to the same-status-report-for-every-project approach, because it makes visible the things that are really impacting the project and forces management to help the project staff remove any real-time roadblocks. It does mean that senior management may be slightly more taxed because they may now have to view different graphic indicators for each project.

- **Resolve to "walk first"**. Recognize that, without effective use of measurement at the project team level, the measurement program within an organization will be weak. This is because most data used within a software-intensive organization comes from the performance of project work. It's easier to get buy-in for collecting a common set of measures across projects after individuals buy-in to the value of measurement.

The disconnect between organizational and project needs can be seen in the following example of the often-required schedule data. While monthly Gantt or schedule variance charts are often required for each project, they provide little insight into the cause of schedule slips and often don't indicate a problem until the problem is of major proportion. Once projects identify the nature of the schedule issue, simple work unit progress charts, like the one shown in Figure 4, can be used to augment or even replace the Gantt chart. Project leads can use these progress charts to pinpoint schedule problems long before they appear on a Gantt chart.

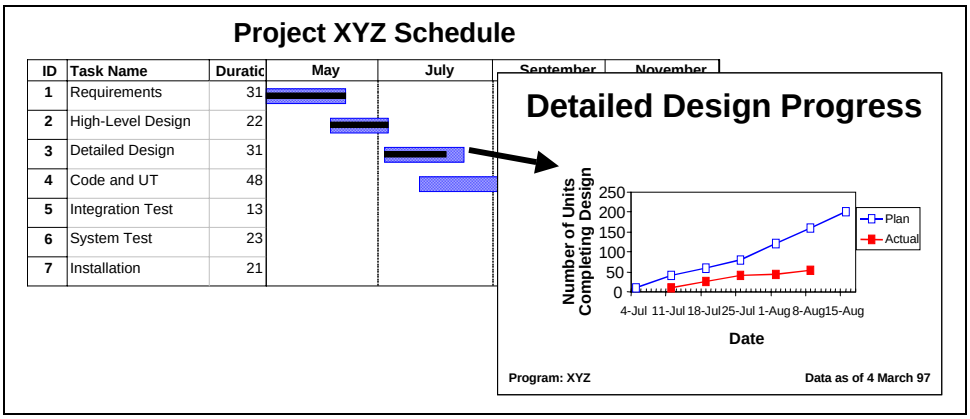


Figure 4. Organizational Reporting Requirements vs. Project Information Needs

Lesson #2: Get "Hooked" on Metrics.

(Translation: Making people need measurement is the best first step towards institutionalizing it.)

"Measurement? Not my job! We have a measurement group"
"Measurement? Who has the time?"
"Measurement? Don't need it – I know where my project stands"

Our philosophy is, rather than fighting with logic the many and varied objections project teams may raise against project measurement, we get them "hooked" on measurement instead. Measurement planning workshops are a good way to do this. We isolate the project team for 1-2 days, immerse them in *PSM*, change their perception of measurement, help them realize their need for information, and make them feel their "data depravity". We show them how the information they need could be derived from their existing process, and show them how the information could be used during a project. This often transforms resistive types into chomping-at-the-bit advocates. A key to this transformation process is getting project teams to take ownership of their issues, and recognizing their responsibility for making visible the things that are happening on the project – things both within and outside their control.

Once people feel they "need" specific data or indicators, they find ways to use them. After they begin using them, they often find ways to build on what they have in order to meet other needs, and build more measurement into their process. To get this cycle going and achieve real institutionalization (e.g., where measurement is a natural by-product of the process and represents the "way we do things here"), startup assistance and ongoing consulting services are usually needed. This is where the measurement group can be of service. Many measurement groups we have encountered have traditionally "owned" the analysis of measurement data. They collect the data from the project and produce various charts and graphs. They interpret the data and often report results within the organization. With *PSM*, measurements are analyzed and used by the project team. We work with these measurement groups to help them transition from measurement "doers" into measurement "consultants." They offer their expertise to projects and help them build simple spreadsheets and collection systems to collect the data they need. They help project members learn how to generate graphs from spreadsheets. They share these simple tools across projects. And finally, they advise projects in the proper use of measurement and in this way, help establish true quantitative software project management within their organizations.

Many of our clients with maintenance or sustaining projects have the reoccurring information need to make visible the impact of a "trickling resource drain." Typically the project staff is constantly being tagged for non-project work, yet this drain is never assumed in project estimates nor is the impact of the drain quantified or even given visibility at the program management/customer level. A fixed staffing level is usually assumed. Project staff identify the resource drain as a major issue, construct simple charts like the one in Figure 5, showing a gap between planned and actual staffing, and present it along with schedule data. Management and customers often say, "Wow – I had no idea!" and are willing to take corrective action to curb non-project activities. A very simple measurement often helps solve a very big problem, and gets project teams hooked on metrics.

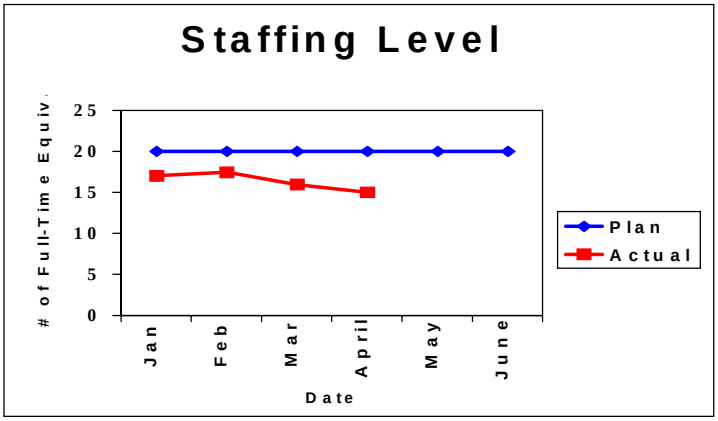


Figure 5. Meeting Simple Project-Specific Needs for Resource Drain Visibility

Lesson #3: What THEY don't know, won't hurt US.

(Translation: The "project culture" will impact the implementation.)

One of the biggest roadblocks to implementing effective project measurement is the prevailing project culture, which is difficult and slow to change. Unfortunately, some organizations still suffer from one or both of the following:

- **Don't make bad news visible or else you'll get blamed.** One workshop attendee lamented: "If this type of data becomes available, management will know where the project actually stands!" Of course, that's precisely what *PSM* is advocating – identify a problem as early as possible in order to fix the problem before it becomes catastrophic or unsolvable. Management that reacts negatively rather than constructively to less-than-stellar performance creates a culture that finds itself constantly in "crisis" mode. We've seen strong quality advocates and management consultants within developer organizations successfully alter this climate by educating and coaching management on the need to personally change from "blamers" to "helpers", but it's not a fast or easy transition.

- **Don't give the customer insight into the development process – they don't understand these things.** *PSM* advocates sharing measurement information with the customer. This is particularly important when the customer is internal (MIS/IT) and/or when a single customer is paying the bill (MIS/IT or contractual situations). In these cases, the customer plays an important role in the success of the project. Their inputs drive the development process and sometimes their decisions directly affect the project outcome. Because of this, they need to understand how the project is performing and why.

The bottom line is that both management and customers must be willing to listen and respond constructively to bad news. If this type of maturity is not present in your current project culture and no attempt is being made to change it, measurement will only be useful within the project's limited scope of control.

Other Suggestions for Getting Started

Based on our experiences with implementing *PSM* to date, we put forth the following additional suggestions for getting started with project measurement:

- **Market the Approach** – Tie quantitative project management to current software process improvement efforts and show how measurement is integral to risk management, meeting project commitments, and improving process maturity. For example, the measurement approach described is *PSM* consistent with, and can be easily linked to, ISO and CMM-based improvement initiatives (i.e., CMM's repeatable level requires planning and tracking of project costs, schedule, effort, size, and quality). This type of marketing must occur continuously.

- **Provide Education** – Ensure that all levels of the organization understand the benefits of measurement and are taught the basic measurement principles, steps, and techniques, such as those outlined in *PSM*. Management, in particular, needs to understand their special role in supporting the process, and their responsibility to actively participate in analyzing/interpreting measurement results and taking corrective actions when needed. We have found that some managers don't understand how to properly analyze and interpret charts and graphs, so be sure to reinforce the use of a systematic analysis process like the one we've discussed in this paper.

- **Conduct Project Planning Workshops** – Consider measurement planning workshops as an alternative to traditional training courses to introduce people to project-level measurement concepts. This enables projects to develop their own measurement plan while learning the process.

- **Focus on a Few Measures** – For projects just starting to perform measurement, ensure that a feasible measurement plan is developed. This may translate into a very small subset of measures. Stress the notion that what you measure can change throughout the process as the project's issues evolve and as the project gains more experience with measurement.

References

[1] Carnegie Mellon University, Software Engineering Institute, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, Reading, MA, page 23.

Obtaining a Copy of the PSM Guidance

At the time of this writing, version 3.1 of the PSM Guidebook (April 13, 1998) was available on the *PSM* Web Site at http://www.psmsc.com.

Acknowledgements

We would like to acknowledge the other authors of *PSM*: John McGarry, Cheryl Jones, David Card, Betsy Bailey, and Joseph Dean; the PSM Support Center; and the many interesting (but un-named!) projects and project team members whose experiences have served as input for the writing of this technical paper.

Requirements, Testing, and Metrics

Linda H. Rosenberg, PhD
Unisys/GSFC
301-286-0087
Linda.H.Rosenberg@pop300.gsfc.nasa.gov

Theodore F. Hammer
NASA GSFC
301-286-7475
thammer@pop300.gsfc.nasa.gov

Lenore L. Huffman
Unisys/GSFC
301-286-0099
Lhuffman@pop300.gsfc.nasa.gov

Key Words: Testing, Requirements, Metrics

ABSTRACT

The criticality of correct, complete, testable requirements is a fundamental tenet of software engineering. Also critical is complete requirements based testing of the final product. Modern tools for managing requirements allow new metrics to be used in support of both of these critical processes. Using these tools, potential problems with the quality of the requirements and the test plan can be identified early in the life cycle. Some of these quality factors include: ambiguous or incomplete requirements, poorly designed requirements databases, excessive or insufficient test cases, and incomplete linkage of tests to requirements. This paper discusses how metrics can be used to evaluate the quality of the requirements and test to avoid problems later.

Requirements management and requirements based testing have always been critical in the implementation of high quality software systems. Recently, automated tools have become available to support requirements management. At NASA's Goddard Space Flight Center (GSFC), automated requirements management tools are being used on several large projects. The use of these tools opens the door to innovative uses of metrics in characterizing test plan quality and assessing overall testing risks. In support of these projects, the Software Assurance Technology Center (SATC) is working to develop and apply a metrics program that utilizes the information now available through the application of requirements management tools. Metrics based on this information provides real-time insight into the testing of requirements and these metrics assist the Project Quality Office in its testing oversight role. This paper discusses three facets of the SATC's efforts to evaluate the quality of the requirements and test plan early in the life cycle, thus preventing costly errors and time delays later. Data from NASA projects are used to support and clarify the concepts discussed.

Linda H. Rosenberg, Ph.D.

Dr. Rosenberg is an Engineering Section Head at Unisys Government Systems in Lanham, MD. She is contracted to manage the Software Assurance Technology Center (SATC) through the System Reliability and Safety Office in the Flight Assurance Division at Goddard Space Flight Center, NASA, in Greenbelt, MD. The SATC has four primary responsibilities: Metrics, Standards and Guidance, Assurance tools and techniques, and Outreach programs. Although she oversees all work areas of the SATC, Dr. Rosenberg's area of expertise is metrics. She is responsible for overseeing metric programs to establish a basis for numerical guidelines and standards for software developed at NASA, and to work with project managers to use metrics in the evaluation of the quality of their software. Dr. Rosenberg's work in software metrics outside of NASA includes work with the Joint Logistics Command's efforts to establish a core set of process, product and system metrics with guidelines published in the *Practical Software Measurement*. In addition, Dr. Rosenberg worked with the Software Engineering Institute to develop a risk management course. She is now responsible for risk management training at all NASA centers, and the initiation of software risk management at NASA Goddard. As part of the SATC outreach program, Dr. Rosenberg has presented metrics/quality assurance papers and tutorials at GSFC, and IEEE and ACM local and international conferences. She also reviews for ACM, IEEE and military conferences and journals.

Immediately prior to this assignment, Dr. Rosenberg was an Assistant Professor in the Mathematics/Computer Science Department at Goucher College in Towson, MD. Her responsibilities included the development of upper level computer science courses in accordance with the recommendations of the ACM/IEEE-CS Joint Curriculum Task Force, and the advisor for computer science majors.

Dr. Rosenberg's work has encompassed many areas of Software Engineering. In addition to metrics, she has worked in the areas of hypertext, specification languages, and user interfaces. Dr. Rosenberg holds a Ph.D. in Computer Science from the University of Maryland, an M.E.S. in Computer Science from Loyola College, and a B.S. in Mathematics from Towson State University. She is a member of Electrical and Electronic Engineers (IEEE), the IEEE Computer Society, the Association for Computing Machinery (ACM) and Upsilon Pi Epsilon.

Dr. Linda Rosenberg
GSFC
Code 300.1, Bld 6
Greenbelt, MD 20771
301-286-0087 (voice)
linda.rosenberg@gsfc.nasa.gov

Theodore F. Hammer

Mr. Ted Hammer is the NASA manager for the Software Assurance Technology Center (SATC) at NASA's Goddard Space Flight Center (GSFC). The SATC, through associations with NASA and GSFC projects and organizations, seeks to improve GSFC and NASA software by improving software quality, reducing development risks, and lowering life cycle costs. A prime focus of the SATC is the provision of software metrics support to GSFC and NASA software development and acquisition projects. In order to meet the needs of these projects, the supporting research done by the SATC is essential to allow the assurance activities to keep pace with the changing software development environment. In addition, the SATC develops techniques, provides software assurance tools, and transfers this technology to NASA and industry.

Prior to this position, Mr. Hammer was a member of the Assurance Management Office where he is responsible for managing the overall quality assurance activities for specific ground system implementation projects, with special emphasis on software quality assurance. Mr. Hammer is also responsible for managing software quality assurance activities for selected spacecraft implementation projects.

Mr. Hammer has over 22 years experience in software development and assurance, 9 with the government at GSFC, and 14 with the government and private industry supporting the Naval Sea Systems Command (NAVSEA). Early in his career he was responsible for test software development for the Combat Direction System on destroyer and frigate classes of ships. He then became responsible for the hardware and software upgrades for the Combat Direction System on these same classes. He moved to private industry, Vitro and ISA, supporting NAVSEA by reviewing software development specifications and witnessing software testing. He later returned to government service (NAVSEA) as project engineer responsible for the implementation, installation and upgrade of the ASW Control System hardware and software on DD963 and AEGIS Class ships. He then worked with the Combat Systems Office and was responsible for planning and coordinating the land based test and evaluation of combat system software upgrades to carriers, cruisers, and destroyers.

He joined NASA/GSFC in 1989. Here he supported NASA Headquarters Software Management Assurance Program, where he participated in the review of the early versions of the military software development standard, MIL-STD-498, as well as NASA software development and assurance standards and guidebooks.

Mr. Hammer received a B.S. in Electrical Engineering from the University of Maryland. He is a member of the American Society for Quality.

Theodore F. Hammer
GSFC Code 302
Greenbelt, MD 20771
(301) 286-7475 (voice)
(301) 286-1701 (fax)
thammer@pop300.gsfc.nasa.gov

Lenore L. Huffman

Lenore L. Huffman is a principal engineer with the Software Assurance Technology Center (SATC). Ms. Huffman has more than 14 years of software engineering and quality assurance experience. She is expert in the design, implementation, and execution of data collection, database structures, and metrics reporting and analysis. She is also expert in the design and use of State-Of-The-Art database reporting systems. Ms. Huffman has extensive experience automating Configuration Management and Problem Reporting Systems and adapting their capabilities to satisfy unique project requirements. She has successfully planned, designed, and implemented software quality assurance projects. Prior to joining the SATC, Ms Huffman developed metrics for software at the Space Telescope Institute, and while working at a chemical research center, was awarded with several U.S. patents. Ms Huffman holds a M.B.A. and a B.S.

Lenore L. Huffman
GSFC
Code 300.1, Bld 6
Greenbelt, MD 20771
301-286-0099 (voice)
Lenore.L.Huffman.1@gsfc.nasa.gov

Requirements, Testing, and Metrics

Linda H. Rosenberg, PhD
Unisys/GSFC
301-286-0087
Linda.H.Rosenberg@pop300.gsfc.nasa.gov

Theodore F. Hammer
NASA GSFC
301-286-7475
thammer@pop300.gsfc.nasa.gov

Lenore L. Huffman
Unisys/GSFC
301-286-0099
Lhuffman@pop300.gsfc.nasa.gov

Key Words: Testing, Requirements, Metrics

ABSTRACT

The criticality of correct, complete, testable requirements is a fundamental tenet of software engineering. Also critical is complete requirements based testing of the final product. Modern tools for managing requirements allow new metrics to be used in support of both of these critical processes. Using these tools, potential problems with the quality of the requirements and the test plan can be identified early in the life cycle. Some of these quality factors include: ambiguous or incomplete requirements, poorly designed requirements databases, excessive or insufficient test cases, and incomplete linkage of tests to requirements. This paper discusses how metrics can be used to evaluate the quality of the requirements and test to avoid problems later.

Requirements management and requirements based testing have always been critical in the implementation of high quality software systems. Recently, automated tools have become available to support requirements management. At NASA's Goddard Space Flight Center (GSFC), automated requirements management tools are being used on several large projects. The use of these tools opens the door to innovative uses of metrics in characterizing test plan quality and assessing overall testing risks. In support of these projects, the Software Assurance Technology Center (SATC) is working to develop and apply a metrics program that utilizes the information now available through the application of requirements management tools. Metrics based on this information provides real-time insight into the testing of requirements and these metrics assist the Project Quality Office in its testing oversight role. This paper discusses three facets of the SATC's efforts to evaluate the quality of the requirements and test plan early in the life cycle, thus preventing costly errors and time delays later. Data from NASA projects are used to support and clarify the concepts discussed.

1. Introduction

The National Aeronautics and Space Agency (NASA) is increasingly reliant on software for the functionality of the systems it develops and uses. The Agency has recognized the importance of improving the way it develops software, and has adopted a software strategic plan to guide the improvement process. At NASA's Goddard Space Flight Center (GSFC), the Software Assurance Technology Center (SATC) and the project Quality Assurance Office are working together to develop and apply a metrics program that utilizes the information available in the requirements phase of the software development life cycle. Metrics based on this information provides insight into the testing of requirements; this information assists the Quality Assurance Office in its project oversight role.

Requirements development and management have always been critical in the implementation of software systems—engineers are unable to build what analysts can't define. Recently, automated tools have become available to support requirements management. The use of these tools not only provides support in the definition and tracing of requirements, but it also opens the door to effective use of metrics in characterizing and assessing testing. Metrics are important because of the benefits associated with early detection and correction of problems with requirements. Problems that are not found until testing are at least 14 times more costly to fix than if the problem was found in the requirement phase.[2]

The first group of testing metrics activities that will be discussed involve the development of a tool and its application early in the life cycle in order to assess the quality of requirements. This paper describes application of

the Automated Requirements Measurement (ARM) tool. ARM parses the text of the requirements into identifiable units in order to evaluate potential words or phrases that may affect their testability. Because both the software acquirer and the software provider must understand and contractually agree to the requirements, specifications are usually written in natural language. The use of natural language to prescribe complex, dynamic systems has at least two severe problems: ambiguity and inaccuracy. Many words and phrases have dual meanings which can be altered by the context in which they are used. 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.

Once requirements are written, methods for ensuring that the system contains the functionality specified must be developed. The next group of testing metrics activities we investigate relate to the test plan links between test cases and the requirements. Using NASA project data we will look at the linkage of the requirements, the relationship between unique requirements and unique tests, and the ratios of test to requirement links. In this section of the paper we discuss three efforts to evaluate the quality of the test plan while still in the requirements phase.

And finally, this paper investigates the quality of the requirements management database schema as it relates to cleanliness of data and the ease with which requirement and testing metrics can be obtained. In the preparation of the database that houses the requirements and tests, both the requirement segment and the test segment must be designed with the identical schema design philosophies; this to enable evaluation of the test plan links to requirements as they are entered into the database. This paper briefly discusses a requirements database schema that supports comprehensive evaluation of requirements driven testing.

There are no published or industry guidelines or standards for these testing metrics—intuitive interpretations, based on experience and supported by project feedback, are used in this paper. NASA project management has reacted favorably to these metrics and has used the analysis results to mitigate perceived risks. The SATC continues working on methods to mathematically validate the intuitive guidelines. The objective is to assist project management in producing high-quality requirements and test plans while identifying and minimizing project risks.

2. NASA Development Environment

In order to demonstrate how metrics can provide the insight needed to get the requirements right, data from two large NASA projects will be used. While the projects must remain anonymous, a general understanding of the development environment is necessary. These projects are implementing large systems in multiple incremental builds.[1] The development of these builds is overlapping, design and coding of the second and third builds having been started prior to the completion of the first build. Each build adds new functionality to the previous build and satisfies a further set of requirements.

The definition of requirements for this system started with the formulation of System Level Requirements, referred to as “Level 1” requirements. These are mission-level requirements for the spacecraft and ground system; they are at a very high level and rarely, if ever, change. Level 1 requirements then undergo decomposition to produce Allocated Requirements, called “Level 2”; these requirements are also high-level and change should be minimal. Project development starts at this requirement level. (We will not discuss Level 1 or Level 2 requirements.) Level 2 requirements are then divided into subsystems and a further level is derived in greater detail; hence, “Level 3 Derived Requirements.” Each requirement in Level 2 traces to one or more requirements in Level 3. This is a bi-directional tracing, with Level 3 requirements refocusing into Level 2 requirements. The Detailed Requirements are found in “Level 4” requirements; these requirements are used to design and code the system. There is also a bi-directional tracing between Level 3 requirements and Level 4 requirements. To verify the requirement, two stages of testing are used. System Tests are designed to verify the Level 4 requirements and then Acceptance Tests are to be used to verify the Level 3 requirements.[8]

[1] Various names are used—deliveries, releases, builds—but the term *build* will be used in this paper.

3. Requirement Specification

The importance of correctly documenting requirements has caused the software industry to produce a significant number of aids to the creation and management of the requirements specification documents and individual specifications statements. However very few of these aids assist in evaluating the quality of the requirements document or the individual specification statements themselves. The SATC has developed a tool to parse requirements documents. The Automated Requirements Measurement (ARM) software was developed for scanning a file that contains the text of the requirements specification. During this scan process, it searches each line of text for specific words and phrases. These search arguments (specific words and phrases) are indicated by the SATC's studies to be an indicator of the document's quality as a specification of requirements. ARM has been applied to 56 NASA requirement documents. Seven measures were developed, as shown below.

1. Lines of Text - Physical lines of text as a measure of size.
2. Imperatives - Words and phases that command that something must be done or provided. The number of imperatives is used as a base requirements count.
3. Continuances -Phrases that follow an imperative and introduce the specification of requirements at a lower level, for a supplemental requirement count.
4. Directives – References provided to figures, tables, or notes.
5. Weak Phrases - Clauses that are apt to cause uncertainty and leave room for multiple interpretations measure of ambiguity.
6. Incomplete – Statements within the document that have TBD (To be Determined) or TBS (To Be Supplied).
7. Options - Words that seem to give the developer latitude in satisfying the specifications but can be ambiguous.

It must be emphasized that the tool does not attempt to assess the correctness of the requirements specified. It assesses individual specification statements and the vocabulary used to state the requirements; it also has the capability to assess the structure of the requirements document. [10]

The results of the analysis of the Level 3 and Level 4 requirements are shown in Table 1 with the comparison to other NASA documents.

56 DOCUMENT	**LINES OF TEXT** - Count of the physical lines of text	**Imperatives** - shall, must, will, should, is required to, are applicable, responsible for	**Continuances** - as follows, following, listed, inparticular, support	**Directives** - figure, table, for example, note:	**Weak Phrases** - adequate, as applicable, as appropriate, as a ninimum, be able to, be capable, easy, effective, not limited to, if practical	**Incomplete** - TBD, TBS, TBR	**Options** - can, may, optionally
Minimum	143	25	15	0	0	0	0
Median	2265	382	183	21	37	7	27
Average	4772	682	423	49	70	25	63
Maximum	28459	3896	118	224	4	32	130
Stdev	759	156	99	12	21	20	39
Level 3 FOS	1011	588	577	10	242	1	5
Level 4 FOS	1432	917	289	9	393	2	2

Table 1 : Textual Requirement Analysis

We are especially concerned with the number of weak phrases since the contract is bid using Level 3 and acceptance testing will be against these requirements. It is also of concern that the number of weak phrases has increased in Level 4, the requirements used to write the design and code and used in Integration testing.

An easy way to compare Project X with other NASA documents is to normalize by lines of text, shown in Figure 1.

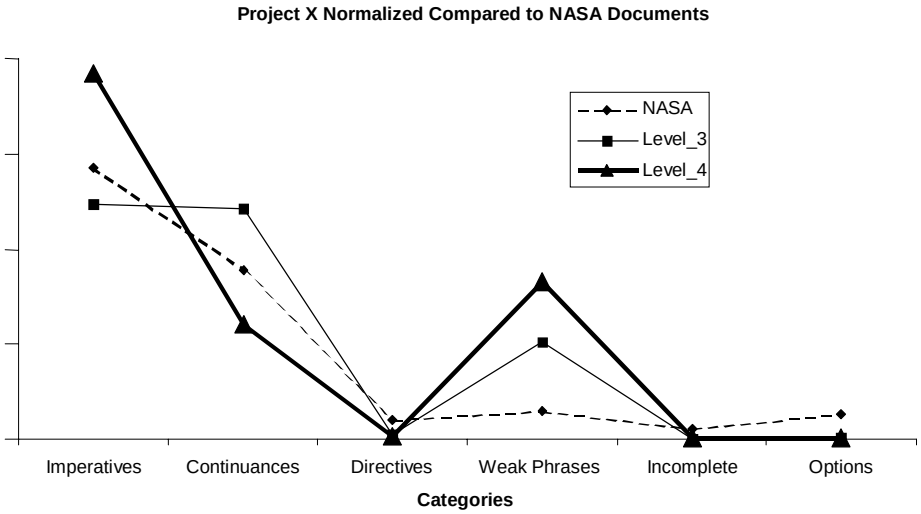


Figure 1 : Project X Normalized and Compared to NASA Documents

From Figure 1 it can be seen that Project X documents are terse without excess extraneous information and with few continuances or directives. This is probably a result of the requirements being analyzed from a database as opposed to a textual document where additional text would be expected. Project X does have a very high number of weak or ambiguous phrases as discussed previously, but few incomplete and optional phrases.

4. Testing Characterization

Once requirements are written, methods for ensuring that the system contains the functionality specified must be developed; this section of the paper discusses three efforts to evaluate testing in the requirements phase. To validate the requirements, test plans are written that contain multiple test cases; each test case is based on one system state and tests some functions that are based on a related set of requirements.[8]

In the total set of test cases, each requirement must be tested at least once, and some requirements will be tested several times because they are involved in multiple system states in varying scenarios and in different ways. But as always, time and funding are issues; while each requirement must be comprehensively tested, limited time and limited budget are always constraints upon writing and running test cases.[2] It is important to ensure that each requirement is adequately, but not excessively, tested. In some cases, the requirements can be grouped together using criticality to mission success as their common thread; these *must* be extensively tested. In other cases, requirements can be identified as low criticality; if a problem occurs, their functionality does not affect mission success while still achieving successful testing.[1,8,9] In order to ascertain the point at which testing benefits become marginal, the SATC developed a third set of metrics based on data available in a requirements management tool using the information design discussed in the previous section.

These metric analyses use the linking information of the requirements to the tests in three ways. The first is to verify that each requirement is tested at least once. The next two analyses characterize the depth and breadth of

[2] For simplicity in this paper, we will use the term *test case* to refer to any type of test for verification and validation of requirement functionality.

the test plan. It is expected that each requirement will be linked to multiple test cases, and that each test case will test multiple requirements.[1,8,9] Data from Project Y is used to demonstrate the metrics application and interpretation in this section.

4A Test Coverage

The first objective is to verify that each requirement will be tested; the implication is that if the software passes the test, the requirement's functionality is successfully included in the system. This is done by determining that each requirement is linked to at least one test case.[6] A query such as those shown below would result in data that could be displayed in a graph shown in Figure 2.

Query: How many requirements in Level 4 Build 1 are linked to a test case?
Query: How many requirements in Level 4 Build 2 are linked to a test case?
Query: How many requirements in Level 4 Build 3 are linked to a test case?

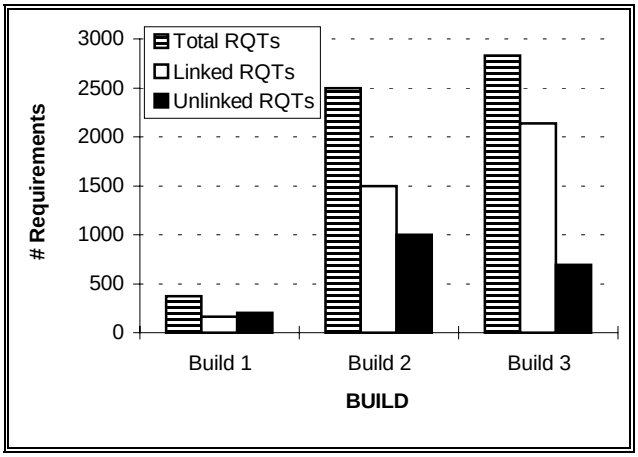


Figure 2: Level 4 Requirement Linkage to Tests

Build 1 appears greater than 60% unlinked due to database problems, the database was not created until after Build 1, hence most of the data from Build 1 was not entered. Build 2 is currently testing Level 4 requirements, but 40% of the requirements are not linked to any test. This data indicates that there is no way to verify whether the functionality of 1,000 requirements is included in the system. It may be possible to link some of the requirements to existing test cases with minimum modification to the test data. If new test cases must be developed, budgetary problems may be created and the testing schedule must be increased. In all cases, further investigation of the missing links is warranted.

For Build 3, just starting the coding phase (coding continues for approximately 10 additional months), only 25% of the requirements are not linked to test cases. This situation needs to be monitored on a monthly basis but is not one for major concern at this time.

4B Test Span

This activity characterizes the test plan and identifies potentially insufficient or excess testing. Requirements are usually tested by more than one test case, and one test case usually covers more than one requirement.[6] Since each test costs money and takes time, the obvious questions are how may requirements are covered by one test, and how many tests cover only one requirement. On the other hand, if requirements are insufficiently tested, functionality may not be verified. The metrics for this analysis are in two parts because of the bi-directional linkage between the requirements and tests. Each direction yields different information. Counting the number of unique tests used for a requirement indicates that requirements at both ends of the graph may have too

much or too little testing. Counting the number of unique requirements tested indicates the exclusivity of the testing.[3] Due to space limitations, we will demonstrate only one direction - unique requirements per test to identify excessive or insufficient testing.

Figure 3 shows an expected profile of unique requirements per test case based on data from NASA projects [5].

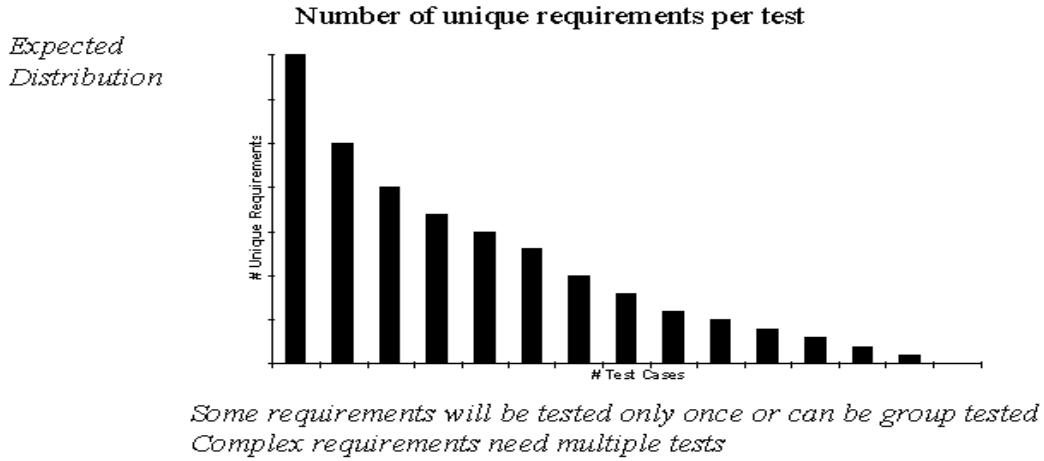


Figure 3 - Test Program Characterization Tests per Requirement

This profile shows that there is an expectation that there will be a large number of requirements tested by only one test case, and that there will be some number of requirements that will be tested by a multiple number of test cases. It is expected that the upper bound of multiple test cases will range in the tens. This makes sense, as more complicated requirements may require different test cases to thoroughly verify all aspects of the requirement. However, there is a limit on the number of test cases. As the number of test cases increases the difficulty of verifying the requirement also increases. This difficulty arises due to the complication in data analysis, understanding the results of the multiple tests cases, and understanding the impact of multiple test case results on the verification of the requirement.

Number of tests per requirements counts the number of unique tests associated with each test. A program query such as the one below might be used.

Query: How many requirements are tested by Test A.1? (Acceptance test, Test1)
Query: How many requirements are tested by Test A.2? (Acceptance test, Test2)
Query: How many requirements are tested by Test A.3? (Acceptance test, Test3)
. . .

The data is then summarized to count the number of unique requirements evaluated by a given test.

This data was compiled for Build 3 Level 4 requirements and is graphed in Figure 3 with bars. The profile curve (solid line) was derived to identify outliers - areas where testing may be insufficient or excessive. The X-axis is the Number of Unique Requirements and the Y-axis the Number of Test Cases.

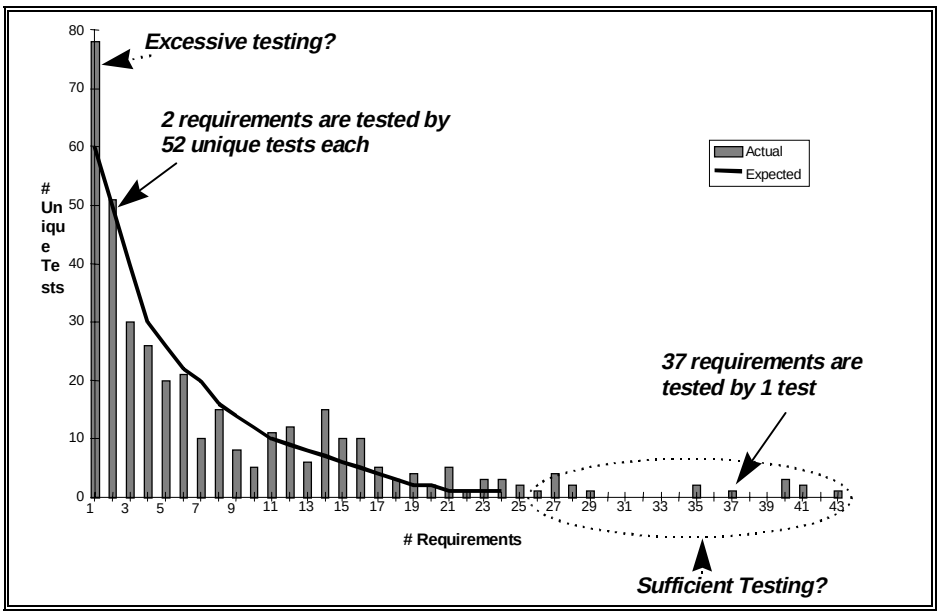


Figure 4: Sufficiency of Testing Plan

The analysis in Figure 3 shows the dilemma of structuring a test program. A testing criterion is to have a one to one relationship between tests and requirements. In this way the validation of requirements is isolated to single tests and so are easily verified. The problem with large systems is that a one to one relationship between tests and requirements will cause a large test program to be developed that will have a huge number of test cases. It will be too costly and too time consuming to complete, due to the number of test cases, the amount of test data required, and the large number of test sessions needed to execute all of the tests. Therefore a balance must be obtained where a one to one relationship between requirements and test cases is developed for critical requirements, but less critical requirements are tested in groups based on system states or functional threads.

4C Test Complexity

Figure 3 indicates there may be excessive testing scheduled for some Build 3 Level 4 requirements due to the large tests per requirement ratio seen on the left hand side of the chart. The next step in evaluating the requirements testing is to investigate the testing magnitude through the complexity of the linkages. One way this can be done is to look at the number of requirements, the number of linkages, and the number of tests. Recall, each link is a connection between a requirement and a test. This data presents a third view of the data previously presented. Figure 4 shows this raw data for Build 2 and Build 3, Level 3 and Level 4 requirements.

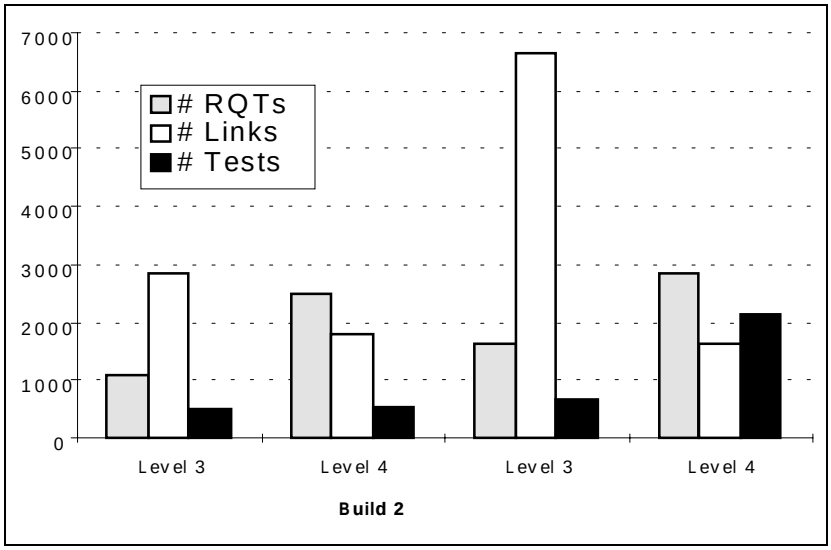


Figure 5: Requirements, Links, and Tests

Looking at Figure 4, it appears the number of links for Level 3 may be disproportionate. Table 2 shows the actual ratios.

Build	Level	Ratio Links to Requirements	Ratio Links to Tests	Ratio Requirements to Tests
2	3	2 : 1	5 : 1	2 : 1
3	3	4 : 1	10 : 1	4.5 : 1
2	4	0.5 : 1	3 : 1	4.5 : 1
3	4	0.75 : 1	4 : 1	1.25 : 1

Table 2: Ratio Links to Requirements

Figure 4 and Table 2 show a number of different perspectives of the test information. First look at the ratio of the number of links to the number of requirements summarized in Table 2 below. In the Level 3 requirements for both Build 2 and Build 3, there are at least two links for each requirement. This means that on average, each requirement is tested by at least two different tests. For Level 4 requirements however, there is less than one link for each requirement. This indicates there are some requirements that are not linked to any test, hence their functionality may not be verified.

Another way of viewing the data is to look at the total number of links to the total number of tests - are there too many? These ratios are also contained in Table 2. Looking at Level 3 requirements for Build 3, the graph in Figure 4 is reinforced by the ratios in Table 2, that show a ratio links to tests of 10 to 1. This indicates that although the number of tests seems adequate for Build 3 (Figure 4), the complexity of the test program is too high; that is, the linkage between requirements and tests is complex (Table 2).

As stated, when metrics indicate a potential problem, further investigation is needed. The tentative conclusion is that the test plan for Build 3 Level 3 requirements is too complex, but the last column in Table 2 indicates the ratio of requirements to tests for Build 3 Level 3 requirements is not out of line. It is likely that the number of tests is sufficient, but the number of links is excessive and may need to be decreased, thus decreasing the complexity of the tests. But while the last column in Table 2 resolves one concern, another is raised. Looking at the ratio of tests per requirement for Build 3 Level 4 requirements, there is a one to one ratio. This indicates a very large number of tests for this Build and Level, suggesting a potential risk of failing to complete testing within schedule and budget. These factors may all be pointing to poor test case design, or, they may simply be a lapse in procedures for requirements and test case management.

As discussed previously in this paper, there are no guidelines for these metrics since they are in research infancy; in evaluating the data in Table 2, we were looking for inconsistencies based on experience.

The objective of an effective verification program is to ensure that every requirement is tested, the implication being that if the system passes the test, the requirement's functionality in included in the delivered system [1,3]. An assessment of the traceability of the requirements to test cases is needed. It is expected that a requirement will be linked to a test case, and may well be linked to more that one test case as shown in Figure 6 [5,6].

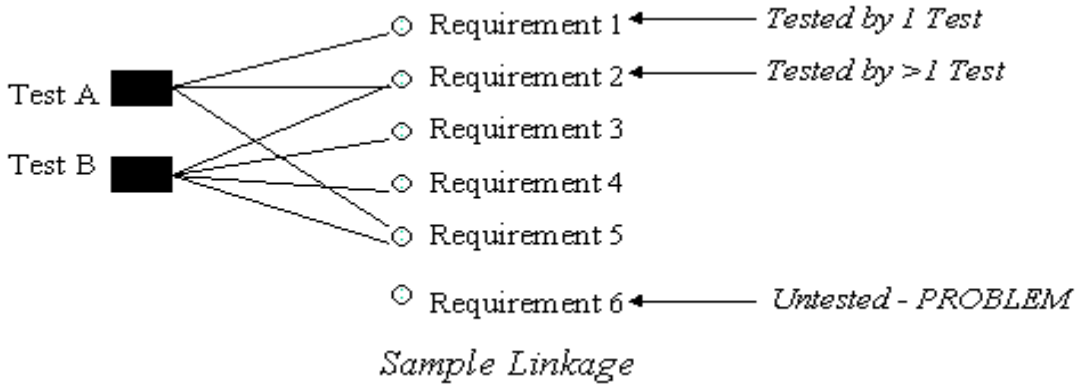


Figure 6 - Requirement Verification - Trace to Test Linkage

The important aspect of this analysis is to determine which requirements have not been linked to any test cases at all.

Figure 7 shows that the traceability of requirements to test cases for Project Y around the CDR time frame for Build 2. The information was extracted from the requirements management database used in support of the development effort. The profiles show several problems.

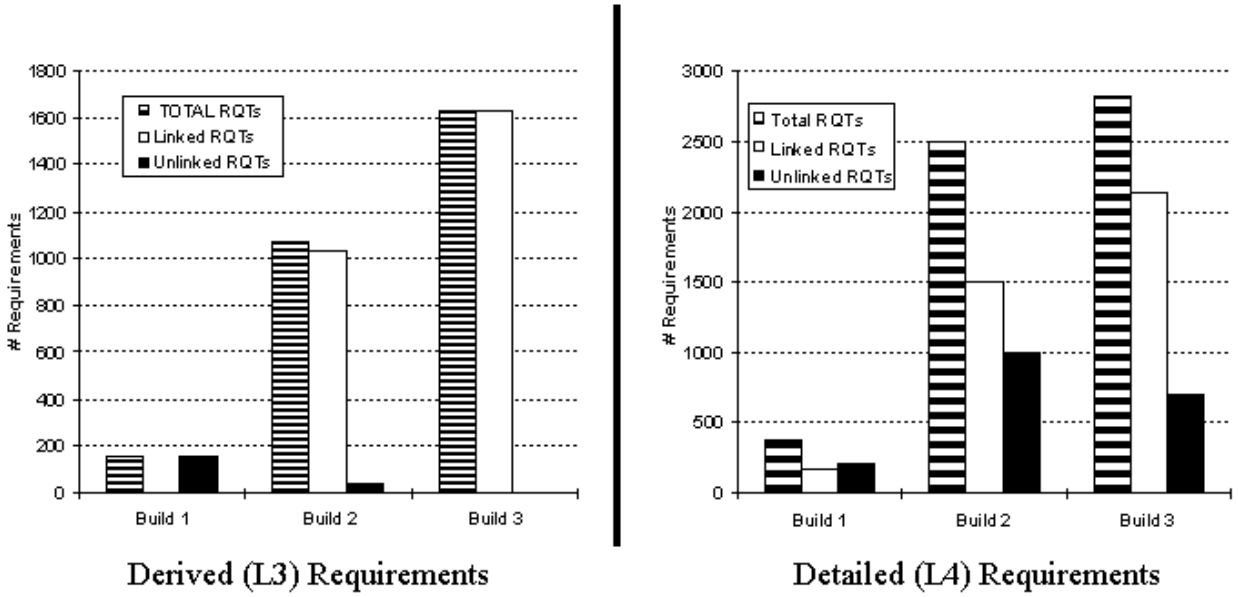


Figure 7 - Requirement Verification Trace to Test

First, the requirements management tool was not used effectively early in the project life cycle. This explains the poor traceability between the requirements and test cases for Build 1. Secondly, there seems to be a mix up in the test priorities by the implementer. The test program for Build 3 is further along than that for Build 2, when it is Build 2 that will be developed and tested before Build 3. Resources may have been inappropriately allocated to the development of the test program for Build 2. Lastly, the test program for the Level 4 requirements is behind that for the test program for the Level 3 requirements. Again, this is backwards. The first tests to be executed will be that for the Level 4 requirements, the system tests, and after that tests for the Level 3 requirements will be executed, the acceptance tests. An explanation for this problem may be found is a previously presented metric. Remember the metric showing the push of Level 4 requirement from Build 2 to Build 3. This movement of requirements from Build 2 to Build 3 may well be the cause of the lack of traceability of requirements to test cases. The test case developers may be having difficulty in keeping up with the changes in requirements resulting in a number of requirements in each build without a link to a test case.

5 Requirement Management

The use of tools to aid in the management of requirements has become an important aspect of system engineering and design. Considering the size and complexity of development efforts, the use of requirements management tools has become essential. The tools which requirement managers use for automating the requirements engineering process have reduced the drudgery in maintaining a project's requirement set and added the benefit of significant error reduction. Tools also provide capabilities far beyond those obtained from text-based maintenance and processing of requirements. Requirements management tools are sophisticated and complex – since the nature of the material for which they are responsible is finely detailed, time-sensitive, highly internally dependent, and can be continuously changing. Tools that simplify complex tasks require skill and a thorough understanding of their capabilities if they are to perform effectively over the lifetime of a project [7].

There are many requirement management tools to choose from. These range from simple word processors, to spreadsheets, to relational databases, to tools designed specifically for the management of requirements such as DOORS (Quality Systems & Software - Mt. Arlington, NJ) or RTM Requirements Traceability Management (Integrated Chipware, Inc. - Reston, VA). The key to selecting the appropriate tool is the functionality (See Table 3 for a comparison of tool capabilities) provided and the capability to develop metrics from the data, secondary contained in the tool.

	Word Processor	Spreadsheet	Relational Database	Requirement Tool
Document config. mgt	X		X	X
Document preparation	X			X
Function decomposition			X	X
Report preparation			X	X
Requirement allocation		X	X	X
Requirement config. mgt		X	X	X
Requirement expansion			X	X
Requirement importation				X
Requirement simplification				X
Requirement storage	X	X	X	X
Requirement traceability			X	X
Test coverage/adequacy			X	X
Metrics			X	X

Table 3 - Requirement Repository Capabilities

The metric capability of the tool is important. It should be noted that most of the metrics presented in this paper to demonstrate how to do requirements the right way were developed from the data contained in a requirement management tool. Table 4 shows a comparison of the metric capability associated with the different tools. Clearly the relational database and requirements management tool provide the capabilities needed to effectively support the management of requirements.

	Word Processor	Spreadsheet	Relational Database	Requirement Tool
Document size	X			
Dynamic changes over time				X
Release size	X	X	X	X
Requirement expansion profile			X	X
Requirement types	X	X	X	X
Requirement verification			X	X
Requirement volatility	X	X	X	X
Test coverage			X	X
Test span			X	X
Test types	X	X	X	X

Table 4 - Requirement Repository metric Capabilities

The selection of a tool is only part of the equation. A thorough understanding of the tool capabilities and the management processes that will use the tool is necessary. The tool should not be plugged into the management processes with no thought as to the impact on the tool capabilities. Adjustments may be needed in the management processes and employment of the tool to bring about an efficient requirements management process.

6 Conclusion

It is generally accepted that requirements are the foundation upon which the entire system is built. And that requirement verification and validation is needed to assure that the functionality representing the requirements has indeed been delivered. However, all too often requirements are not satisfied, leading to a process of fixing what you can and accepting the fact that certain functionality will not be there. A better approach is to get the requirements right the first time, complete, concise and clear, that will provide the implementer a clear blue print with which to build the system. This is not done by magic but through the application of tools and metric analysis techniques in the areas of requirement specification, requirement verification and requirement management

The use of an automated tool to track requirements and their test cases has opened the door to the use of new requirements and testing measures. The ARM tool can be used to point out requirements that may be ambiguous or otherwise poorly worded and thus subject to testing problems. Since the data base that contains the requirements can be repeatedly analyzed, quality trends can be tracked and partial sets of requirements can be monitored and investigated.

The same database contains the test data, which allows new measures that characterize a test program in terms of its structure and complexity, and to assess whether all requirements are verified by test cases. The use of an automated tool for requirements management is essential for gaining insights not otherwise available. The metrics presented here were the result of many different attempts to display and use the data. Key to this analysis was access to the requirement database.

Based on the work done to date, four conclusions can be reached:

- Requirement metrics assist in identifying potential project risks
- Metrics are available in the requirement phase to assess test plans

- Multiple metrics are needed for comprehensive evaluation
- Metric collection is cheaper, faster and more reliable with requirement management tools

REFERENCES

1. Beiser, B. *Software Testing Techniques* (1983), Van Nostrand Reinhold Company.
2. Boehm, B. *Tutorial: Software Risk Management* (1989), IEEE Computer Society Press.
3. Hammer, T., Huffman, L., Wilson, W., Rosenberg, L., Hyatt, L., Requirement Metrics—Value Added in *Proc. Third IEEE International Symposium on Requirements Engineering,* (Annapolis MD, January 1997) IEEE Computer Society Press.
4. Hammer, T., Rosenberg, L., Huffman, L., Hyatt, L., Measuring Requirements Testing in *Proc. International Conference on Software Engineering* (Boston MA, May 1997) IEEE Computer Society Press.
5. Kitchenham, Barbara, Pfleeger, Shari Lawrence, Software Quality: The Elusive Target, *IEEE Software 13, 1* (January 1996) 12-21.
6. Marconi Systems Technology, *RTM User's Manual* (1994).
7. Software Technology Support Center, *Software Test Technologies Report* (August 1994).
8. Wilson, W., Rosenberg, L., Hyatt, L., Automated Quality Analysis of Natural Language Requirement Specifications in *Proc. Fourteenth Annual Pacific Northwest Software Quality Conference,* (Portland OR, October 1996).
9. Brooks, Frederick P. Jr., No Silver Bullet: Essence and accidents of software engineering, *IEEE Computer, vol. 15, no. 1,* April 1987, pp. 10-18.
10. Hammer, T., Huffman, L., Rosenberg, L., Wilson, W., Hyatt, L., "Requirement Metrics for Risk Identification", Software Engineering Laboratory Workshop, GSFC, 12/96.
11. NASA, *Software Assurance Guidebook*, NASA Goddard Space Flight Center Office of Safety, Reliability, Maintainability, and Quality Assurance, 9/89.
12. Wilson, W., Rosenberg, L., Hyatt, L., "Automated Analysis of Requirement Specifications", Fourteenth Annual Pacific Northwest Software Quality Conference, 10/96.
13. Hammer,T., "Measuring Requirement Testing", 18th International Conference on Software Engineering, 5/97.
14. Hammer,T., "Automated Requirements Management – Beware How You Use Tools", 19th International Conference on Software Engineering, 4/98.
15. Hansen, Gary W., Hansen, James V., Database Management and Design, Prentice Hall, 1992.
16. Chen, M., Han, J., Yu, P. "Data Mining: An Overview from a Database Perspective", IEEE Transactions on knowledge and Data Engineering, Vol 8, No. 6, 12/96

Management and Technical Opportunities and Barriers to Applying Statistical Continuous Quality Improvement to Software Quality

by

Mervin E. Muller

Abstract:

Concern about software quality is noted. Obstacles to improving quality are identified. Explanation is given as to why there are no quick fixes to overcome the barriers to improving software quality. Managerial and statistical concepts and processes important for improving software quality are considered. It difficult to find a good operational definition of quality. The conventional life cycle for software engineering is reviewed to draw attention to its limitations and to introduce the use of the MUSH label to focus efforts to improve quality, productivity, and performance through a life cycle of expanded scope. A metaphor, identified as the Four Voices, is used to focus effort to overcome the current major gaps in data needed to support quality improvement processes. Some of Deming's 14 points are cited as relevant for achieving improvements in software. The importance of understanding the goals of an organization so that software deliverables can be assessed with respect to their contributions to the goals of the organization is described. The usefulness of understanding whether an organization invests in software from the objective of cost avoidance or from the objective of value added is explained. The need for Constraint Identification and Critical Success Factors Identification is mentioned. The need for measurements and data to provide a focus on deliverables rather than only having data and measurements related to expenditure of effort is considered. Quality software models are mentioned to illustrate their questionable applicability for use in improving the creation or use of software. The increasing likelihood of buying, rather than building, software is mentioned to draw attention to some of the data and people constraints that need to be addressed when deciding whether to build or buy software.

Background

Mervin E. Muller is the Robert M. Critchfield Professor, Department of Computer and Information Science, Department Chair from 1985-94, and Professor of Statistics, The Ohio State University. He has held academic positions at the University of Wisconsin, Princeton, and Cornell and held senior management and technical positions at the World Bank and IBM He has provided consulting assistance and lectures in 36 countries on information technology, statistics, quality improvement, performance analysis of systems, international development, and financial planning and analysis systems. He is a Fellow of the American Statistical Association and an elected member of the International Statistical Institute.

Address
781 Dreese Labs
1210 Neil Avenue
The Ohio State University
Columbus, Ohio 43210-1277 Tel: (614) 292-4281 e-mail: muller-m@cis.ohio-state.edu

1. Introduction: Why both management and statistical practices are critical to improving software.

One often hears complaints that the development of software historically is associated with the burden of being over budget, missed schedules, providing functionality not needed, and defective implementation. Furthermore, there is the perception that software products and services must suffer from poor quality.

In an effort to understand the root causes of the complaints and perceptions, as well as identify the barriers that need to be removed to improve software, this author had a series of meetings with individuals from organizations either dependent upon software or developing software. Both groups were critical of the academic instructional and research programs offered in software engineering. They wanted more useable guidance and tools to improve quality and productivity. From the academic side I heard the common theme that what was being requested did not belong in the formal academic setting provided by computer science departments. The requests were "too mushy"! From these conflicting points of view, I coined the term MUSH. **MUSH** stands for: **M**ajor **U**ncompromising **S**oftware **H**urdles. Mush is introduced to focus attention on the need to expand the spectrum of the lifecycle of software engineering as well as eliminate the gaps in software engineering tools and practices by addressing the mushy topics.

To address the concerns that have been raised, this paper addresses the need to involve professionals with knowledge and experience in management practices and modern statistical practices. In addition, concern about data poverty, lack of data relevant to improving software engineering is raised. The challenge of creating data relevant to improving software engineering offers many rewards, but removing the barriers to obtaining relevant data will not occur quickly nor easily, as noted in the paper.

Quick fixes to the data problem are not suggested because quick fixes have not been successful. A major barrier is the lack of management acceptance of the need for long-term commitment to quality improvement based upon: (1) an understanding and sharing of goals and values with the workforce of an organization, and (2) a willingness to invest in processes to acquire and use data to understand how to improve software quality. This barrier has mushy aspects that need to be incorporated into the software engineering life cycle that is taught in computer science departments. Removal of this barrier should make it possible to develop processes to acquire data and use statistical tools to evolve better software quality, productivity, and performance. Effective statistical data analysis tools are available, so only brief mention of tools is provided.

An area of data collection and analysis that has had considerable attention is the recording of programming bugs. More attention to processes and data to identify the root causes of the bugs is desirable, but will not be stressed in this paper. The important aspects of improving programming and testing is also not addressed Instead, attention is focused on the mushy aspects of software engineering.

Many of the successes in achieving major improvements in quality of products or services, as opposed to software, during the past 50 years can be traced to the management and statistical concepts and practices emphasized by Deming and others concerned about improving quality of manufacturing in Japan after World War II. Although statisticians such as Deming, Juran, and Morguti (also spelled as Moriguchi) focused on the introduction of statistical concepts and practices in Japan, if one reflects on what they emphasized, one can appreciate that they were successful through gaining management commitment to taking a long-term perspective. This lead to the introduction and joint use of valid statistical practices and effective management practices, the Japanese experience. By analogy, it is unlikely that one will realize significant and sustainable improvement in the quality of software products or services without attention to the selection and joint use of effective and sustainable management techniques and statistical techniques supported by relevant data.

The application of appropriate statistical techniques buttressed by appropriate management practices has enabled organizations worldwide to address successfully challenges to improve quality. Management and statistical concepts needed for sustainable improvements in quality for software are summarized below.

2. Necessary managerial concepts for doing the right thing relative to the goals of an organization.

The management processes that are used must ensure that both the managers and the workers are clear that they are doing the right thing with respect to contributing to the goals of the organization. Only then does it make practical sense to emphasize doing things right. To identify that an activity is contributing to the long-term measurable goals of the organization is often a significant challenge. Giving adequate attention to long-term goals can avoid the pitfalls of only using short-term financial measures to manage an organization with the resulting consequences of poor quality, low productivity, lowered morale of the workers, and risk of lack of sustainablity in the market place. The concern of being driven only by short-term financial measures was often mentioned by Deming as a cause of poor quality, see for example, Deming (1993). Successful leaders balance the short-term needs with the long-term vision and goals.

2.1 Clarity and acceptance of goals: A prerequisite for achieving opportunities for better quality and productivity.

Managers have the responsibility to understand and articulate the organizations goals. They also need to use processes to involve those reporting to them to determine whether or not there is understanding, commitment, and support for achieving the articulated goals. As part of the management process, it may be necessary to modify or replace goals based upon insights provided by workers or customers. The need for changing goals arises if the goals are: (1) not understandable, or (2) not seen as achievable, or (3) better alternatives have been identified. It is also important to understand how the meeting of goals can be monitored.

With a meaningful commitment to shared goals it is realistic to expect that management and workers are able to allocate their efforts to be working on the "right" problem. Then attention can be focused on doing things "right" through attention to quality, productivity, and performance. Most of the tools and practices associated with improving products or services will be ineffective or not relevant if people are not working on the right problem. Without attention to goals and having data, analysis, and reports that directly relate worker output to contributions to the goals of an organization, barriers to improvement will persist and opportunities for improvement will be missed.

2.2 Focus on deliverables rather than expenditures.

Management processes are sometimes described as fostering work through others. Whether or not one accepts this concept, it is vital that the management processes being used foster focusing on deliverables. Work activities should relate to goals and be measured in terms of deliverables that contribute to goals rather than only monitoring and reporting on effort spent. Most of the individuals complaining about limitations in software engineering education and research were still measuring workers by expenditures rather than deliverables. Most of the popular software metrics, if being used at all, upon reflection can be seen to relate to measurements of expenditures and not deliverables. Furthermore, the measurement of effort expended, at best, supports description of expenditures. However, it would be irresponsible to ignore expenditures. When using expenditure measures it is important to use feedback processes with measures and statistical tools to alert both the workers and managers that corrective actions need to be put into use in a timely manner to ensure that expenditures do not reach or exceed out-of-control limits. Statistical process monitoring tools such as describe in Box and Luceno (1997) can be useful.

Metrics that are used can be inhibitors of creativity and commitment, such as measuring contribution of programmers only in terms of lines of code or function points. To achieve improvement of quality, productivity and performance there is need to identify and measure work activities in terms of concrete deliverables. Then workers and managers can focus on aspects of work activities that contribute to products or services.

None of the current widely advertised software metrics directly measure deliverables; for example, neither lines of code(LOC) nor function points(FP) is a meaningful deliverable unless the LOCs or FPs represent a useable unit. Furthermore, I have found that except for budgeting purposes or estimating removal of bugs, the use of LOC or FP can be a significant demotivator for those being measured by them. The use of lines of code or function points can also discourage reuse of code or willingness to consider objectively the purchase of software instead of building software. What is needed for measuring an accomplishment are data that can be used to confirm that a meaningful and useable part of a product or service has been

delivered and that some tangible value can be assigned to the delivered unit. This was the initial intent for the introduction of function points. Examples of a completed software unit could be a program module, subroutine, macro, an input or output icon, a completed object, a module, a test plan, test material, or a completed unit of the documentation of some user requirements or internal or external documentation for some part of a product or service.

3. Some of Deming's contributions that can be relevant to software.

Attention is drawn to the contributions of Deming in order to emphasize the need to take a long-term perspective that involves the managers of organizations. Attention is then drawn to successes in agriculture and manufacturing to encourage those interested in software engineering to give additional attention to the need for investing in better data collection and analysis processes. Simple mechanistic solutions for improving software engineering have not worked. They also did not work in agriculture nor manufacturing. People are required to achieve improvements in software, so we need to alter our focus to having data to help people. In software engineering data collection has often been used in a punitive way with respect to the workforce. Deming fostered the providing of data to the workforce so they could understand how to do their work better.

Deming made remarkable contributions to improving quality which in turn contributed to productivity and performance. Some of his contributions relate to management philosophy, policies, and practices that are necessary to have in place before turning to use the statistical ideas and practices he emphasized. Some of his ideas are captured in his famous 14 points, for example, page 23 of Deming (1986), others relate to some of his favorite sayings. I will cite two of his saying that he and I discussed on many occasions because I believe they relate to the process of creating or using software. The two examples are: (1) "There is no instant pudding" and (2) It is the responsibility of management to give workers tools for them to understand how to do their work better and give them the opportunities to be involved to make suggestions to identify opportunities to improve work processes. I have selected five of his 14 points to mention here although all are worthy of inclusion. These five quotations can be considered a summary of his management concepts that I am stressing. For each of the selected five points I have prefixed the quotation by a brief title. The selected quotations are: (1) Constancy of Purpose: "Create constancy of purpose toward improvement of product and service, with the aim to become competitive and to stay in business, and to provide jobs", (5) Continuous quality improvement: "Improve constantly and forever the system of production and service, to improve quality and productivity, and thus constantly decrease costs", (8) Positive and constructive feedback: "Drive out fear, so that everyone may work effectively for the company", (10) Have meaningful and constructive objectives: "Eliminate slogans, exhortations, and targets for the work force asking for zero defects and new levels of productivity. Such exhortations only create adversarial relationships, as the bulk of the causes of low quality and low productivity belong to the system and thus lie beyond the power of the work force.", and (12) Have meaningful objectives: "(12a) Remove barriers that rob the hourly worker of his right to pride of workmanship. The responsibility of supervisors must be changed from sheer numbers to quality, and (12b) Remove barriers that rob people in management and in engineering of their right to pride of workmanship."

4. Definition of quality and recognizing quality as a process.

One can take many different positions related to establishing an acceptable definition of quality. Two extreme positions are: (1) quality is a semantic abstraction that only has a useful meaning in a specific operational context, or (2) it is obvious what is meant by quality. One may find it challenging to find or develop an operational definition of quality that is useful. Among the references included in this paper, many of them include quality in their title. We will not dwell on whether or not each of the references provides a useful definition. However, if you accept the challenge to scan these references you may find how few of them provide a definition of quality. Two of the extreme definitions used by several authors are: (1) quality is free--I disagree with this statement as an operational definition. However, the statement is sometimes used correctly when it is true that after investing resources to improve the quality of a product or service one can determine that the benefits exceed the cost of improving the quality, or (2) quality is in the eyes of the beholder. When addressing the issue of quality it should be in the context of what is:

- ∅ Needed,
- ∅ Can be obtained or delivered, and
- ∅ Can be afforded.

One of the difficulties in establishing an acceptable definition of quality can occur if one does not take into account that quality is usually a function of several variables as displayed in equation (4.1) below:

(4.1) Quality = F(customer, people, competition, process, product, technology, resources | constraints) +ε.

The people variable in equation (4.1) is intended to reflect the impact that people have on quality when taking into account their knowledge, experience, competence, commitment, work ethic, and availability. Similarly, the technology variable and the resources variable are intended to reflect the impact that technology and resources have on quality through their use in the product and/or process. Equation (4.1) includes conditional dependencies to draw attention to the need to take into account constraints when formulating models or equations of quality. The constraints can related to any of the variables identified, such as failure to allocate resources to establish and validate requirements, or time deadlines, or the unwillingness to recognize the need to develop pilot or prototype capabilities to understand what affects the meeting of quality requirements. The variable ε is part of equation (4.1) to signify that the quality function can have a stochastic component in it, the unexplained random error.

Below are two lists of factors that can come to mind when considering the quality of a product or process.

Quality of Product	Quality of Process
∅ Functionality	∅ Within specifications
Within specifications	∅ Fault detection
Works correctly	∅ Fault removal
∅ Ease of use(ease of learning)	∅ Rework
∅ Availability(reliability)	∅ Availability(reliability)
∅ Performance	∅ Performance
∅ Resource consumption	∅ Resource consumption
∅ Cost(initial, ongoing)	∅ Cost(initial, ongoing)
∅ Maintainability	∅ Maintainability
∅ Recovery from misuse	∅ Recovery from failure
∅ Competitive position	∅ Ease of learning

Quality is not an end in itself. In this context, it is helpful to think of quality as a process, usually an ongoing or continuous processes. If we think of software creation as a process then the data and measurements we need are to elucidate and improve the process. Software quality as a process is dependent upon the major variables cited above. Ultimately what matters is that the product or service has an acceptable performance.

Performance can involve: (1) availability when needed, (2) consistency of response time, (3) handling of peak loads, (4) avoidance of congestion bottlenecks, (5) reliability, (6) graceful degradation, (7) recovery from failure, (8) cost, both initial and ongoing, and (9) security protection. However, to achieve an improvement in performance it is necessary to achieve first an improvement in quality.

Quality can be thought of as a process in the same sense that architecture is a process. The architectural process is conducted to yield, say a useful and attractive building, in the presence of balancing desires, needs, requirements, resources, and constraints. If one is not aware of the importance of addressing constraints throughout a process, one is urged to read the reference by Goldrattt (1990).

If one considers that the architecture metaphor is confined to the development of drawings and blueprints then one will have difficulty in considering architecture as a process that includes the physical construction. However, if one has been involved with the creation of a building of any significant

complexity that meets the needs of the occupants then one has observed a very dynamic and interactive process. The head architect of a successful creation is usually the facilitator of a process to resolve conflicting objectives in the presence of many constraints. Similarly, if one views quality of software engineering as being centered on the construction of programming and its testing then one may also have difficulty viewing it as a process. The limited view of not seeing architecture as a process also occurs with respect to the construction of a building if one considers that the drawings do not represent a process but only as a specification for the preparation of forms to hold the steel and the pouring of the concrete for a steel reinforced concrete building.

5. Why link quality, productivity, and performance as interdependent ?

If one accepts that quality is a process rather than an end in itself, then one ought to be able to accept that quality is an ongoing or continuous process. With each incremental improvement in quality, one can expect a related subsequent improvement in productivity, whether the productivity relates to creating or using a product or service. This occurs because whenever an activity has a quality problem there is waste of time and resources because something needs to be done again, a loss in productivity.

The gap between expectations and realizations in quality improvement in software can be explained in part by the relatively short history of software engineering and the rapidity of technological changes. The challenge to reducing the gap is to recognize the interdependencies among quality, productivity, and performance. Progress to close the gap will depend upon recognizing opportunities, recognizing barriers to be removed, and the willingness to accept the need for better management practices and understanding of how to improve the quality of software engineering by investing in acquiring measurement data through making better use of available statistical tools to aid in obtaining and using relevant data. This should enable one to work smarter rather than harder. It would be desirable to get to the point where with each incremental improvement in productivity and performance it is possible to assess whether or not additional investments in quality improvements are cost-effective and necessary relative to goals.

6. Understanding the commitment to data and analysis in manufacturing and agriculture--an intellectual transfer that is needed for software engineering.

Manufacturing as a metaphor for software engineering is often criticized for not being relevant. When the manufacturing metaphor is dismissed the reason usually given is that once code is written there is no problem of replication, whereas this is a key problem in most manufacturing activities. There are several important factors that are not taken into account if the metaphor is dismissed based upon the issue of replication. The development and maintenance of data and the use of statistical tools have evolved over the past 75 years to support effective and efficient manufacturing.

There are several important ideas that are missed by looking for immediate solutions from Deming's ideas rather than understanding his approaches. He did not offer simple mechanistic solutions. We should consider his approaches to the mushy problems of his day so we can lean from him how to apply them to software engineering without expecting "instant pudding." Deming did not expect "instant pudding." Much of the effort of manufacturing is addressed to (1) understanding requirements and specifications relevant to meeting marketing needs and being competitive, (2) creating pilot or prototype manufacturing capabilities to understand how to meet requirements and specifications, (3) monitoring and controlling the actual manufacturing, (4) monitoring performance of products and services related to maintenance needs, and (5) monitoring customer services. All of these activities are supported by ongoing investments in data collection and analysis supported by modern statistical techniques. The successful efforts of Deming and others in Japan caused major investments in data and statistical tools for a wide variety of manufacturing sectors, world-wide. Deming also was successful in promoting data collection processes that provided the workers with data. It is a major mistake to dismiss the process concepts that have been developed in the manufacturing sectors without thoughtful consideration of their relevance to software.

Agriculture should also be examined for its many successes resulting from major commitments to the creation and use of data. This examination may help one appreciate the need for an ongoing commitment to collect and analyze data needed to improve software engineering. R.A. Fisher and others in England in the 1920s started a revolution in agriculture based upon the use of data and statistical tools. We will not

focus on the data or the tools useful to agriculture, although many of the ideas and tools are relevant. What is important to keep in mind is the ongoing long-term commitment of those working on improving agricultural products to the use of statistical tools for the collection and analysis of data. It is unrealistic to expect instant pudding when trying to decide on the variables that affect yield and quality of an agricultural product. Consider the constancy of purpose related to the revolution in producing rice, wheat, or corn, to mention only three crops. The uncertainty and complexities of producing better agricultural products has immense intellectual challenges. Why should those managing organizations that depend upon software expect to be less committed than those involved with improving agriculture to the need for constancy of purpose with respect to data and statistical tools related to understanding how to improve the building or acquiring of software products or services?

7. The software is a one-time creation?

One of the barriers to gaining adequate support for the investments in data and tools to direct software engineering is the belief that software is produced only once. Sometimes it is acknowledged that software can go through several versions. However, one only needs to address some of the concerns associated with the year 2000 problem to help one realize how often computing applications have gone through many generations of software and hardware. Furthermore, for the belief that software is only created once to have validity, one must also accept that the needs of an organization will remain static in a dynamic world or that new ideas will not be discovered that make it desirable to change the software. If software is created only once then statistical thinking has little to offer to improve software quality.

8. Need for a statistical perspective.

A brief explanation is given here, and in the next section, of why it is essential to expect to use some rather advanced statistical tools to address effectively and efficiently the improvement of quality, productivity, and performance. The correct use of valid statistical tools can be justified on several grounds. There is the desire to: (1) collect valid and representative data as well as avoid collecting unnecessary data, (2) draw valid conclusions from data as well as avoid drawing invalid conclusions with acceptable likelihood, and (3) make decisions or inferences as correctly as possible with as much precision as is affordable.

The need to rely on a statistical approach to a particular problem arises when one is attempting to interpret events or predict future event in the presence of uncertainty and incomplete information. This is certainly the usual situation where creating software products or services. This is where the theory and practice of modern statistics can make significant contributions. When acquiring data it is necessary to understand the importance of sampling and randomization, see for example, Box, Hunter, and Hunter (1978), Cochran (1953), or Deming (1950). Furthermore, to ensure that the we can explain and make valid inference from data involving more than one variable it is necessary to make use of the developments in the design of experiments and response surface methodology. Although it is not obvious, some aspects of modern statistical theory and practice provide tools to protect one against what one does not know. If data have to be collected over more than one time interval, it is necessary to understand how to collect and analyze time series data effectively; see for example, Box, Hunter, and Hunter (1978) and Box, Jenkins, and Reinsel (1994). It is also necessary to be able to interpret and adjust data so that valid conclusions can be made taking into account the impact of one or more shifts in policies, procedures, or practices that have occurred over time. For example, it is vital to understand the immediate and longer-term consequences with respect to quality, productivity, or performance, benefits, or cost, if one or more changes have occurred. A change could result in using a different approach to determining user requirements, or using new or modified software or hardware capabilities , or using new testing procedures, or using a new training method for software engineering. More examples could be provided. What is essential to keep in mind is that if changes in policies, procedures, or practices have been introduced, it is often necessary to use rather computationally demanding advanced statistical tools to understand correctly the data and to interpret correctly the underlying impact of changes. An intervention can have an immediate impact that is not sustainable or an impact that is sustainable but with constant or varying impact. Detecting and understanding the presence of an intervention and the type of impact that may be present require the use of sophisticated statistical tools, especially if the underlying process has any amount of variability. Failure to use Intervention Analysis to take into account the impact of interventions can result in drawing faulty conclusions from data. Using Intervention Analysis as created by Box and Tiao, and supported by the

appropriate time series analysis tools, see Box, Jenkins, and Reinsel (1994), one has tools to interpret data with respect to the presence of past interventions or to make statistically sound forecasts for future events in the presence of past interventions.

Much of the literature on software engineering includes mention of methods or practices to improve quality. Unfortunately, many of the claimed benefits are not realized by those trying to apply the recommended methods or practices. Often the discrepancy that occurs can be explained in terms of the stated claims not having been substantiated by dependable data from well designed experiments or appropriate statistical analysis.

9. Evolution of statistical theory and tools from quality assurance to quality improvement.

There are some rather simple but useful statistical tools that are associated with the quality assurance movement. The initial focus of statistically based quality assurance practice was to use tools for the selection of units for inspection and/or use of control charts to decide if a process needed to be changed to stay within acceptable control limits. Deming and others have made it clear that while inspection may be necessary, inspection does not build quality into a product or service. At a minimum, for quality improvement, one needs to identify and remove the causes of defects so there is not a reoccurrence of past problems. There are some rather simple but effective statistical tools for quality assessment that are often identified as the seven basic quality assurance tools which can be classified into two categories:

I. Data gathering/collection and presentation: (1) counts, (2) measurements, and (3) plots of observations, priority focusing: (4) histograms and (5) Pareto diagrams, (6) Control charts of various types, such as the Shewart-type of control charts, and
II. Root cause and effect analysis: (7) Ichikawa diagrams (Fishbone diagrams).

The basic tools in Category-I have found considerable use in monitoring the occurrence of software bugs. Use of tools in Category-II can help avoid the reoccurrence of old bug problems if the root cause of a bug is detected and if process changes are made to ensure that the bug will not occur in the future.

A very clear and concise summary of the seven basic tools appear in the compact handbook by Brassard and Ritter (1994). They also provide a concise presentation of some useful diagramming approaches: Affinity diagrams, Interrelationships, Matrices, Prioritization Matrices, Process Capabilities, and Trees.

To use the classical control charts to monitor correctly the quality of an activity, one is making several important assumptions; namely, it is assumed that the observations are independent, the process has a mean, and the process is in a state of statistical control (stationary). To determine whether or not n data points are independent and can be assumed to be in a state of statistical control requires the use of rather advanced concepts in time series analysis. Fortunately there are powerful tools available to perform the necessary tasks. If the assumptions are not valid, then rather than using the simple control chart, there are more powerful monitoring tools to use, such as described in Box and Luceno (1997). For example, if the mean of the processing is changing, than instead of use of a Shewhart-like control chart, they illustrate how much more insight can be obtained using Cumulative Sum procedures, see page 74 of Box and Luceno. Being able to decide if the observations are independent is fundamental to a sound analysis. Furthermore, if one is attempting to understand whether two or more factors explain a set of results, and whether or not the factors have interactions that effect the results, it is necessary to use statistical tools beyond those that are a part of the seven basic tools.

The next several levels of statistical tools require more knowledge of theory and application of statistics than the seven basic tools; such as how to: (1) sample, (2) select data and protect against bias, (3) design experiments to estimate the factors of relevance and whether or not the factors interact, under this topic I include the tools for response surface methodology, (4) understand the various kinds of time dependence that may be present in a set of data, (5) select the appropriate variable(s) and measures for analysis or modeling, including selection of transformations of variables to simplify analysis by using Box-Cox transformations, (6) monitor and control processes that do not satisfy the conditions required to use

conventional control charts, (7) perform advanced time series analysis using Box-Jenkins models, transfer functions, and Intervention Analysis, and (8) develop models for forecasting events. These tools also support a shift from quality assurance to quality improvement.

There are powerful statistical packages to support some or all of these statistical concepts identified above. The packages that I am most familiar with are from SCA(1998). Those packages support all of the methods cited above. Examples of these methods which can also be analyzed using the SCA packages are presented in the books by Box and his co-authors. Collectively these statistical tools and computer packages relieve one of the concern for how to perform the calculations for the analysis. Use of the packages enable one to focus on determining what to do with respect to collecting and analyzing data effectively and efficiently. Examples of the use of these tools for an in-depth modeling and time series analysis for 138 time periods is provided by Liu and Muller (1987). Without the use of the powerful tools supported by computing capabilities, little meaningful insight would have been available from the data.

Experienced statistician are often heard to say that the significant information is in the deviations of the data from the underlying model that is being assumed to be correct. There are powerful time series analysis tools to analyze and interpret the deviations to detect failure of assumptions about the validity of the model being used. When a failure of assumptions is detected it is then necessary to modify the model. This kind of model modification is part of the iterative nature of the scientific method that needs to become common practice in software engineering. Some workers in quality assurance like to use the acronym PDCA, Plan, Do, Check, and Act instead of the term, scientific method. I am not aware of available statistical assessments of the software engineering models that are currently in vogue.

10. Commitment barriers in the context of cost avoidance/ cost reduction or value added contributions.

Over the years I have tried to understand why it is so difficult to obtain the kind of commitments to collecting and analyzing data to improve software engineering that Deming demanded and received from organization he worked with who were concerned about quality and success in the market place. Over the years I have often seen organizations change managers of software related activities, change vendors, and change software and/or hardware without achieving sustainable improvements in quality. They have looked for "instant pudding." What they needed was to invest in having clear goals and a commitment to making changes to improve quality based upon relevant data and statistical analyses.

At the risk of over-simplifications, I have found it useful to characterize organizations as being in one of two modes with respect to how management views investments in software, cost avoidance/reduction or valued added. Taking the time to understand which category an organization belongs to can help identify the constraints that need to be removed to achieve commitments needed to support improvements.

∅ Cost avoidance or cost reduction with respect to software. Organizations that fit this classification build or buy software to perform activities to support their business activities. However, they do not create or market software. Consequently, their objective is to minimize or avoid expenditures related to software. For such organizations it is not surprising that they might not be willing to invest in efforts to sustain the collection and analysis of data. This helps to explain their data poverty situation. It also helps to explain why these organizations fail to see the need to look upon software as contributing to goals or providing added value. However, some organizations in this category are dependent upon software as being critical to the operations of their business. They have often made many large and often unsuccessful, or only partially successful, investments in quality assurance units and/or research and or development units related to software. Unfortunately, such units seldom have as their mission the creation and use of data to understand what to do to improve quality, productivity, and performance. At times it seems almost hopeless to ask for a sustained investment in software process improvement from an organization that has fallen into the software cost-avoidance mode.

∅ Value added and contributions of software to product or service. Organizations that fit this classification build or buy software to market software products or services. While they may be very concerned about software cost they recognize that improvements in the software contribute to the value of their products or services. The opportunity to justify investments in data and tools to improve the quality of their software and to improve the productivity and performance of the people involved with software is

easier to justify here in the sense that it is easier to relate these investments to the success of the products or services in the market place that use the software. Furthermore, they in essence can amortize the cost of investments in quality improvement over their present or expected customer base.

11. Need for an extended life cycle model of software engineering to cope with MUSH.

As indicated in the Introduction, the term MUSH was coined to draw attention to the reality that the scope of the typical software engineering life cycle often does not cover some of the factors and tasks that influence software products or services. The software engineering life cycle usually includes activities of specification, implementation, operation, and maintenance. Some consider the life cycle to include the activities of establishing requirements and considering replacement of existing software. The mushy aspects of software engineering relate to the establishment and maintenance of meaningful organizational requirements in the broad context of relevance to vision/mission, goals/objectives, strategy, resources, operating plan, and management of an organization. Replacement is also included under the mushy aspects, although some life cycles models include some attention to replacement.

To address MUSH, management needs to take a long-term perspective consistent with Deming's 14 points, but especially "Constancy of Purpose." There is no "instant pudding" to achieve understanding and acceptance of managerial and statistical practices needed to address effectively the MUSH factors through meaningful and relevant data and analysis. Furthermore, to address MUSH it is important to focus on how to remove the barriers to obtaining commitment to data collection and analysis.

12. MUSH and the extended life cycle for software engineering.

There are many descriptions of the life cycle for software engineering. Two of the best known models are the waterfall and the spiral, see for example Sommerville (1996). The use of these models assumes one begins with the establishment of specifications followed by a series of steps, with or without feedback and recycling among the steps. The steps include the implementation of the software followed by its operational use and maintenance. The establishment and maintenance of the requirements as an iterative step is usually not included in the conventional model, or if included is not given ongoing attention once specifications have been accepted. The subsequent frustrations and disappointments with software products or services can usually be traced to incomplete and or incorrect requirements that have been the basis for the specifications.

In some situations the specifications being used become out-of-date because during the time duration taken for implementation, the requirements need to be changed due to the changes in the needs of an organization. What is needed here is a feedback process that keeps customers or users involved throughout the life cycle, but especially through use of prototypes or early partial releases so that the need for changing specifications or requirements are done as early as possible in the life cycle to achieve the lowest possible cost related to making a change. There seems little disagreement that getting the requirements correct, and keeping them correct, can be shown to be cost-effective.

Below is listed the 12 steps of the extended lifecycle. Note that the extended lifecycle is identified below under the title, " A Multiple Iteration of 12 Steps of the Software Lifecycle." The use of the term Multiple Iteration is intended to remind one that at each step of the life cycle it may be necessary to go back one or more steps As part of the iteration process, at each step, and in parallel with each step, it is suggested that an assessment activity, a meta-step, needs to be executed that takes into account the seven factors listed below under the meta-step description. At each meta-step there are four specific types of decisions to be made relevant to the assessment being performed. The four decisions to be made are listed below the seven factors to be addressed as part of the assessment at each step.

A Multiple Iteration of 12 Steps of the Software Life cycle

(1) Goal(s): identification, collection, and assessment
(2) Requirements (needs): Establishing, reviewing, and assessing, including constraints
(3) Feasibility

(4) Decision to proceed, with or without creating prototype(s)
(5) Specifications to satisfy agreed to requirements and decisions
(6) Design to satisfy specifications of deliverables
(7) Development
    Prototype(s)
    Full scale
(8) Deployment & Conversion, including Parallel Operation and Rollout
(9) Operation, full scale
(10) Maintenance and Service Support Establishment and Operation
(11) Training after full scale operational status
(12) Replacement

Meta-steps: Performed in parallel with the 12 steps of the life cycle

At each step of the life cycle an assessment is made based upon a rational decision process. Here is where data and statistical analyses can contribute to effective management of the life cycle while contributing to quality. The assessment at each step is to take into account the following factors: (1) Reassessment of constraints, (2) Risk exposure and opportunities to reduce risk, (3) Identifiable benefits delivered as part of the step being assessed, (4) Critical Success Factors if earlier steps need to be addressed again, or the CSFs for the current step if it needs to be executed again, and if necessary, modification of the CSFs needed for the subsequent steps,(5) Use of resources, (6) Current estimates of the cost and schedule, and (7) Opportunities to take enlightened and timely actions to make changes, using when possible relevant data and statistical tools.

What is done at each meta-step

(1) Assess **deliverables** from step and decide to: (a) Continue as is, (b) Iterate on given step, or earlier steps, (c) Modify future steps, (d) Terminate all activities
(2) Assessment of **time schedules** for deliverables and decide to: (a) Continue as is, (b) Modify schedules, (c) Terminate all activities
(3) Assessment of **resources** required and resources available and decide to: (a) Continue as is, (b) Acquire additional needed resources, or (c) Release available resources
(4) Decide on the **adjustments** to the activities based upon interactions among the steps using interactions with those involved, all four voices as described below in Section 13.

13. Listening to four voices to acquire data to achieve quality improvement.

13.1 Introduction.

Currently there is a severe imbalance in the effort to collect and analyze data to take into account needs and contributions of all those involved with creating, using, and maintaining software. I have found it useful to employ a metaphor of four voices as it relates to quality improvement of software to focus attention on achieving a more balanced approach to the creation and use of data, see also, Muller(1997). The four voices are: (1) voice of the customer, (2) voice of the process, (3) voice of the manager, and (4) voice of the worker. Each voice is explained below as well as why attention to each voice can contribute to improving software engineering. It is also important to listen for the interdependencies among the voices as well as listening to the individual voices.

13.2 Voice of the Customer (VOC).

Manufacturing organizations often includes a large investment in obtaining the voice of the customer. For example, they use market research to support data collection and analysis to understand what the customer wants, what the customer needs, what the customer can afford, and what competitors offer. Some statisticians play a significant role in gathering data and analyzing the data to obtain a useful voice of the customer. One approach to product design is to factor the voice of the customer into a process that weighs design and feature alternatives. With respect to software, one area where the voice of the customer is getting an increased hearing is in the design of user interfaces. One of the constraints

affecting design of software is to consider only two levels of customers: novices and experts. It is recommended that the number of levels needs to be extended as well as including other important factors; see Muller(1991).

13.3 Voice of the Process (VOP).

Successful manufacturing organizations have effective systems to collect and analyze data to understand the voices of the manufacturing process. Statisticians and computing people have contributed to systems for gathering and using data for monitoring the voice of the process in many sectors of manufacturing. In some manufacturing organizations they have systems in place so that data are identified, collected and analyzed so that when a defect is detected there is a way in most instances to locate the root cause of the problem and make changes in the process so the defect will not occur again. Comparable efforts are needed to establish enhanced processes to collect and analyze data to understand and act on improving the voice of the software process for creation, maintenance, and service of software.

13.4 Voice of the manager (VOM).

This voice is addressed in most manufacturing sectors. It has been my experience that most software metrics that are widely used are to provide data to aid managers. The managers also use some of the metrics to monitor the process. There is considerable opportunity for improvement. For example, the author believes one should expect to use several styles or approaches to management in all software related activities. Furthermore, there should be processes in place to collect and analyze the management styles being used. The voice of the manager needs to be conditional depending upon the interaction of several major variables that include: (1) the level of competence of each worker and supervisor, (2) the level of commitment of each worker, and (3) the complexity of the tasks to be performed. The voice of the manager may be as a coach, participant, controller, or facilitator depending upon each individual's competence and commitment as well as the complexity of the tasks.

13.5 Voice of the worker (VOW).

This author is not aware of major resources committed to obtain the voice of the worker related to software engineering. Deming often emphasized the need to understand and involve the workers because they had so much knowledge of the process. He was often critical of managers who had elaborate data collection and processing capabilities but failed to provide the data and analyses to the workers so they could understand and learn how well they were doing. Furthermore, and most likely even more important, there are seldom effective processes in place to encourage the workers to come forward with useful data and ideas to improve the process. With process data for the workers, the workers would have better understanding of how they are performing and they would be better able to derive constructive insights to improve on their work, work of others, and the entire process, rather than checking their brains when they come to work. The need for cross-functional teams and the use of motivational tools to assist workers to be effective in teams are recognized in many manufacturing sectors. The need for supporting and listening to cross-functional teams is especially important and inadequately supported in many organizations creating or using software, see Creek (1994). Statistics and computers can help here. The challenge is to adopt knowledge and experience from other areas, to apply the knowledge and use of statistics to collect and analyze data to hear the voice of the worker creating or maintaining software. Understanding is needed for how to motivate, direct, and reward individuals to be effective in cross-functional teams creating and maintaining software.

14. Data censorship and data contamination

Attention has been given to the need for data. Using the metaphor of the four voices may help focus attention and priorities to have data to assist the workers. However, one needs to be careful to ensure that the data made available are not censored or contaminated. An example may help clarify what is meant by data censorship or contamination. I was once asked to help an organization understand how they could improve on their meeting of promised delivery schedules and stay within specified budgets. I asked for data on projects that had been successful and on projects that had major problems with respect to

schedules and budgets. The intent was to try to find factors that might explain the differences between the successful and troubled projects. I was provided a database of all project data. Much to my surprise, not a single project had data showing a missed delivery target date or budget overrun. The vice president of the IS organization had mandated that all project data records contain only the current schedule and current budget. The database was always updated without retaining prior targets or commitments. Maybe this is an extreme example of data censorship or data contamination. The vice president was replaced. However, there is less obvious and often unintended censorship or contamination; for example, lack of data on former customers or former workers who stopped complaining.

15. Constraints and Critical Success Factors: Barriers to achieving opportunities for sustainable quality improvement, productivity, and performance.

The need to address constraints when considering how to improve a process, but especially a software engineering process, is emphasized here because without attention to constraints it is unlikely that significant sustainable improvements will be achieved. Goldratt (1990) provides powerful examples of why constraints are important to address. He also illustrates why it is important to rank constraints in their order of importance followed by efforts to remove the constraints in their order of importance. He also provides a conceptual approach to overcome constraints.
Attention to constraints is also mentioned in this paper because the models usually used in software engineering seldom include provision for taking into account constraints; for example, most database models or the object-oriented paradigms usually fail to include the specification of constraints as part of the model or paradigm. The models seldom include provision to specify and take into account capacity constraints, or quality and performance requirements, or cost or time considerations. Attention to the required mix of people skills and experience is seldom given adequate attention in the software modeling process. Attention to constraints in software engineering is seldom addressed in a concrete way with adequate data, until there is a problem during some step of the life cycle. It is important to indicate the presence of constraints but it is not sufficient to merely give passing attention to constraints.

The people-related, constraints can be the most important, even if they are considered by some to be too mushy to be included as part of the life cycle for software engineering.. Constraints due to insufficient resources (funds or time), hardware (computer devices and networks) and firmware are important to address. To remove constraints requires one to develop understanding of why the constraints exists, their relative importance, and how to overcome each constraint in priority order.

A particular type of constraint or barrier can be the failure to identify the factors critical to the success of a software engineering project. The identification and use of Critical Success Factors, CSFs, enables one to establish priorities for what one does, including the order in which constraints of the type mentioned above should be addressed. By understanding what are the CSFs, it should be possible to allocate resources effectively and avoid unnecessary risk as well as enhance the likelihood of success.

16. Data poverty as a major constraint to improving software quality

The purpose of this section is to make it clear that **a** major constraint, if not **the** major constraint to improving quality, productivity, and performance related to software is the lack of data, Data can be lacking to support the product related processes of managing, planning, producing, marketing , or assessing a product. It is tempting to suggest that the improvement of software quality is dependent upon giving adequate attention to issues of management, statistics, and MUSH. With such attention one would expect to evolve the appropriate policies, procedures, and practices. However, from the perspective of someone who has managed and consulted on software projects, I know that the major constraint has not been given sufficient attention, lack of relevant and accurate data. It may be difficult to convince management, or those not frustrated by the need for data, that data are the problem.

This lack of appreciation for data seems especially hard to overcome in view of the dynamic changes taking place, such as the world wide web or the amazing amount of relatively inexpensive software for PCs. In addition there is the expectation that data problems will be resolved by giving attention to the Baldridge Award process, or ISO 9000, or SEI-CMM process.

It is also tempting to shift the problem from the inadequacies of data to the inadequacies of software metrics. However, the major constraint is the lack of understanding and commitment to the need for relevant data to guide one in improving the software life cycle through improvements in policies, procedures, and practices. As already mentioned, part of the practical difficulty is the lack of constancy of purpose.

Under real pressures of time constraints and resource constraints, it is difficult for some organizations to make an ongoing commitment to have sufficient resources to explore how to collect and analyze data to gain insight for improving the software process. One often hears the excuse that there is not time to focus on data needs, or that the field of software is too dynamic to make long-term investments in data. Obtaining better data can be viewed as part of a renewal process. Consequently, it is not too surprising that there is not sufficient time or resources for renewal in the sense of Covey and his metaphor of taking time to sharpen the saw, see Covey(1989).

Most organizations that depend upon software are in the state of data poverty with respect to having data to guide acquisition of software or to make improvements in processes to acquire or use data to understand how to improve software quality, and related productivity and performance. Several organizations that I have visited had made relatively large misguided investments in acquiring data to decide whether to use Lines of Code or Function Points as a useful metric. They are similar to the drunk on a dark night who when asked why he was crawling around on his knees under a lamp post: His first answer was," I lost my car keys." When asked where did you loose the keys, the answer: "Over by the car." When asked why are you looking here, the answer, "because the light is over here." In all likelihood we will continue in the state of data poverty unless the collection and analysis of data is given more attention and resources. Determining what data are needed can be a difficult ongoing task. This is why attention to this constraint is given special emphasis here. Data poverty is compounded by the dynamic nature of software tools that appear so often in the market place as well as the changing roles of software professionals. However, in spite of data poverty, data should not be collected unless it can be justified on a long term basis with respect to assisting in improving quality.

Many organizations have some form of a Help Desk or Problem Reporting System. However, I have observed that most organizations fail to harvest the potential data benefits from such systems as a support for marketing or as an aid to the technical staff using or supporting computing applications.

17. Data poverty and the people dimension of the life cycle.

An especially challenging dimension of data poverty is having data needed to understand the kinds of training to use to improve the quality of the work being performed at each of the 12 steps of the life cycle. For each of the tasks within each of the 12 steps it may be necessary to select a very specific type of training. Understanding how to select a specific type of training can be very mushy. Depending upon the task or area it would be reasonable to have data to take into account the learner effect and the type of instruction effect. Furthermore, it would be desirable to have data to estimate the short-term and long-term benefits that can be associated with making an investment in a particular training offering for an average individual or an individual with a particular set of characteristics. Below are listed some of the factors that would be desirable to include so as to have data to understand and assess the expected impact and value of investing in training.

- ∅ Topic and task level dimension.
- ∅ Training dimension, including coverage level: introductory, survey, in-depth(fundamentals or advanced), or refresher, etc.
- ∅ Type of training: Lectures only, lectures and hands-on exercises, mentoring, etc.
- ∅ Learner dimension: Intelligence, aptitude, knowledge and experience related to the topics, prior demonstrated knowledge, experience, competence, commitment, past performance.
- ∅ Instructor dimension: Some indication of past performance, effectiveness, knowledge, experience, competence.

Another aspect of training is to maintain data to assess the specific benefits realized from the training. Assuming one had the desired data one would expect to have a set of time series of data that would

enable an assessment of the value of the intervention due to training for a particular task or set of tasks for a specific individual or group of individuals.

18. Metrics in relation to MUSH.

The use of lines of code or function points have been useful to some as a metric to estimate size of a project as a step in forecasting the budget and schedule of a project. However, with respect to the mushy aspects, such as a metric for measuring workers in terms of deliverables or productivity or performance, neither LOCs nor FPs have much to contribute. Furthermore, the use of either can be a significant demotivator that encourages protective behavior; such as: "you want to measure me by LOCs or FPs, guess what: You will receive inflated values from either metric." Neither metric takes into account the value, correctness, or contribution of the program code nor the difficulty in determining what or how to code.

Lines of code can be used to estimate the expected number of bugs or the progress made in removing bugs. However, as initially reported by Adams (1984) the detection of a defect does not always relate to a system failure. There is also the problem of data pollution because of incorrect identification of the cause of the bug. There is also the frustration associated with the process that is intended to remove a bug. Those with experience will acknowledge that they have observed situations where bugs get introduced during the process of removing bugs.

A limitation of most metrics currently associated with the software life cycle is that they are descriptive. They provide no insight in what to do differently. There may not be a satisfactory improvement until sufficient investments are made to address the MUSH aspects.

There are some measures that are intended to assist in the design of software so that the software is less complicated and easier to maintain. In spite of such claims, for example the Cyclomatic measure of complexity, the claims do not seem justified, see Chapter 14, by D.C. Ince, in Fenton(1991), or Chapters 8 in Fenton and Pfleeger (1996).

In this paper and in most of the literature on software metrics, the term metric is not correctly used. Are there any of the software metrics that satisfy the conditions of being a metric in the mathematical sense of the distance between two points, i.e. the distance between two points is zero if and only if the two points are the same, the distance between two points A and B is the same, i.e., distance from A to B equals the distance from B to A, and the triangle inequality holds? What is the distance between two programs?

Below are two tables. Table 1 contains data to show where bugs are created in the life cycle. Table 2 has data to show where the bugs are removed. I am indebted to Dr. David Cohen, president of sente, for providing me data for the first column of Tables 1 and 2. I provided the data for the second column. I have collapsed the number of steps associated with the life cycle to ensure that the confidentiality of the data is not compromised. It is important to recognize that bugs persist 10 years after software is released due to introduction of bugs as part of the maintenance activity. The second column of each table is intended to show the hypothetical impact and benefits of an expanded life cycle that includes steps that give attention to MUSH.

There seems to be general agreement that if a bug fails to be removed during the specification step that the cost of removal increases almost as a power of 10 as one proceeds through the later steps of the life cycle. In Table 1 a conjecture is shown that about 50% of all bugs were created as part of the development and use of requirements. If this figure is anywhere near what others experience, it is clear where attention to data and measurement would have a high payoff, the MUSH associated with requirements. One can argue about the other relative values in column under Life cycle (MUSH), but the need to focus on MUSH is what is being addressed.

Table 2 is incomplete with respect to the percentage of bugs removed during the requirements step, because one seldom finds data associated with this step, especially if the requirements are modified after initial approval. The other entries in Life cycle (MUSH) column are missing for lack of data.

Table 1

Sources of Bugs by Activities
according to two models

Activity	Life cycle	Life cycle (MUSH)
Requirements	? %	50%
Design	80%	33%
Testing	10%	9%
Field Use	10%	8%

Data such as appears in the column under Life cycle of Table 2, although not used here, can be used to estimate the mean time to failure assuming some form of an exponential decay in the number of bugs as a function of time . What was surprising when using the data provided by Dr. Cohen to estimate the MTTF was to realize that while there was an approximate five-fold improvement in the MTTF over the ten year period, the MTTF was still about 30 days if the software was used 24 hours every day.

Table 2

Where bugs are removed
according to two models

Activity	Life cycle	Life cycle (MUSH)
Requirements	? %	30-50%
Design	53%	? %
Inc Test	30%	? %
Full Test	10%	? %
Field Use	6%	? %
Bugs after 10 yrs	1%	? %

19. Metrics and Personal quality.

Roberts and Sergesketter (1993) describe and suggest that one establish a “Personal Checklist” to identify those activities that one performs which one would like to improve. The authors describe their experiences and the experiences of others in establishing a checklist of things they do that they would like to improve. Using their personal check list they record on a daily basis all of the occurrences when a defect occurred, such as failing to file a report. The purpose of the “Personal Checklist” is to serve as a focus and to have a count on how well one is reducing the occurrences of undesirable events that one has under their own control which they feel need to be eliminated to achieve improvement in personal quality. It can be difficult to collect data about one’s own activities. I have made several attempts to use the “Personal Checklist”. I have also had others try using it. We all noted some success. The experience was also useful when tried by students because it provided them experience in the obstacles to data collection. Many of us reported how the monitoring of defects became depressing at times. Many of us wanted to have some form of positive feedback to motivate retaining a commitment to improvement. It is highly recommended that one try to establish and use a “Personal Checklist”. At a minimum one should derive a greater appreciation for the demotivating aspects that can be created by a simple data collection task for a measure that provides little if any positive feedback. Furthermore, the process provides no insight in how to improve.

20. Current Software Quality Models.

Many software quality models have been promoted in the past, see Moriguchi (1997) for a current review of many of the models. One of the models included is the well known Walters and McCall Software Quality Model. This is a three level model. At the highest level, the applicability level one finds three entities: (1) Product operation, (2) Product revision, and (3) Product transport. At the next level of the model there are 11 factors such as usability and interoperability. The third level contains 25 criteria. The pictorial presentation of the model shows the connectivity of the criteria to the factors and then the connectivity of the factors to the applicability level. It is assumed that there are data and metrics supporting the use of the 25 criteria. One is hard pressed to find data and metrics that enable the models for software quality to be used to guide software quality improvement This should not be a total surprise, these models are examples of descriptive models that do not define explicitly what is quality or indicate that quality is a process and not an end in itself. What is surprising is that one cannot find explicit attention to how to address quality, productivity or performance when using the models.

21. Build or buy software.

This final section is included to draw attention to a shift that has been taking place on an increased level during the past few years. When one now considers replacing an existing system or developing a new system there are many opportunities to consider buying software rather than building the system from essentially the ground up. Under the current software life cycle and the current set of metrics there is real danger that rational and cost-effective decisions will not be made, especially by those organizations lacking data related to quality, productivity, and performance. Significant attention needs to be given to the MUSH aspects. For example, if an organization is measuring the contribution of an Information Systems unit by LOCs or FPs it would not be surprising that the members of the Information Systems unit would find all kinds of justification for building software rather than buying it. There can be a significant problem in arriving at a rational decision if attention to the MUSH aspects have not been taken into account. There can be serious gaps in the data that are needed, for example: (1) how can one assess the importance of retaining core competency if software is purchased? (2) how can you know what to assess about the product and company supplying the product to be purchased? Failure to address these two questions can have serious long-term consequences for an organization.

Many organizations are either already at great risk, or are approaching a situation of great risk because of undue dependencies on buying software applications. They no longer have significant core competence to assess if their own needs can be met by purchasing and installing Commercial Off the Shelf Software (COTS), or if their own needs can be met by contracting for the purchase and installation of software.

Final Note. . Management attention needs to be given to "walking the talk" of constancy of purpose and indicating how to relate the contributions from software applications to the goals of the organization. There is need for a major commitment to acquire data and use relevant statistical tools to support the application of quality improvement concepts and practices to software engineering, in addition to current practices of quality assurance. Attention and resources to address the MUSH factor of an expanded life cycle of software engineering are also needed to: (1) identify and overcome constraints, (2) create acceptance that application of statistical thinking is critical to improving the understanding of the root causes that affect quality of software or the productivity and performance of those creating, maintaining, or using software, and (3) provide data to the workers (VOW) so their full potential is realized. There are significant obstacles and challenge here, but think of the benefits.

Acknowledgments: I want to thank Dr. Peter Gutterman, Peter Wolfe, and Chuck Adams for their helpful comments and suggestions. I also want to indicate my appreciation to Dr. David Cohen for use of his data.

References

Statistical book on general statistics, sampling, experimental design, control, and time series:

Box, G.E.P., Hunter, William G., Hunter, J. Stuart, (1978), Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building, Wiley.
Box, G.E.P. and Luceno, Alberto, (1997), Statistical Control by Monitoring and Feedback Adjustment, Wiley.

Box, G.E.P., Jennkins, Gwilym M., and Reinsel, Gregory C., (1994), Time Series Analysis: Forecasting and Control, Third Ed. Prentice-Hall.
Cochran, William G., (1953), Sampling Techniques, Wiley.
Deming, William Edwards, (1950), Some Theory of Sampling, Wiley.

CQI related:

Brassard, Michael and Ritter, Diane, (1994), The Memory Jogger II, Goal/PC.
Covey, Stephen, R. (1990), The 7 Habits of Highly Effective People, Simon and Schuster.
Creech, Bill (1994), The Five Pillars of TQM, Dutton.
Crosby, Philip B., (1979), Quality is Free, Mentor.
Deming, W. Edwards,(1986), Out of Crisis, MIT.
Deming, W. Edwards,(1993), The New Economics for Industry, Government, Education, MIT.
Greene, Richard Tabor, (1993), Global Quality, ASQC.
Joiner, Brian L., (1994), Fourth Generation Management, , McGraw-Hill.
Juran, J.M.,(1995), A History of Managing for Quality, ASQC.
Roberts, Harry V; and Sergesketter, Bernard F, (1993), Quality is Personal: A Foundation for Total QualityManagement, Free Press.

Software, Software Engineering , and Quality Related References

Adams, E.,(1984),"Optimizing preventive service of software products," IBM Journal of Research pp. 2-14.
American National Standards Institute,(1994) American National Standard, ASQC. The four documents of this set on quality and management: ANSI/ASQC Q9000-1-1-through 1-4.
Arthur, Lowell Jay,(1985) Measuring Programmer Productivity and Software Quality, Wiley.
AT&T,(1987), Process Quality Management & Improvement Guideline, AT&T.
Bohem, Barry W.,(1981), Software Engineering Economics, Prentice-Hall.
Brocka, Bruce and Brocka,(1992), M. Suzanne, Quality Management, Irwin .
Burr, Adrian and Owen, Mal, Statistical Methods for Software Quality Using Metrics for Process Improvement ,(1996),Thompson Computer Press.
Burrill, Claude W. and Ellsworth, Leon W.,(1980), Modern Project Management Foundations for Quality and Productivity, BEA.
Burrill, Claude W. and Ellsworth, Leon W.,(1985), Quality Data Processing The Profit Potential for the 80s, BEA.
Cho, Chin-Kuei,(1980), An Introduction to Software Quality Control, Wiley.
Conte, S.D. Dunsmore, H.E., Shen,V.Y.,(1986), Software Engineering Metrics and Models, Benjamin/Cummings.
Evans, Michael W. and Marciniak, John J.,(1987), Software Quality Assurance & Management, Wiley. .
Fenton, Norman E.(1991), Software Metrics, Chapmann & Hall, 1991 , London.
Fenton, Norman E., and Pfleeger, S.L.,(1996), 2nd Edition, Software Metrics: A Rigorous Approach. International Thomson Computer Press.
Gause, Donald, C. and Weinberg, Gerald M.,(1989), Exploring Requirements Quality before Design,Dorset.
Ghezzi, Carlo; Jazayeri, Mehdi; Mandrioli, Dino,(1991),Fundamentals of Software Engineering,PrenticeHall.
Gitlow, Howard, Oppenheim, Alan, and Oppenheim, Rosa,(1990), Planning for Quality Productivity & Competitive Position, ASQC.
Gitlow, Howard, Oppenheim, Alan, and Oppenheim, Rosa, (1995), Quality Management, Irwin.
Glib, Tom, Principles of Software Engineering Management, 1988, Addison-Wesley.
Grady, Robert B, and Caswell, Deborah I, (1987), Software Metrics: Establishing a Company-wide Program, Prentice-Hall.
Heitzel, Bill,(1988), The Complete Guide to Software Testing, Second Edition, Wiley.
Humphrey, Watts S.,(1990), Managing the Software Process, Addison-Wesley.
Humprhrey, Watts S.,(1995), A Discipline for Software Engineering, Addison-Wesley.
Jones, Capers,(1991), Applied software Measurement: Assuring Productivity and Quality, McGraw-Hill.
Jones, Capers,(1994), Assessment and Control of Software Risks, Yourdon.

Jones, Keith A.,(1993), Automated Software Quality Measurement: Computer-Assisted Information Resource Management of Applications in IBM Mainframe Environments, Van Nostrand Reinhold.
Kan, Stephen H. Kan,(1995), Metrics and Model in Software Quality Engineering, Addision-Wesley.
Liu, Lon-Mu and Muller, Mervin E., "Statistical Methods and Software for Productivity Improvements and Modeling Time Series Data,",(1989), Recent Developments in Statistics and Their Applications, Klein, J. and Lee, J. editors, pp 227-266, Freedom Academy.
Maguire, Steve,(1994), Debugging the Development Process, Microsoft.
Meyers, Glenford J.,(1976), Software Reliability Principles and Practices, Wiley.
Mills, Harlin D.,(1988), Software Productivity, Dorset.
McCarthy, Jim,(1995), Dynamics of Software Development, Microsoft Press.
McClure, Carma, (1992), Three R's of Software Automation, Re-engineering, Repository, Reusability, Prentice Hall.
McConnell, Steve, (1993), Code Complete: A Practical Handbook of Software Construction, Microsoft.
Moriguchi, Shgechi,(1997), Software Excellence, Productivity.
Muller, Mervin E.,(1991), "Critical Success Factors to Exploit Information Technology Effectively in the 21st Century," Proceedings, SEARCC Conference on Information Technology, v.1, pp1-40.
Muller, Mervin E., (1997) "Quality Improvement in Software Engineering, Proceeding of the 51st Session of the ISI, Turkey, to appear.
Mynatt, Barbee Teasley,(1990), Software Engineering with Student Project Guidance, Prentice-Hall.
National Institute of Standards and Technology, Malcolm Baldrige National Quality Award 1998 Criteria for Performance Excellence, (1997), NIST.
Paige-Jones, Meilir,(1985), Practical Project Management, Restoring Quality to DP Projects and Systems, Dorset.
Perlis, Alan J., Sayward, Federick G., and Shaw, Mary,(1981), Software Metrics, MIT.
Pfleefger. S.L., 1998, Software Engineering: Theory and Practice,(1998),Prentice Hill.
Putman, Lawrence H. and Meyers, Ware,(1992), Measures for Excellence Reliable Software on Time, within Budget, Yourdan Press.
Pressman, Roger S.,(1992), Software Engineering A Practitioner's Approach Third Edition, McGraw-Hill.
Ryan, Thomas P,(1989), Statistical Methods for Quality Improvement, Wiley.
SCA: Scientific Computing Associates,(1998), Chicago, Il.
Schach, Stephen R., (1993), Software Engineering, Second Edition, Irvin.
Schulmeyer, G. Gordon,(1990), Zero Defect Software, McGraw-Hill.
Sigwart, Charles D., Van Meer, Gretchen L., and Hansen, John C.,(1990), Software Engineering: a project oriented approach, Franklin, Beedle & Associates.
Sommerville, Ian,(1996), Software Engineering, Fifth Edition, Addison-Wesley
Vincent, James, Waters, Albert, and John Sinclair,(1988), Software Quality Assurance, Volume I, Practice and Implantation, Prentice Hall.
Weinberg, Gerald M,(1991), Quality Software Management, Volume I Systems Thinking, Dorset.
Weinberg, Gerald M,(1993), Quality Software Management, Volume II First-Order Measurement, Dorset.
Weinberg, Gerald M, and Gause, Donald C.,(1989), Exploring Requirement Quality before Design, Dorset.
Wesner, John W., Hiatt, Jeffrey M., Trimble, David C.,(1995), Winning with Quality Applying Quality Principles in Product Development, Addison-Wesley.

Theory of constraints:

Goldratt, Eliyahu M.,(1990),. Theory of Constraints, North River.

Some reference for learning more about team processes:

Fritz, Robert , (1984), The Path of Least Resistance, Fawcett Columbine
Parker, Glen M, (1994), Cross-Functional Teams, Jossey-Bass
Senge, Peter M, (1990), The Fifth Discipline, Doubleday

Scholtes, Peter R , (1988), The Team Handbook, Joiner.

Metrics and the Financial Health of Software Organizations : Is There a Relationship?

Abstract

As part of a Department of Defense initiative to enhance the state of software engineering management, Carnegie-Mellon's Software Engineering Institute developed a model software development organization. The Capability Maturity Model (CMM) identifies various levels an organization evolves through as it strives to improve it's processes. Using a combination of questionnaires and on-site interviews, assessments are made of the maturity level of a software development organization.

The CMM has altered the landscape for software vendors contracting with the Department of Defense. Additionally, other government agencies are beginning to utilize this model as a basis for selecting qualified bidders. With the federal government being one of the largest consumers of software, it is easy to envision the profound impact on software engineering this model has and will continue to have on the software industry.

One of the attributes that is deemed desirable by the CMM is the presence and active use of a metrics program. A metrics program is seen as a tool that promotes comprehensive process measurements and analysis. The concept behind the metrics program is to provide a closed loop system in which metrics is used to calibrate the development process. For those who have worked in a quality oriented manufacturing environment, it is easy to see the merit of such programs.

This paper attempts to examine the relationship between metrics programs in companies within the prepackaged software industry and their financial health.

Author Biographies

Currently a student at Oregon Graduate Institute, Paul Doherty is a Certified Management Accountant who holds undergraduate degrees in economics and computer science. He has experience as a Quality Assurance Engineer and has held senior management positions in the manufacturing sector.

An associate professor from the University of Portland's School of Business, Don Springer obtained his PhD from Colorado State University. He has extensive experience in the IS field, as both an educator and participant. He has performed a wide range of IS functions, including IS Director, in a variety of industries.

pauld@cse.ogi.edu
springer@uofport.edu

Metrics and the Financial Health of Software Organizations : Is There a Relationship ?

Introduction

As part of a Department of Defense initiative to enhance the state of software engineering management, Carnegie-Mellon's Software Engineering Institute (SEI) developed a model software development organization. The Capability Maturity Model (CMM) identifies various levels an organization evolves through as it strives to improve it's processes. Using a combination of questionnaires and on-site interviews, assessments are made of the maturity level of a software development organization [1].

The CMM has altered the landscape for software vendors contracting with the Department of Defense. Additionally, other government agencies are beginning to utilize a derivative of this model as a basis for prequalifying bidders [2]. With the federal government being one of the largest consumers of software, it is easy to envision the profound impact on software engineering this model has and will continue to have on the software industry.

One of the attributes that is deemed desirable by the CMM is the presence and active use of a metrics program. A metrics program is seen as a tool that promotes comprehensive process measurements and analysis. The concept behind the metrics program is to provide a closed loop system in which metrics is used to calibrate the development process [3]. For those who have worked in a quality oriented manufacturing environment, it is easy to see the merit of such programs.

Metrics have been well received in the financial and manufacturing environments. Changes in financial statements and associated ratios send stock prices scurrying and typically trigger an analysis of the resulting changes. In manufacturing facilities, quality control programs ensure that unacceptable deviations from specification are identified and rectified immediately. Leading edge organizations spend considerable resources tracking financial and manufacturing metrics and they are typically used as a management tool to guide the organization. With this backdrop, one would expect that there would be a relationship between firms that utilize metrics programs and their profitability. Curiously, there is very little empirical evidence that suggests that software development organizations that have active metrics programs produce better software much less produce better financial results. As software development represents an ever increasing amount of the nation's resources, it is critical that empirical research be undertaken to examine the relationships between metrics and an organization's financial health.

Research Problem

The focus of this research paper is to examine evidence to determine if those software organizations that have active metrics programs have better financial results than those that do not. If the CMM model is sound, one would expect that organizations with active metrics programs would on average be financially stronger than those without a program. While a decree for a government department, all be it a very powerful department, may be sufficient to move many organizations to develop a metrics program, in the absence of on any strong financial incentives, it is unlikely that these programs will become engrained in the management psyche of most software development companies. The U.S. auto makers spent significant sums of money on quality programs only when it became apparent that consumers would no longer accept the poor workmanship that routinely left the domestic car plants. While profits for the domestic car manufacturers have reached record levels in recent years, management has seen fit to keep and more often increase spending in quality programs because it is financially prudent. Given that software development organizations have the same profit motive as auto companies, one would expect that unless there is evidence that metrics programs are financially sound, it is unlikely that they will be widely used.

There are a number of difficulties with the analysis that make this research susceptible to criticism. Clearly, it is foolish to suggest that there is an absolute correlation between a metrics program and the financial health of an organization. However, the concept of continuous process improvement and the use of tools such as a metrics

program is widely accepted in management circles in a number of industries. If one assumes that the software development process is one that can be managed successfully, one would expect that a well run metrics program is an integral part of the tool set management would use to assist it.

Another area of concern dealt with isolating software development organizations and their associated financial health. Clearly, there are many organizations that make substantial software development investments yet their primary focus is not selling software. The large financial institutions are one such example. In many cases these investments have lead to a tremendous strategic advantage. However, it was deemed too difficult to segment the gains/losses associated with software development expenditures in these organizations much less to correlate these figures with the presence of a metrics program. Therefore, the initial analysis was limited to publicly traded companies in the prepackaged software industry.

Another concern with this research is that the vast majority of software development organizations are not publicly traded. By using a sample drawn from publicly traded software companies, it is expected that these companies would represent some of the healthiest and mature software organizations in existence and represent an excellent proxy for the industry.

Methodology

The research consisted of two components. The first component dealt with a survey of the publicly traded companies in the United States with a standard industry code (SIC) indicating prepackaged software. The second component of the analysis involved a series of interviews with Information System Directors. It was hoped that this two pronged approach provided both sufficient breadth and depth from which to make accurate assessments as to the state of metrics programs within publicly traded companies.

A listing of all publicly traded software companies in the United States was obtained from a local brokerage house. The sample included 376 companies. Equal dollar sampling was used to select the sample participants. In equal dollar sampling, every dollar of industry revenue has an equal chance of being selected. Under this technique, one would expect that on average the sample would be heavily weighted with companies that represent some of the industry's biggest revenue producers. There were thirty-three companies selected to participate in the survey. During the course of the survey, two of these companies were sold. To avoid any potential influence on the results, the surveys from these two companies were not considered in the final results. With respect to the financial health of the organization, the firms return on investment as computed by FactSet Data Systems, an on-line investment research service served as a proxy measure.

The survey (Exhibit #1) was based largely on measurement questions used by the SEI in assessing the maturity levels of companies. Additional questions were added to the survey to explore issues such as the rationale for a company not having a metrics program, the type of metrics collected, and how metrics programs are justified financially for those with active programs. Input from software educators and sociologists was used in developing the survey. In an attempt to validate the survey, an initial pilot study of seven local companies was examined. The preliminary survey not only provided very interesting results, it also provided feedback with respect to the survey itself.

Many of the survey questions used were taken from the CMM questionnaire and it was uncertain that these questions accurately reflected the state of metrics programs of the responding company. Further, the authors have not been formally trained in CMM assessment. In the end, it was decided that use of portions of the CMM questionnaire was the best course of action. It not only avoided the risk of injecting personal biases into the survey questions but also represented a body of work that has been widely accepted.

One of the concerns with the study involved attempting to determine what exactly constituted an active metrics program. For the purposes of this study, there was two criteria used to determine the presence of an active metrics program. The first criteria was that the company had to answer yes when asked if they had an active metrics program. The rationale behind this criteria was that the survey participants are most often in the best position to assess their company. The second criteria was the responses to the first fifteen questions on the survey – the same

questions used by the SEI assessment process. Positive responses were required in at least thirteen of the fifteen questions.

The focus of the interviews was to validate the data obtained from the survey and also to further explore why metrics programs had not been established or to identify substantive instances in which metrics programs made significant contributions. In the case where metrics programs existed, efforts were made to determine how the cost justifications were made for the program.

Participant Profile

One of the difficulties with this type of research is to balance the interests of its readers with that of its survey participants. Many companies are unwilling to participate or only participate on the condition that they not be identified so as to avoid disclosing proprietary information. Readers, on the other hand, want to know what others companies are doing, how they are solving problems facing the industry. The following section attempts to balance these competing interests.

In all but a few cases, the survey respondents represent some of the largest software organizations in the world. For the most part, they are companies that are household names. Some of the software industry sectors that are represented include connectivity, database, educational, gaming, office productivity, desktop publishing and CAD.

With respect to the maturity profiles of the survey respondent, the intent of the survey was to assess the presence of an active metrics program and not the maturity level of the companies involved. However, the according to the CMM model, companies without an active metrics program would be have a maturity level of no higher than level 3 [3].

Results

Of the thirty-one companies surveyed, a total of twelve responses representing approximately 41 % of industry revenue were received. In all but one survey, there was no evidence that any of the companies had active metrics programs.

The company that was deemed to have an active metrics program had a return on investment (ROI) in the upper 20% range. This compared favorably with the industry average of (12.73) %. This company had revenues in excess of $500 million. The average ROI for companies within the $500 million and over category was slightly over 19 %. This particular company exceeded the 19% average return by over 40 %. In short, this organization had outstanding financial results.

One of the most interesting aspects of the survey came from the company that had an active metrics program. When asked to identify barriers to expanding metrics programs, they identified the lack of automation. Software companies are acutely aware of personnel costs and every effort is made to automate processes to ensure the best utilization of labor. This company saw automation of process support as a means to improve metrics coverage at lower overall cost. It was clear that metrics were very much a part of the development process in this company. No formal attempt has been made to develop a cost-benefit analysis of the metrics program but the team member was certain as to the programs bottom-line benefits. They viewed metrics as “part of doing business” and that their customers demanded this type of approach.

As for the companies that did not have active metrics programs, a number of reasons were cited. One company had outsourced much of the development effort in the past and did not require that its suppliers have a metrics program. They intended to move much of the effort in-house and it was hoped to have a metrics program running “in the next few months.” Another company was uncertain as to which metrics to use so they decided not to use any. Most companies cited a lack of resources as the reason why they did not have a metrics program. One of the most interesting surveys suggested that the company’s roots and associated “hacker mentality” precluded a metrics

program. One participant noted that management supports metrics programs but when schedules slip, it no longer becomes a priority and much of the effort is lost.

Judging by the responses to both the pilot study and the actual survey, there appears to be a strong awareness of the CMM. In all but two cases, the survey participants had heard of the CMM. Given the diverse backgrounds of those in the software industry, this result should be encouraging to the SEI and others, as it appears that their message is reaching its intended audience.

Summary

The CMM is widely regarded as one of the premier models for software development organizations and yet metrics programs, an integral component within the model itself, is largely ignored in practice. The survey raises a number of interesting issues with respect to the software development process and the CMM that require further exploration. It may be that the model is better suited for certain sectors within the software industry. It may be that the model itself is flawed. Many point to the development processes of a few industry leaders whose software development processes differ from the CMM model, particularly with respect to metrics programs, as a more viable model.

The lack of congruence between the CMM model and the standard operating practices of publicly traded companies in the software industry should be very disconcerting to many. The software industry is one of America's most prolific industries and as we move into the 21st century all indications are that growth will continue into the next century. If it is assumed that the CMM is the most desirable software development model, it is imperative from both a corporate and national perspective that the software industry reexamines their current development procedures. With respect to metrics programs, a host of additional research issues are raised by this survey. Listed below are a handful of these issues; there are undoubtedly more.

- identify what sectors of the software industry are more apt to have metrics programs
- determine why these sectors deem metrics valuable.
- determine the actual costs, both startup and ongoing costs, and associated benefits of metrics programs over an extended period of time.
- identify the barriers to the introduction and ongoing maintenance of these programs and how these barriers may be overcome.

Propelled by the hardware and software industries, it is said by many that we are embarking on the Information Age. Companies are beginning to place strategic value in their ability to develop, cultivate and ultimately respond to data from both internal and external sources. One cannot help but note that the software industry, an industry so closely aligned to the Information Age, chooses to largely ignore a vast internal data store, it's own development process. A metrics program attempts to tap into this information base yet as evidenced by the survey, many software companies do not see this as valuable.

The U.S. auto industry should provide ample insight into an industry that lost focus and ignored quality issues for many years. In the late 70's and early 80's the U.S. auto makers pushed poor quality products out of their factories. The offshore manufacturers were only too happy to address consumer quality concerns. As domestic factories idled, the U.S. auto companies were forced to recognize that quality issues were not something to be ignored. Huge investments in quality programs were made. Individuals with quality control backgrounds were some of the most sought after in the industry. Today quality control managers wield enormous power in any auto plant, domestic or foreign, a far cry from a few short decades. While the U.S. software industry has a stranglehold on the global market today, it only need look at the auto industry to comprehend how fleeting this may be. Quality control and metrics programs became a major part of the business process in the auto industry when the auto executives realized that they were critical to their company's financial success. It is clear from this survey that the majority of software development organizations do not see these programs as being very valuable.

Currently, there is considerable discussion regarding exempting all taxes on goods sold over the Internet for a period of five years. Proponents argue that this incentive would foster development of an ecommerce industry. One can only wonder whether it might not be prudent to consider tax incentives to encourage software companies to develop metrics programs. As painful as it may be to develop an industry, it is more painful to lose it.

References

1. Humphrey, Watts S. *Managing the Software Process*. Reading, MA: Addison-Wesley, 1989

2. Terry B. Bollinger and Clement McGowan, *A Critical Look at Software Capability Evaluations*, IEEE Software, July 1991, pp 25-41.

3. Roger S. Pressman, *A managers guide to software engineering*, McGraw-Hill 1993.

Exhibit # 1

Dear Colleague,

As part of a graduate course in software metrics at Portland State University, we are investigating the use and results of metrics programs in the prepackaged software industry. Much has been written about the benefits of metrics programs. Proponents argue that metrics will result in an improved software development process that would, on balance, lead to improved software quality and lower development costs. On average, one would expect that those organizations with lower development costs and higher quality would be more profitable. Yet, we are unable to locate any empirical evidence that demonstrates a relationship between the presence of a metrics programs and the improved financial health of the organization. It is hoped that this research will shed some light into the financial merits of a metrics program. Your firm was randomly selected from a list of publicly traded companies involved in prepackaged software development. It's financial results are a matter of public record. It is hoped that by completing the following survey (approximately 15 minutes in duration), you can further the knowledge of software engineering practices in the future.

In appreciation of your effort, we will provide you with the results of the research as soon as they have been compiled. Thanks in advance.

Paul Doherty
Don Springer

Acknowledgment

Some of the questions used in the survey were developed as part of Software Engineering Institute's Capability Maturity Model at Carnegie Mellon University. In accordance with their permission to reproduce, listed below are the applicable copyright and "No Warranty" statements.

Section 1 - Respondent Background

1. What best describes your current position ? (Please mark as many boxes as apply)

   [ ] Project or Team Leader
   [ ] Technical Member
   [ ] Manager
   [ ] Software Process Group Member
   [ ] Other (Please specify).

2. What activities do you currently work ? (Please mark as many boxes as apply)

   [ ] Software Requirements
   [ ] Software Design
   [ ] Code and Unit Test
   [ ] Test and integration
   [ ] Software Quality Assurance
   [ ] Configuration Management
   [ ] Software Process Improvement
   [ ] Other (Please specify)

3. What reporting unit (division) do you work for ?

4. Are you familiar with the Software Engineering Institute's (SEI) Capability Maturity Model (CMM).

   [ ] Yes [ ] No

5. What is your software experience in : (Please specify for each category)

   Your present organization ? [ ] Years
   Your overall software experience [ ] Years

Section II - Instructions

Below most questions there are boxes for the four possible responses.

Select **Yes** when the practice is well established and consistently performed. The practice should be considered well-established and consistently performed as a standard operating procedure.

Select **No** when the practice is not well established and inconsistently performed. The practice may be performed sometimes, or even frequently, but is omitted under difficult circumstances.

Select **N/A** when you have the required knowledge about the project or organization and the question asked, but you feel the question does not apply to the project.

Select **Don't know** when you are uncertain about how to answer the question.

Feel free to add comments for any elaborations or qualifications about your answers to the questions.

Unless advised otherwise, answer all question and check one of the boxes for each of the questions.

Section III Survey

1. Are measurements used to determine the status of activities performed for managing the allocated requirements. (e.g. total number of requirements changes that are proposed, open, approved, and incorporated into the baseline) ?

   [ ] Yes [ ] No [ ] N/A [ ] Don't know

   Comments:

2. Are measurements used to determine the status of the activities for planning the software project(s) (e.g. completion of milestones for the project planning activities as compared to the plan) ?

   [ ] Yes [ ] No [ ] N/A [ ] Don't know

   Comments:

3. Are measurements used to determine the status of activities for software tracking and oversight (e.g. total effort expended in performing tracking and oversight activities)?

   [ ] Yes [ ] No [ ] N/A [ ] Don't know

   Comments:

4. Are measurements used to determine the status of activities for software subcontracts (e.g. schedule status with respect to planned delivery dates and effort expended for managing the subcontract)?

   [ ] Yes [ ] No [ ] N/A [ ] Don't know

   Comments:

5. Are measurements used to determine the cost and schedule status of activities performed for Software Quality Assurance (SQA) (e.g. work completed, effort and funds expended compared to the plan) ?

   [ ] Yes [ ] No [ ] N/A [ ] Don't know

   Comments:

6. Are measurements used to determine the activities for software configuration management (e.g. effort and funds expended for software configuration management activities) ?

   [ ] Yes [ ] No [ ] N/A [ ] Don't know

   Comments:

7. Are measurements used to determine the status of the activities performed to develop and improve the organization's software process assessment (e.g. effort expended for software process assessment and improvement) ?

   [ ] Yes [ ] No [ ] N/A [ ] Don't know

   Comments:

8. Does the organization collect, review and make available information related to the use of the organization's standard software process (e.g. estimates and actual data on software size, effort and cost; productivity data; and quality measurements) ?

   [ ] Yes [ ] No [ ] N/A [ ] Don't know

   Comments:

9. Are measurements used to determine the functionality and quality of the software products (e.g. numbers, types, and severity of defects identified)?

   [ ] Yes [ ] No [ ] N/A [ ] Don't know

   Comments:

10. Does a project or projects follow a documented plan for conducting quantitative process management.

    [ ] Yes [ ] No [ ] N/A [ ] Don't know

    Comments:

11. Is the performance of the project's defined software process controlled quantitatively (e.g. through the use of quantitative analytical methods) ?

    [ ] Yes [ ] No [ ] N/A [ ] Don't know

    Comments:

12. Does the project follow a written organizational policy for measuring and controlling the performance of the project's defined software process (e.g. project plans for how to identify, analyze and control special causes of variations) ?

    [ ] Yes [ ] No [ ] N/A [ ] Don't know

    Comments:

13. Are measurements used to determine the status of the activities for managing software quality (e.g. the cost of poor quality)?

    [ ] Yes [ ] No [ ] N/A [ ] Don't know

    Comments:

14. Are measurements used to determine the status of defect prevention activities (e.g. the time and cost of identifying and correcting defects and the number of actions items proposed, open and completed)?

    [ ] Yes [ ] No [ ] N/A [ ] Don't know

    Comments:

15 Are measurements made to determine the status of the activities for software process improvement (e.g. the effect of implementing each process improvement compared to its defined goals) ?

    [ ] Yes [ ] No [ ] N/A [ ] Don't know

    Comments:

16. Does your organization have an active metrics program ?

    [ ] Yes [ ] No [ ] Disbanded Program

    If you answered no, please go to question #24. Otherwise, answer all the following questions including question #24.

17. As a percentage of total hours for software development, how many man-hours are spent on metrics related activities annually ?

    [ ] None [ ] 0-1 % [ ] 1-5%[ ] 5% or more

    Comments:

18. Do you feel that the percentage in question #17 is adequate for your organization ?

    [ ] Yes [ ] No [ ] N/A [ ] Don't know

    Comments:

19. In determining how many resources to devote towards metrics collection and analysis, how do you attempt to justify it ? (e.g. is there a formal cost-benefit analysis)

    Comments:

20. What type of product metrics (i.e. size and complexity) do you collect?

    Comments:

21. What type of process metrics (i.e. phase completion and cost) do you collect ?

    Comments:

22. What type of quality metrics (i.e. number of bugs and average days open for a bug) do you collect ?

    Comments:

23. Attempt to quantify positive or negative aspects of these programs. (i.e. the metrics program resulted in a net savings of 1% of the software development budget or the program was shelved after three years because the gains that were toted did not materialize).

    Comments:

24. What are the barriers to introducing or expanding the metrics programs at your organization (e.g. uncertain as to its merits, lack of resources, lack of senior management support) ?

    Comments:

Successful Strategies for Implementing Software Metrics at Intel

ABSTRACT

At the end of 1996, an initiative was started to propagate software metrics across Intel's software development community. A strategy and process was used in assessing the current needs, defining the metrics, getting buy-in, and deploying pilot projects. The experience with the pilot projects provided both technical and cultural learning, some of which were very surprising. This paper will describe the strategy and process used and the results of this initiative.

BIOGRAPHIES

Jeanne Yuen-Hum

Jeanne Yuen-Hum has been involved in software quality since "prehistoric times" (before Windows). She has developed software for real-time embedded systems, managed a Software Quality Assurance department and led a Software Engineering Process Group while working at Ericsson Communications and Canadian Marconi. She joined Intel's Software Corporate Quality Network 2 years ago and is currently working with the Network Infrastructure Operations to improve their product quality. Jeanne received her Bachelor and Master of Science in Computer Science from McGill University in Montreal, Canada.

Paul Dittman

Paul Dittman has been immersed in the quagmire of software development for the past 22 years at Tektronix and Intel. Currently, Paul is part of Intel's Software Corporate Quality Network and is helping the Intel Architecture Labs with improving their software quality -- a most challenging and rewarding job. Paul received his Bachelor of Electrical Engineering and Computer Science from the University of California at Berkeley.

Pat Dix

Pat has developed hardware and software products at Intel for the past 14 years. During that time, she has received both the Intel Achievement Award and the Perfect Design Quality Award and was awarded a patent for Flash memory design. Pat currently is working with the Intel Microcomputer Software Lab to improve the quality of Software Development Kits for future Intel microprocessors. She holds a Bachelor of Electrical Engineering from Cornell University and a Masters of Science from Santa Clara University.

Authors' Address

Intel Corporation
2111 N. E. 25th Avenue
Hillsboro, Oregon 97124-5961

INTRODUCTION

Software Corporate Quality Network (SWCQN) is the software arm of Intel's Corporate Quality Network, which is chartered to

> "Deliver market competitive product quality & reliability while driving breakthrough quality practices for the business."

SWCQN members act as consultants to Intel's engineers and managers and develops software process assets to improve product quality and process.

In late 1996, an Intel software management review committee, led by the SWCQN organization, evaluated a number of proposals for software process improvement initiatives. Given the approximate 2000 software engineers working on product software at Intel in organizations that span a wide spectrum of process maturity and geographical locations, the initiative would need to be universally applicable and scalable. Also, Intel's culture demanded tangible results in a short time frame. Only one proposal was approved, a metrics initiative.

In this paper, we describe the process undertaken to research, scope, design, and validate the metrics, and reveal the learning from our pilot projects. We examine what strategies were successful and share with you the lessons we learned during our journey.

BACKGROUND AND HISTORY

Why metrics? The management review committee realized that there was no consistent means to accurately characterize and measure the software product quality levels across Intel. It was felt that metrics were a way to accomplish this. Metrics would also start to supply hints as to where development problems existed so managers could focus their improvement efforts more reliably.

The goal for the metrics initiative was defined to be "a common set of software product quality metrics that could be shared across groups in order to improve the ability to measure, manage and improve product quality levels". It was assumed that this could be fairly easily accomplished by documenting the metrics and best known methods used across Intel, communicating them amongst the groups and then supporting the education and deployment. It ultimately took about 15 months to demonstrate tangible results and some initial success in meeting our goal.

RESEARCHING CURRENT PRACTICES

Our first step was to determine the current metrics practice at Intel. This was kicked off by the formation of a Joint Engineering Team (JET), led by SWCQN and comprised of key project members from various software development organizations at Intel. Initially the JET was tasked to research and collect the best-known methods (BKMs) for metrics from their organization and then to identify "a common set of software product quality metrics that could be shared across groups".

The results of the research proved disappointing. Most of the metrics collected were post-development (usually after code freeze). The data gathered was often not fully utilized. There was no common repository for the data and as such, much of the data collected could not be shared. There was no common definition of software quality and product quality levels were not consistently characterized. There were very few common metrics found between the groups. We realized that there was a big challenge ahead of us to achieve a common set of software product quality metrics.

The JET members began to educate themselves on metrics. Books and articles were selected, read and discussed by the JET. Several members attended conferences and shared what they had heard. This gave everyone a good knowledge base and motivation for good metrics practices.

From the education and the research of Intel's practices, some basic principles about metrics became fairly obvious. For metrics to be successful they need to be very visible and consistent in their use. They need to be simple. Metrics should only be collected when actions can be taken based on the data. Metrics collection, reporting and analysis all require effort and thus resource must be allocated. There must be management commitment to using the metrics. Metrics is more successful when there is a project goal driving them.

SCOPING THE METRICS

We wanted to treat the metrics like any other product that we would be developing. We started with customer research to understand our customers' needs – the customer being Intel's software community. A survey was created that was taken to a cross-section of managers throughout Intel. In the survey the managers were asked about their quality goals, their need for metrics, and what they saw as barriers to implementing metrics. The surveys were conducted as interviews in which issues could be probed and more information could be acquired.

From the survey results, we identified the top project and quality goals as meeting functionality and meeting schedule. We also learned of the project managers' desire for early prediction of risks on a project.

The survey also revealed a variety of barriers that needed to be overcome for the metrics to be successful. The metrics would take time and resources to collect the data and produce the reports. This would take scarce resources away from revenue generating projects currently underway; a conflict that needed to be dealt with. Lack of defined process was also perceived as a barrier. For example, it is difficult to collect defect metrics if there is no defined defect tracking process or defect tracking tools. The survey pointed out that in many cases management was not asking for the data. Without this demand for the data, the resources and time needed to collect and report the data would eventually be seen as overhead and the effort would be dropped.

In parallel to conducting the survey, we enlisted the help of a metrics expert as a consultant to our initiative. We learned of three distinct uses of metrics: to support decision-making, to perform root-cause analysis for focusing improvements and to collect baseline (historical) data for future comparison and planning.

Given Intel's environment of rapid reorganization due to changing technologies and business needs, and its culture to demonstrate tangible results and success in a short time, the JET decided that our product would be leading indicators of project health that would support decision-making. These indicators would show the most immediate impact and visible benefit within the project development cycle.

DESIGNING THE METRICS

At this point we realized that we could not "design by committee". The JET was composed of 10-12 senior engineers and managers, who did not have the bandwidth for the job ahead of us. We formed a 3-person core team with only SWCQN resources who could dedicate 50-80% of their time to the metrics initiative. The role of the JET was that of reviewers. They would provide input and feedback from the organization's perspective, review and ratify the assets generated by the core team and promote the metrics within their organization.

We set some design goals for the metrics. The metrics would provide a leading indication of project health and product quality by measuring status against plans or targets. The data would be used to support decisions on the readiness of the software for release. The metrics also had to be scalable for projects of different size and process maturity. Using the "Goal-Question-Metric' paradigm, in which the metrics are designed to answer specific questions tied to meeting project goals, the core team designed a set of metrics based on the prioritized list of goals identified by the project managers in the survey.

Once the preliminary design of the core metrics had been completed, we decided we needed a "design review". The objective was to get feedback from beyond the JET, but we also saw it as an opportunity to start our marketing campaign. We set up a "walk-around" – which is very much like a visit to the art gallery. We filled the walls of a large conference with color posters depicting graphs of each metric and a description of its purpose. Food and drinks

were provided and the event was widely publicized. We contacted managers directly to ensure good management participation. The walk-around was conducted over a couple of days to make them available to as many people as possible.

During the walk-around JET members were stationed next to each of the metric posters. The JET members talked to the participants to determine what they liked and didn't like. We also looked for areas of confusion and misunderstanding of the metrics. Visitors were asked to complete feedback forms. We had a great turnout and received very useful feedback (some of which resulted in substantial changes to our metrics design). We also got quite a few leads with projects interested in piloting the metrics.

The documentation of the metrics turned out perhaps even more of a challenge than the design of the metrics. Until we were forced to write out the explanation of each metric simply and unambiguously, we all had different interpretations.

We had two separate objectives with the document we generated. One was a sales pitch, providing motivation for use. The other was the "how to" handbook, providing sufficient guidance to implement the metrics. The document contained a separate section for each metric, describing the motivation for use, required measurements, sample reporting, and underlying processes and definitions need.

Guidance on how to analyze the data and possible corrective actions were described. This included other metrics to look at if a metric showed the project to be behind plan, or what actions a project manager might take if a specific metric showed the project to be behind plan.

The details of the core metrics are beyond the scope of this paper. What is presented here is a brief summary of the resulting core metrics.

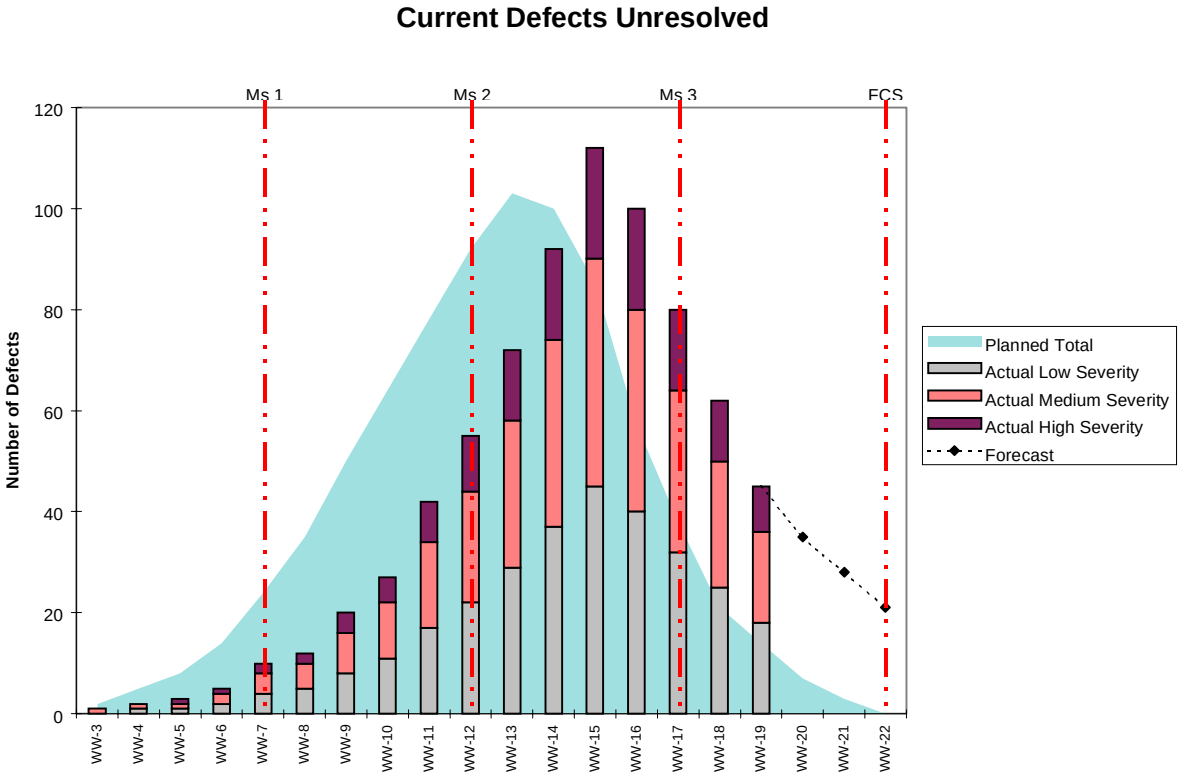
The graphs of each of the core metrics consisted of a shaded curve indicating the plan line with a superimposed bar graph indicating the actual progress. The bar graph may have multiple colors allowing more detailed data to be shown. Each graph has several variations of detail and weighting to allow scaling for projects of various size, complexity and maturity.

Development Progress shows the progress of feature development within the project.

Integration Progress shows the progress of the integration of specific features for specific milestones and quality criteria within the project.

Test Progress shows (in separate plots) the development, execution and passage of test cases in the project.

Defect Measurement is actually two metrics: Cumulative Defects Reported and Defect Resolution. The cumulative defects reported shows the total number of defects reported against time. This metric allows the management to tell if the rate of defect discovery is slowing for a release. The defect resolution metric shows the progress of closing bugs over time. This enables a manager to tell if the defects will be resolved (fixed) in time for a planned release.



Example of Defect Resolution Metrics

Product Performance is a series of metrics that is very project specific. This metric is intended to measure performance such as memory footprint, CPU utilization and frames per second (for video applications).

Requirement Growth is a metric that measures feature creep that has not been incorporated into the project's schedule.

Staffing Profile shows the number of resources on the project and may include effort expended by functions (e.g. development, maintenance, test). This metric is also important if the metrics will be used later for baseline or historical data.

VALIDATING THE METRICS

Early on we recognized the need for pilot projects. One reason was to validate that the core metrics defined would be useful and provide benefits. The other reason was to identify any barriers that were in the way of successful deployment and to learn to overcome them to ensure success. Any failure would result in bad publicity and deter other projects from wanting to implement metrics.

We established strategic criteria for choosing the pilot projects. To increase the chance of success, we selected projects within organizations that SWCQN had an established relationship with and limited the scope to three projects. This would enable SWCQN to provide a dedicated resource for each pilot to drive the metrics implementation including collection, reporting and analysis. It was also important that the pilot project success

would be widely recognized and accepted by Intel's software community. Thus, we only selected projects from organizations where software was a significant part of its business, and ensured that the pilot projects covered a diverse set of project types (new development, a mature product and a 3rd party software integration).

The pilot projects turned out to be challenging. For some projects there was a substantial amount of effort required to understand the project goals and processes such that the metrics could be properly implemented and shown to be adding value. It also turned out that in some cases major process changes had to be made in order to collect the data that was desired (features lists, test plans, change management processes, etc.) or to collect the data at the granularity and frequency that was desired. The right time to engage with the project was also difficult to determine. Engaging too early implied that plans were not in place, while engaging too late would restrict the potential process change required to collect some of the data.

As we were starting engagement with pilot projects in July 1997, we invited a well-published metrics expert to give a seminar at Intel on creating a successful measurement program. This seminar took place about three months after the "walk-around". Again, a lot of effort went into publicizing the event. The seminar drew a lot of interest and was a good motivator for members of the project teams engaged in the metrics pilot.

BENEFITS OBSERVED IN THE PILOT PROJECTS

Software quality is a vague term, and most managers at Intel have difficulty defining software quality, so setting goals and achieving it is almost impossible. Helping them make quality tangible and measurable was a key benefit.

The pilots produced some significant behavioral changes resulting from the introduction of the metrics. Data is far more powerful than words with most Intel engineers and managers. The metrics allowed them to see specific areas of weakness in the project for which there was little debate about the need to change. For example, defect data that exceeded the plan line caused resources to be shifted to bug fixing, features to be cut and milestones to be changed. This type of action was normally taken in past projects when it was too late to still hit the target.

Much of the value of the metrics was not in the data collected, but in the development of the plan lines. The exercise of putting the plan lines together forced planning that might otherwise not have happened. It also brought about important process discussions such as "when is a feature done?", or "what defect trend is acceptable for release?" There was also an improved perception of the need to follow processes. The metrics were viewed as pulling process changes.

At the end of October 1997, three months after initial engagement with the metrics pilot, we conducted two half-day workshops on metrics at the Intel Software Developers Conference. The conference included participants from Intel sites in the U.S. and International. The workshop focused on the motivation for using metrics and the basic principles of metrics do's and don'ts. Project managers and lead engineers from the metrics pilot were invited to attend the workshop to give their testimonials of their pilot experience. By this point, the metrics pilot had already provided some benefits to the projects by improving planning, by identifying process gaps, and by providing the data to support management decision to adjust resources and schedules. This was the starting point to our marketing of the pilot successes. We took our show on the road for the next two months, giving presentations at staff meetings and publicized events across Intel sites.

CULTURAL ISSUES

Once the metrics were collected and reported, interesting behaviors were observed on some of the pilots. These behaviors included "shooting the messenger", laying blame, withholding of information, denial and lack of action.

We quickly learned the importance of buy-in and training of the management and the team. It was crucial that everyone understood how the data would be collected, what actions would be taken as a result, and how this would benefit the team and reduce their pain. Without this understanding, incorrect assumptions were made, and suspicions and lack of trust resulted.

Confidentiality was also an important issue among the projects. If the data was reported beyond the team using them, they were often perceived as a report card. If the data was used to produce metrics that would look negative, the data was often withheld.

It turned out to be important to make sure that the data was reported in a manner that was not embarrassing to members of the team. This was accomplished by showing the results to affected individuals before reporting to management and allowing the individuals to determine action plans that could be communicated with the metrics. This helped give confidence that the project was under control and any required corrective action was in place.

FUTURE DIRECTION

Our long-term strategy is to generate more demand for metrics. The success of the pilot projects resulted in management demand for department-wide metrics to monitor project health and product quality. For each department in which the metrics were piloted, metrics are now being collected on all key projects and being reviewed by management at the department level. The metrics initiative will continue to use these successes as advertisement to start the seed for new pilots in other departments.

CONCLUSION

Metrics can be a very effective tool for process improvement efforts. For it to be successful, thought must be put into the motivational aspects as well as the technical aspects for widespread adoption. Some of the lessons we learned:

“Don’t underestimate the effort. Adequate resource is a must.”

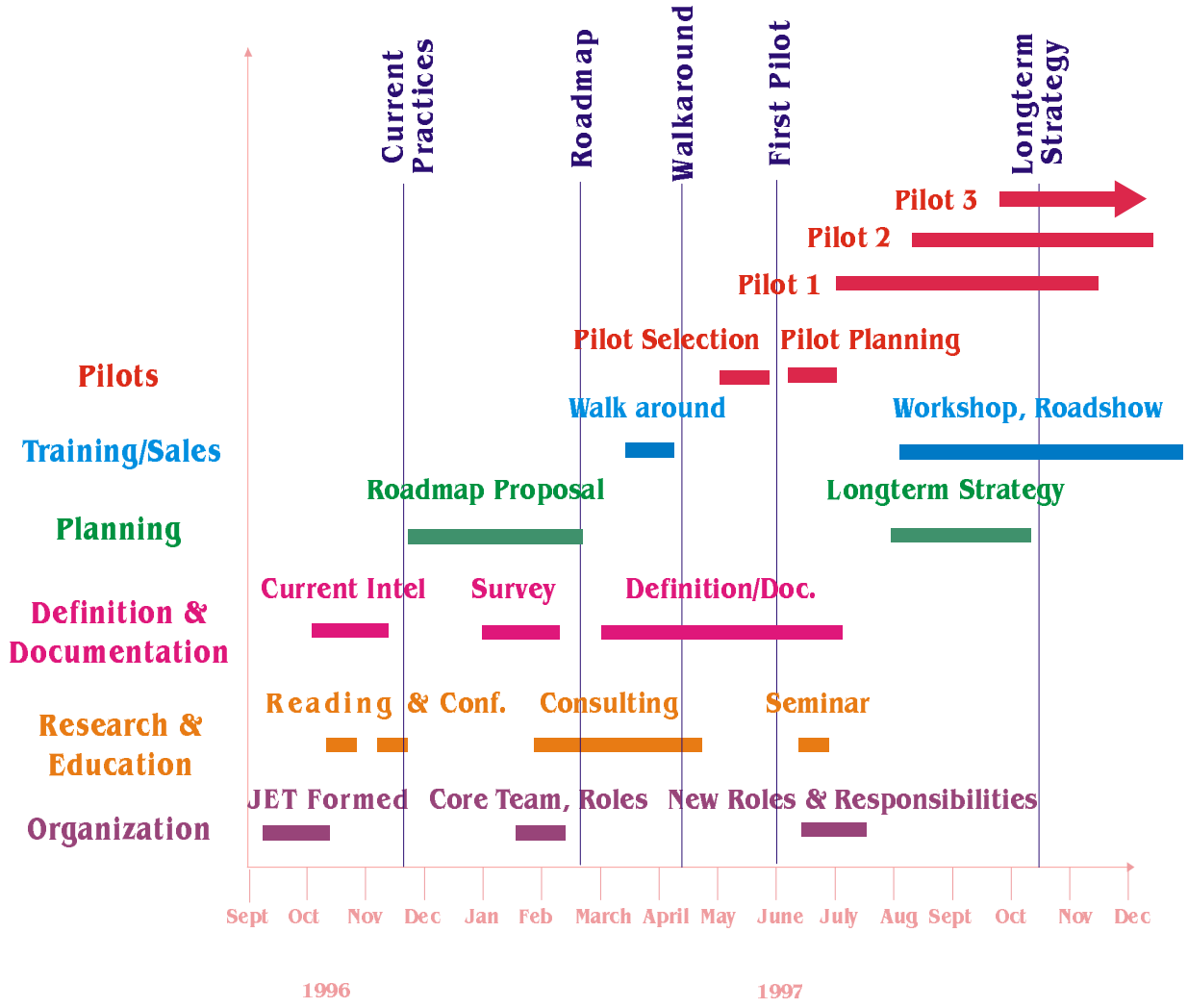
- Don't underestimate the dependency of new quality initiatives to the underlying management cultures, practices, and processes. The resulting change to projects from using metrics was much greater than we expected, and the effort to get it started in the initial pilots was much larger as a result.
- Provide dedicated (SWCQN) resource in the initial pilots. Most project teams cannot dedicate the initial resources or bandwidth needed to get new initiatives established.
- On the other hand, don't do it all for them. Help the organizations by working with them. Make sure they feel the ownership of the data, and can trust it. It also makes it possible to transition out, once the pilot reaches a sustaining mode.

“Be a constant marketer.”

- It's critical to keep key sponsorship strong. Build your allies and work to keep them engaged.
- The best evangelists and advocates to help spread the word are successful project teams. Give them the right forums to make use of this resource. (e.g. the ISDC Workshop)
- Become a constant marketing and communication engine. It's very easy to lose touch and the attention of your audience, since they exist in a completely different world within their individual projects.

“Manage this initiative as you would a project.”

- Set goals, follow a roadmap and track to your plan.
- Spend time on your long-term strategy so you don't forget the forest for the trees.
- Force yourself to document things. The value is not in the document, but in forcing you to think about how to explain things simply and unambiguously. If you can't explain it simply, no one will understand it in the same way, especially if it's verbal!
- Get a strong, dedicated leader with adequate bandwidth. Tons of time was spent gluing the efforts together, poking the team into action, and greasing the skids behind the scenes. We couldn't have done it without this role!
- Build a small good team, keep pushing yourselves, and have lots of fun! I think we showed that we could be effective and also enjoy the project at the same time!



Overall Metrics Timeline

The metrics effort at Intel, still gaining momentum and growing, has shown significant behavioral changes within the projects using them. The metrics are an effective technique for starting and ensuring lasting and productive process changes in an organization.

View Slides

DEFINING TEST POLICIES and STANDARDS

A Must For Climbing the CMM

A Must For Y2K Planning

PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

and

INTERNATIONAL CONFERENCE ON SOFTWARE QUALITY

Cheryl Y. Moore
Program Management Advisor
Federal Express Corporation

TABLE of CONTENTS

I. ABSTRACT

Most test efforts span multiple organizations, groups and test teams. The major objective of standardizing test processes is to define policies and procedures to be supported by ***all*** test participants during the testing phase. This paper outlines "best practices" for accomplishing quality testing across multiple departments involved in the testing of mainframe and distributed applications. Regardless of the level of testing required (system, integration, acceptance), policies defined in this paper can assist with addressing the following test issues:

- Test Planning, Coordination and Management
- Test Execution
- Identification of Continuous Test Improvements
- Development of Test Measurements/Metrics

Outlined within this paper are *recommended* methods for:

⇒ Effective communications during the test planning and execution phases;
⇒ Defining a test structure to standardize roles and responsibilities for test accountability;
⇒ Defining deliverables for documenting test plans, schedules, cases, application checklists;
⇒ Establishing formal meetings and reviews for developing test plans, reviewing statuses, analyzing test defects and fixes, implementation planning and coverage;
⇒ Establishing methods for identification of continuous test improvements.

II. INTRODUCTION

With the onset of critical corporate initiatives and projects such as YR2000, well defined testing policies are key for ensuring software quality. For many IT organizations, software testing phases still tend to be very time consuming, labor intensive efforts. In order to progress up the *Capability Maturity Model (CMM)* for software development and ultimately increase customer satisfaction, many corporations are aggressively implementing and improving test policies and standards. By interviewing Developers, Test Managers and other test participants, and identifying "best practices", standardized processes can be developed and implemented to successfully test software components across various platforms and groups. Initiating a Software Process Improvement (SPI) Program can facilitate continuous improvements to defined test processes. In an effort to share a well defined test process with developers and testers of other companies, these test policies and standards are being presented at the annual Pacific Northwest Software Quality Conference and International Conference on Software Quality.

To ensure successful end-to-end testing, a dedicated group must be empowered to set the direction for recommending and defining standards, processes, methods and tools for the Systems Development Life Cycle. The designated group should create and maintain a ***"Developer's Roadmap"*** of guidelines for developing and implementing quality products. By utilizing corporate websites, developers and testers can navigate through procedures for planning, executing and improving test activities. For most corporations, these responsibilities usually reside in a Software Engineering Process Group (SEPG); a Quality Assurance Group (QAG); or a selected Systems Development Group (SDG). "Best Practices" for planning, coordinating and facilitating end-to-end testing across multiple divisions, groups and platforms are the basis for this paper.

III. TEST PLANNING and COORDINATION

Test Planning is the process of gathering, organizing, documenting and communicating information relevant (and often critical) to the Test Execution Phase. For most companies, software components (mainframe and distributed) are packaged together for a scheduled implementation or release. It is recommended that packaged software be tested and implemented in accordance with an established ***Load Process*** governed by a set of milestones. Milestones dates are negotiated and supported by all areas impacted by loading components. A ***Load Exception Process*** should be developed and implemented to handle software components that, for various reasons, must load outside the published schedule.

Software testing policies should encourage the initiation of *Test Planning Sessions*. These sessions should start several weeks (during the Requirements Definition Phase), and in some cases, several months, prior to the start of the Test Execution Phase. The complexity of loading projects and software components, along with the level of testing required, determines exactly when test planning begins. To facilitate test planning activities, weekly meetings and/or working sessions are held to discuss, coordinate and often negotiate activities, schedules and other issues critical to the Test Execution Phase.

Test roles and responsibilities must be assigned and communicated at the onset of the Test Planning Phase. To ensure accountability for test activities across multiple divisions, the test structure depicted below is recommended.

Software Testing Process Structure

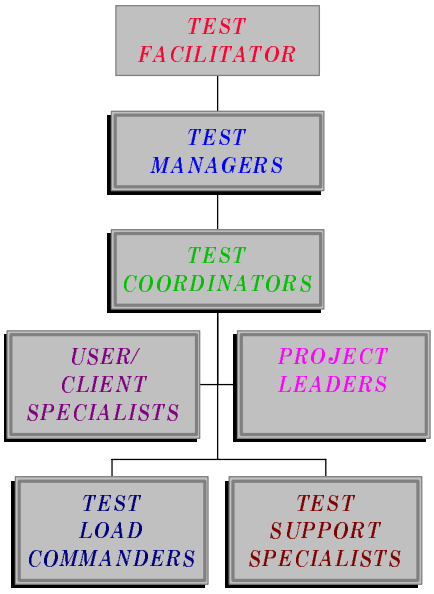


Figure 1

Testing Process Structure:

A ***Test Facilitator*** is appointed from the SEPG or QAG to plan, coordinate, document and facilitate end-to-end testing across multiple divisional groups and application areas. The ***Test Facilitator*** deploys program management techniques to plan and coordinate test activities, align schedules from multiple groups, and facilitate test progress and status.

Test Managers are appointed (at the Managing Director level) for each divisional area impacted by and participating in the Test Execution Phase. ***Test Managers*** provide leadership, support and control during the end-to-end test and ensure that all testing policies are adhered to.

Test Coordinators are assigned by ***Test Managers*** to coordinate, plan and document activities to thoroughly test specific applications. ***Test Coordinators*** create Application Test Plans detailing all tasks and schedules required to support the end-to-end test.

Project Leaders support the Testing Process by providing ***Test Coordinators*** with information related to loading projects and software components for specific applications.

User / Client Specialists ensure that business requirements are thoroughly tested by developing and documenting detailed test scripts and cases; and validating expected results during the Test Execution Phase.

Load Commanders ensure that loading software components are migrated to the appropriate test environment(s) according to published schedules. ***Load Commanders*** also ensure that defective software is removed and enhanced software is returned to the appropriate test environment(s) and/or libraries.

Test Support Specialists are responsible for executing and monitoring planned tests in accordance to published schedules.

The Test Planning Phase should conclude with the following objectives being accomplished:

- Identify areas/groups impacted by loading projects and components;
- Identify test participants and assign roles and responsibilities;
- Identify loading projects and software components;
- Determine test scope and objectives;
- Coordinate test activities and align schedules for each level of required testing;
- Finalize and coordinate test scripts and cases;
- Conduct risk assessments and develop contingency plans;
- Status check developmental progress of loading components;
- Resolve all open issues for the Test Execution Phase;
- Create and publish the Master Test Plan.

Outputs/Deliverables for the Test Planning Phase

Test Coordination Planning Document:
A Test Coordination Planning Document summarized results of the Test Planning Phase and is published to all areas impacted by loading components and projects. In conjunction with Master and Application Test Plans, the Test Coordination Planning Document can be used as a reference document during the Test Execution Phase and also during future Test Planning Phases.

Sections *recommended* for ***Test Coordination Planning Documents*** include:

1.0 ***Introduction***
A brief description about the purpose of the Test Coordination Planning Document.

2.0 ***Scope of Planning Phase***
A brief narrative description explaining the scope of test planning activities.

3.0 ***Test Objectives***
Document planned objectives for the Test Execution Phase.

4.0 ***Critical Success Factors***
List of critical success factors for the Test Execution Phase.

5.0 ***Test Participants***
Identification of all test participants and associated roles for the Test Execution Phase.

6.0 ***Work Breakdown Structure***
Identification of all tasks/activities required to coordinate and prepare the for the Test Execution Phase.

7.0 ***Test Milestones and Associated Dates***
List of significant events and planned dates for the Test Execution Phase.

8.0 ***Open Issues and Assumptions***
List of all *unresolved* issues and assumptions that may impact the Test Execution Phase.

9.0 ***Projects Scheduled for Implementation***
List of all projects, along with project description and overview, schedules to be tested during the Test Execution Phase.

10.0 ***Applications and Associated Interfaces***
List of applications participating in the Test Execution Phase along with associated interfaces required to support end-to-end testing.

11.0 ***Primary Contacts***
List of all test participants and/or primary application contacts.

12.0 ***Meeting Diary***
A repository of Test Planning Meeting Minutes and attendance records.

Application Test Plans:

Application Test Plans are created to document detailed tasks and associated schedules required to thoroughly test applications participating in the Test Execution Phase. Application Test Plans are used to coordinate test activities and align schedules across multiple divisions and various platforms. Application Test Plans are major contributors for creation of the Master Test Plan.

Sections *recommended* for ***Application Test Plans*** include:

1.0 Introduction
A brief description about the application being tested, along with the scope and objectives of testing the application.

2.0 Testing Items
Identification of the projects, initiatives, work requests or components being tested by this application.

3.0 Features To Be Tested
Identification of software features and components being tested.

4.0 Features Not To Be Tested
Identification of software features and components excluded from the test.

5.0 Test Approach
A narrative description of the approach for testing application features and functions.

6.0 Suspension and Resumption Criteria
Specify criteria for suspending, aborting and re-starting application tests.

7.0 Testing Tasks
Identify all tasks/activities (including task dependencies) required to perform application testing.

8.0 Environmental Requirements
Specify environmental components (i.e. test libraries, environmental security features, test tools) required to perform application testing.

9.0 Test Responsibilities
Identify assigned resources and responsibilities to support test activities.

10.0 Pass/Fail Criteria
Specify pass/fail criteria for each iteration/cycle of the test.

11.0 Test Certification Criteria
Specify criteria for ensuring components are ready for the next level of testing or the production environment.

12.0 Application Turn-Over or Acceptance Criteria
Specify the exact criteria and/or conditions (or handshake) that must exist between interfacing applications.

13.0 Risks and Contingencies
Identify high risk factors or assumptions and specific contingency plans for minimizing risks.

14.0 Application Test Schedule
Include schedule of all activities planned to thoroughly test application and software components.

Master Test Plan:

A Master Test Plan is developed by the SEPG or QAG to summarize, document and communicate activities and schedules to support end-to-end testing of projects and components to be implemented for the scheduled load. The Master Plan governs the entire Test Execution Phase and should be supported by all areas participating in the test or impacted by loading projects and components.

Sections *recommended* for the ***Master Test Plan*** include:

1.0 Introduction
Narrative about the purpose of the Master Test Plan.

2.0 Test Scope
List of applications participating in the Test Execution Phase, along with descriptions of loading projects and components.

3.0 Critical Success Factors
List all factors that will have a major impact for ensuring the success of the Test Execution Phase.

4.0 Projected Milestones and Dates
List all significant events and planned target dates.

5.0 Testing Environmental/Techniques
A detailed description of the environment(s) required to test components, along with tools and techniques to be utilized.

6.0 Test Tasks
Identify coordinated tasks/activities required to support each level of testing.

7.0 Test Schedules
Document the overall aligned test schedule to be supported by all application areas and test teams participating in the Test Execution Phase.

8.0 Problem Reporting
Explain how problem logs (defects) will be detected, reported, tracked, escalated and resolved during the Test Execution Phase.

9.0 Issues and Concerns
Document any questions, issues, concerns or problems that may impact or hinder test efforts or activities.

10.0 Approvals
Obtain Management approval (at the Managing Director level) for supporting activities and schedules documented in the Master Test Plan.

IV. TEST EXECUTION

The results of successful test planning leads to smooth execution of test activities. Test Execution can be defined as an orderly and timely progression through selected applications and associated interfaces to determine if specific requirements are satisfied. Objectives to be accomplished during Test Execution Phases include:

1. Discovering errors and defects with loading software and/or related components;
2. Gaining knowledge about capabilities of applications;
3. Understanding the performance limits for tested applications/systems;
4. Gaining confidence that applications/systems perform with acceptable risks;
5. Discovering and documenting valuable information about the ***quality*** of loading projects and software.

It is recommended that the Test Execution Phase be initiated with a Test Kick-Off Meeting. The Kick-Off Meeting is chaired by the Test Facilitator and is attended by all test participants. The Master Test Plan is distributed in advance so that participants can be prepared to discuss the Plan during the meeting. The purpose of the meeting is to ensure that:

- Test roles and responsibilities have been assigned and widely communicated;
- Proper communication channels and guidelines are in place and understood;
- Preliminary testing tasks have been completed;
- Loading software is available and ready for execution;
- Test environments are controlled and stable;
- The Master Plan has been thoroughly reviewed and approved;
- Test activities and schedules can be supported;
- All open issues are resolved;
- Test participants are committed to a quality test effort.

Dependent upon the level of testing required, the Test Execution Phase may span several weeks or months. To facilitate an orderly progression through applications outlined in the Master Plan, a series of standardized meetings are held to monitor test and load status. Each meeting is scheduled with a defined purpose and agenda. Standardized meetings are discussed in Figure 2.

Standardized Meetings To Support Software Testing Processes

Standard Meeting:	***Purpose:***
Load Planning Meeting	Discuss status of loading projects at the divisional level;
	Discuss plans and schedules for future loads.
Daily Test Meeting	Discuss overall test progression, status and schedules;
	Communicate environmental issues;
	Discuss global test problems and defects.
Test Case/Plog Review Meeting	Analyze and discuss test case results;
	Document defects and discuss fixes;
	Review and prioritize "Open" Plogs;
	Ensure validated Plogs are "Closed".
Go/No-Go Meeting	Assess load risk according to outstanding Plogs;
	Discuss strategies to minimize risks;
	Determine and communicate if Load is to proceed (Go).
Implementation Planning Meeting	Discuss plans for migrating components to production;
	Coordinate outage and start-up schedules;
	Provide contacts for Load support.
Load Check-Out Meeting	Communicate status of loaded components;
	Ensure loaded components reside in proper libraries;
	Verify application start up schedules.

Figure 2

Many organizations struggle with whether testers should be within the development groups or an independent organization. Some test experts argue that better test results can be accomplished by allowing objective testers (and not developers) to plan and execute the tests of loading software. Regardless of this decision, organizations must standardize policies to ensure that quality testing can be successfully planned and executed.

Maintaining robust testing tools can prevent the testing phase from being a labor intensive, time consuming effort. Selection of the appropriate testing tools can reduce test cycle time and add quality to verification of overall results. However, serious consideration should be given to ***when*** and ***how*** to automate a test. Measurable reduction of test costs and time; along with an increased number of *additional* defects should be the return on investing in test automation. Using a combination of manual and automated testing is usually the most effective method of ensuring quality testing.

V. IDENTIFICATION OF CONTINUOUS TEST IMPROVEMENTS

To be improved, test processes must first be defined, customized and managed. Once processes are defined, "Process Roadshows" can be conducted (usually by the SEPG or QAG) to communicate defined processes and obtain buy-in for support. "State of the Process" presentations should be conducted to re-enforce test improvement activities. After software components are successfully tested and implemented, test processes and activities should be evaluated during a series of ***Post Testing Reviews***. ***The Post Testing Review Summary*** is used to document problems that must be proactively addressed and corrected during future testing phases. The ***Post Testing Review Summary*** can also be used to develop and document *Process Quality Indicators (PQIs)* for improving test policies, activities, deliverables and schedules.

Sections *recommended* for the **Post Testing Review Summary** include:

1.0 Evaluation and Recommendations for Improving Test Planning Activities and Deliverables
2.0 Evaluation and Recommendations for Improving the Test Process Structure
3.0 Evaluation and Recommendations for Improving Test Execution Activities, Schedules and Deliverables
4.0 Evaluation and Recommendations for Improving Standardized Meetings
5.0 Evaluation and Recommendations for Improving Implementation Planning Activities
6.0 Evaluation and Recommendations for Improving Overall Quality of Testing

Metrics should be collected during the Test Planing and Execution Phases to identify improvement opportunities. A statistical ***Scorecard*** can be produced by the SEPG/QAG to report the following metrics:

⇒ Milestone Compliance
⇒ Meeting Participation
⇒ Approved Load Exceptions
⇒ Summary of Plogs
⇒ Number of Participating Managing Directors
⇒ Percent of Reloaded Modules

Metrics related to specific test activities and included below are also collected and reported to upper management in the ***Executive Test Summary Report***:

⇒ Number of Planned Projects, Work Requests Schedule To Load
⇒ Number of Actual Projects, Work Requests Loaded
⇒ Explanation of Variance

⇒ Number of Planned Components Scheduled To Load
⇒ Number of Actual Components Loaded
⇒ Explanation of Variance

⇒ Planned/Scheduled Load Date
⇒ Actual Load Date
⇒ Explanation of Variance

⇒ Planned Test Execution Start Date
⇒ Actual Test Execution Start Date
⇒ Planned Test Execution Finish Date
⇒ Actual Test Execution Finish Date
⇒ Explanation of Variance

⇒ Number of Test Cycles Planned
⇒ Number of Test Cycles Executed
⇒ Explanation of Variance

⇒ Planned Duration of Test Execution Phase
⇒ Actual Duration of Test Execution Phase
⇒ Explanation of Variance

⇒ Number of Plogs Opened
⇒ Number of Plogs Closed
⇒ Explanation of Unresolved Plogs

The ***Test Process Model (TPM)***, depicted in Figure 3, can also be used to identify and implement test improvement initiatives. The TPM reinforces the CMM by encouraging use of the same levels to mature and optimize test processes.

The Test Process Model

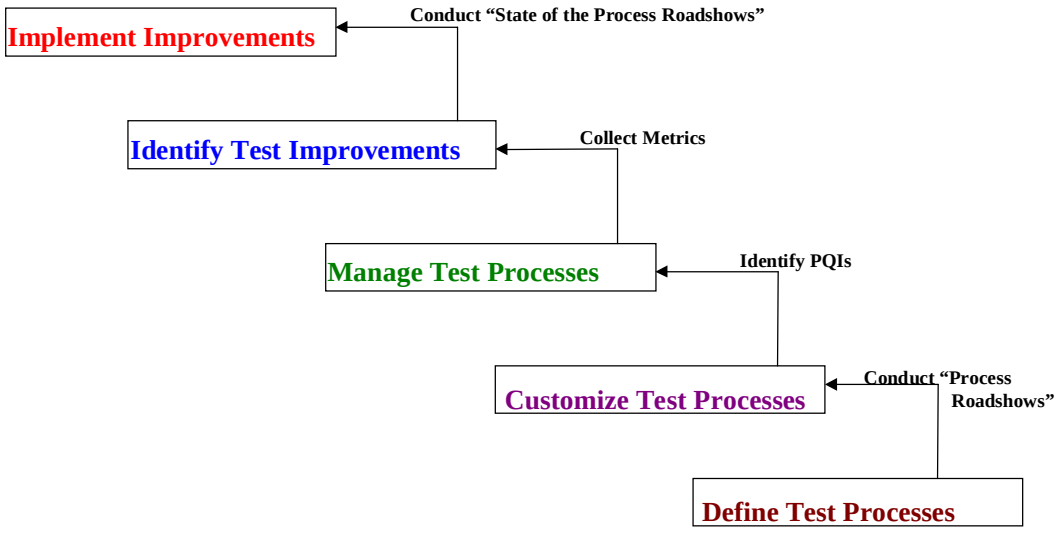


Figure 3

VI. CONCLUSION

Successful testing requires discipline to keep doing the same tasks over and over and over again. By defining standardized testing policies and allocating sufficient time and resources to Test Planning activities, quality results can be accomplished for Test Execution Phases. Test policies may be difficult to define, but the benefits are well worth the investment. Remember what the experts tell us about discovering software defects. "The earlier the detection, the cheaper the fix….., the later the detection, the more expensive!"

Defining Test Process Models and sticking to them will move your organization up the Capability Maturity Model and ensure delivery of quality products and services to your customers. As Software Engineers, we have settled into the "faster, cheaper, better" era. Let's go one step further to "smarter". And when "assigned" to support the testing phase, maybe we won't be included to offer up our first born instead. ***Good Luck!***

For more information regarding processes discussed in this paper, please contact:

Cheryl Y. Moore, Program Management Advisor
FedEx Corporation
Strategic Revenue Solutions
2600 Nonconnah Blvd.
Memphis, Tennessee 38194-1909
(901) 922-1015
Email Id: ***cymoore@fedex.com***

REFERENCES

1. Burnstein, I., T. Suwannasart, and C. R. Carlson, "Developing a Testing Maturity Model: Part 1", *Crosstalk*, STSC, Hill Air Force Base, Utah, August, 1996.

2. Hetzel, B., "The Complete Guide to Software Testing", 2d ed., QED Information Sciences, Inc., Wellesley, Massachusetts, 1990.

3. Moore, C., "FedEx Software Testing Process Document", FedEx Corporation, Memphis, Tennessee, 1996.

4. Moore, C., "FedEx Quality Assurance Handbook", FedEx Corporation, Memphis, Tennessee, 1992.

5. Thayer, R., "Software Engineering Project Management, A Top Down View", IEEE Tutorial, Software Engineering Project Management, R. Thayer, ed., IEEE Computer Society Press, Los Alamitos, California, 1990.

Towards Quality Programming in the Automated Testing of Client/Server Applications

Huey–Der Chu

Graduate School of Resource Management
National Defense Management College
Chungho, Taiwan 23500
Tel: (886) 2 2225 5830
Fax: (886) 2 2222 4496
Email: jchu@casq.org
URL: http://www.casq.org/jchu

John Dobson

Centre for Software Reliability,
Department of Computing Science
University of Newcastle upon Tyne, NE1 7RU, U.K.
Tel: (44) 191 222 8228
Fax: (44) 191 222 8788
Email: john.dobson@ncl.ac.uk
URL: http://www.cs.ncl.ac.uk/~john.dobson

Abstract

This paper presents a statistics–based framework which is an extension of the testing concept in Quality Programming for automating client/server testing. We specify all possible delivered messages between events by means of a 'Symbolic Message Attribute Decomposition' (SMAD) tree with the casual message ordering. To address two crucial issues in software testing, when to stop testing and how good the software is after the testing, this framework not only can automatically generate the test scripts and valid the test results on client sites with the SMAD tree, but can examine the interaction behaviour between client and server sites with the causal message ordering. The implementation of automated testing for a 3–tier client/server banking application which incorporates the framework is also described.

Biographical Sketch

Huey–Der Chu is currently a PhD student in the Centre for Software Reliability at the University of Newcastle upon Tyne funded by the National Science Council in Taiwan. He will teach some courses at National Defense Management College in Taiwan if he finishes his work in Newcastle this summer. His current research interests focus on building an integrated environment for testing distributed applications and applying mobile agents with Java RMI to software testing.

John Dobson is the Professor of Information Management at the University of Newcastle upon Tyne. His current research interests are in investigating methods of analysing systems for possible breaches of safety and security policy in a way that can be applied equality well to computer systems or organisational systems or mixture of both; and in the process of handling changing requirement.

Towards Quality Programming in the Automated Testing of Client/Server Applications

Huey–Der Chu

Graduate School of Resource Management
National Defense Management College
Chungho, Taiwan 23500

John Dobson

Centre for Software Reliability,
Department of Computing Science
University of Newcastle upon Tyne, NE1 7RU, U.K.

ABSTRACT

This paper presents a statistics–based framework which is an extension of the testing concept in Quality Programming for automating client/server testing. We specify all possible delivered messages between events by means of a 'Symbolic Message Attribute Decomposition' (SMAD) tree with the casual message ordering. To address two crucial issues in software testing, when to stop testing and how good the software is after the testing, this framework not only can automatically generate the test scripts and valid the test results on client sites with the SMAD tree, but can examine the interaction behaviour between client and server sites with the causal message ordering. The implementation of automated testing for a 3–tier client/server banking application which incorporates the framework is also described.

Keywords: Quality Programming, Client/server applications, Automated Software Testing, Causality Relation

1 Introduction

In today's competitive market, the production of high–quality software systems is an important issue. Software quality is the degree to which a customer or user perceives the software as meeting his or her composite expectations (Deutsch & Willis, 1988). To achieve high quality software, it is essential to sufficiently test the software before delivery. This testing is for discovering product defects and useful as an independent assessment of software execution in an operating environment. Testing is a very time–consuming and tedious activity and accounts for over 25% of the cost of software development (Beizer, 1990; Myers, 1993; Norman, 1993). Manual test efforts tend to find the majority of defects at the end of the release effort or during beta testing, where the errors are more expensive to fix. In addition to its high cost, manual testing is unpopular and often inconsistently executed. If the testing process could be automated, the cost of developing software could be significantly reduced (Ince, 1987).

Client/server applications are traditional applications re–cast for a new environment of multiple interlinked computers and have been designed as systems whose data and processing capabilities reside on multiple platforms, each performing an assigned function within a known and controlled framework contained in the enterprise. Applications can now be broken into pieces that are common to more than one application and therefore stored, maintained and executed in a central location (a server), with the results sent back to the requesting client. The complexity of the client/server makes testing more difficult and poses new challenges to the development organization.

Most automated testing tools are an elaboration and more modern implementation of the capture/playback paradigm. There are indeed many tools that allow test scripts to be recorded and then played back, using screen captures for verification. There are some inherent problems with these capture/playback testing tools (Zallar, 1997; Pettichord, 1997): Firstly, test automation is only applied at the final stage of testing when it is most expensive to go back and correct the problem. Secondly, the testers do not get an opportunity to create test scripts until the

application is finished and thirdly, modifications are made to the application, invalidating the screen captures, making playback fail.

Moreover, for client/server applications, there are some limitations with the capture/playback paradigm (Mooney & Chadwick, 1998; Quinn & Sitaram, 1996): Firstly, the communication mechanism between clients and servers uses technology like an RPC protocol that current capture/playback tools cannot effectively capture. Secondly, these client testing products may not provide a way to test the effect of multiple users of the software and thirdly, there are non–deterministic behaviours in a client/server application. As a result of indeterminacy, repeated execution of a client/server application with the same test script may execute different paths and produce different results. Therefore, some mechanisms are required in order to exercise these test scripts and examine the test ordering.

To address these problems, a statistics–based framework which is an extension of the testing concept in Quality Programming (Cho, 1988) is presented for automating client/server testing. In this framework, we specify all possible delivered messages between events by means of a 'Symbolic Message Attribute Decomposition' (SMAD) tree. It combines classification and syntactic structure to specify all possible delivered messages in a client/server application. In the upper level of the SMAD tree, all delivered messages are classified into three types of messages: input message, intermediate message and output message. Each type of message has a syntactic subtree describing the characteristics of messages with a 'happened before' relationship so that it can be determined whether messages were delivered in an order consistent with the potential causal dependencies between messages. With the SMAD tree, this framework not only can automatically generate the test script and validate the test results on client sites, but can examine the test ordering on the server site, with respect to the causal message ordering under repeated executions.

In Section 2 of this paper, an overview of Quality Programming is presented, as established by our previous work. A statistics–based framework which is an extension of the testing concept in Quality Programming for distributed applications is presented in Section 3. In Section 4, a 3–tier client/server application, a banking application written using Java Remote Method Invocation (RMI) and Java DataBase Connectivity (JDBC), is described in some detail and explains the process of fitting it into the framework. How the test of the application is conducted and comment on the results obtained are discussed in Section 5. Section 6 summarizes our research.

2 Quality Programming

Current statistical testing techniques involve exercising a piece of software by supplying it with test data that are randomly drawn according to a single, unconditional probability distribution on the software's input domain (Curritt *et al.*, 1988; Dyer, 1992; Thévenod–Fosse *et al.*, 1995). This distribution represents the best estimate of the operational frequency for the use for each input. The main benefit of statistical testing is that it allows the use of statistical inference to compute probabilistic aspects of the results of the testing process, such as reliability, mean time to failure (MTTF) and mean time between failures (MTBF). However, these techniques are insufficient for many types of software, because the probability of applying an input can change when the software is executed (Deck, 1996; Whittaker and Tomason, 1994). As a result of this difficulty, we introduce the inverse concept, Quality Programming, presented by Cho (1988).

2.1 The Concept of Quality Programming

In Quality Programming, each execution of the software is considered equivalent to 'sampling' an output from the output population. The goal of software testing is to find certain characteristics of the population such as the ratio of the number of defective outputs in the population to the total number of outputs in the population. It uses the number of executions of the software to assess the software reliability, which is different from previously mentioned measures which use the execution time. Gathering the data for the error history of a piece of software requires a long period of time, and even then, the reliability measure is often difficult to quantify. However, a piece of software is not subject to deterioration such as wear, tear or burn, that is, the reliability of a piece of software is independent of time but dependent on the frequency and nature of software usage. Cho (1988) gives the following definition:

Software reliability is 1 – θ, the probability that the software performs successfully, according to software requirements, independent of time,

where θ is the defective rate of the software output population. This definition is a natural consequence of following the principles of software engineering with statistical quality control. From this point of view, determining the defective or non–defective outputs from software requires corresponding input data. The input domain is the source from which input data are constructed for the software. If the input domain is not well defined, the input data will not be properly constructed and will be of poor quality. Following Cho, the approach specifies the input domain of a software by means of a 'Symbolic Input Attribute Decomposition' (SIAD) tree which is a syntactic structure representing the input domain of a piece of software in a form that facilitates construction of random test data for producing random output for quality inspection. The SIAD tree enforces the development of well–defined user requirements and imposes discipline in both design and implementation. From a fault forecasting point of view, a comparative analysis (Thévenod–Fosse and Waeselynck, 1991) concluded that the best evaluation is provided by Cho's approach, particularly when few failures are observed during a test experiment.

2.2 The process of Quality Programming

In quality programming as introduced by Cho, a statistical approach to control the quality of software is used. When a software is viewed as a factory, processing of input units into product units becomes conceptually equivalent to taking random product units from the software's population. If the product units in the population are all of a good quality, then the units taken will be of a good quality. A simplified view of the process of quality programming is shown in Figure 1.

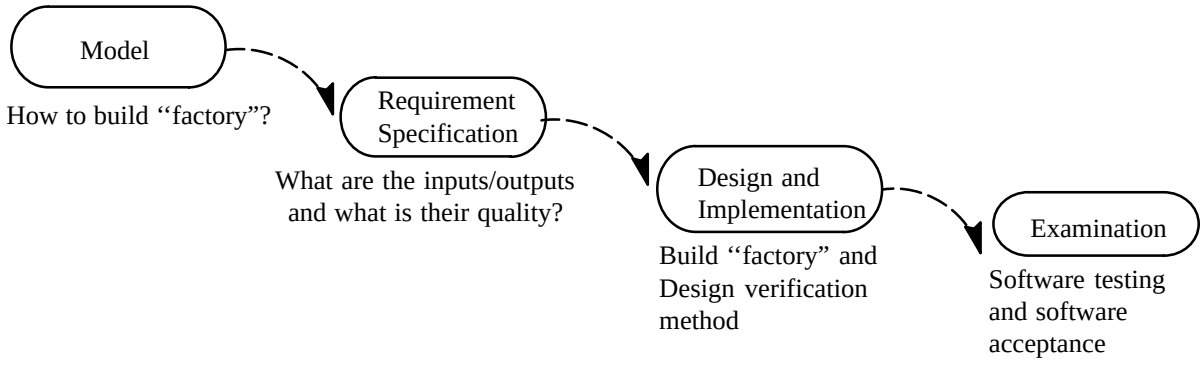


Figure 1: The process of Quality Programming

As shown, the process is divided into the following stages (Chu, Dobson & Liu, 1997): model, requirements specification, design and implementation and examination. This process ensures that quality is built into software from the beginning of its development.

- Model: given a system to be developed, a model is developed to analyse and understand the problem. Models including a description of the problem and product to be generated by the software are built to form the basis of product design and the concept of the software being developed.

- Requirements specification: requirements are then generated as a result of the modeling activity. Included in the requirements are software and test requirements. Software requirements define the functions the software is to perform and the quality characteristics such as response time, throughput, understandability and portability. Test requirements define the product units and product unit defectiveness for statistical sampling, sampling methods for estimating the defective rate of the software population with which to judge software quality, statistical inference methods and the confidence levels of software output population quality, the acceptable software defect rate and the generation methods of test input units.

- Design and implementation: with well–defined requirements, software development can be divided into two channels which can proceed concurrently: software design and implementation and software test design and implementation. Top–Down programming and critical–module–first implementation methods are used in the software channel. The formulation of sampling plans are used in the test channel. During the design and implementation phases, interfaces between the channels are incorporated to ensure that quality is built into the software at every stage of development.

- Examination: software testing is performed again on a most–critical–module–first basis to ensure that the software is integrated on a secure–quality–part basis. If the software passes the test requirements delivery to the user takes place. The user employs quality control tools to determine the acceptability of the software output population, and this becomes the basis for accepting or rejecting the software.

3 A Framework for Automated Testing Distributed Applications

In Quality Programming as introduced by Cho, the generation of test data can be automatically achieved based on the SIAD tree, however, it lacks a clear framework which would indicate how automated testing is to be be achieved. In this section, we develop a framework for the automated testing of client/server applications. the concept of the SIAD tree will be extended to a new construct which we term the SMAD tree making it a more powerful technique for test data generation and test result inspection in client/server applications. The framework is shown in Figure 2.

3.1 Modelling Distributed Software

A distributed computation describes the execution of distributed software by a collection of processes. The activity of each sequential process is modelled as executing a sequence of events. An event may be internal to a process and cause only a local state change, or it may involve communication with another process. A graph model is used to define both sequential and distributed software as follows:

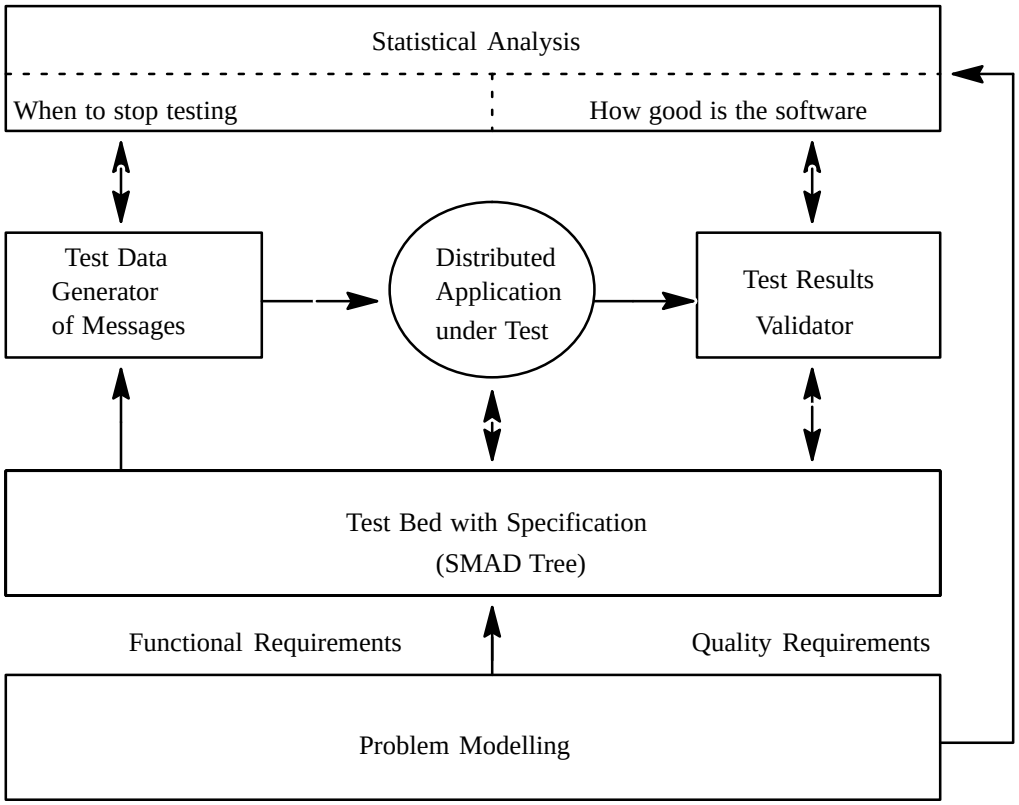


Figure 2: A statistics–based framework for testing distributed applications

Definition 1: A *message flow graph (MFG) of a software S* is a directed graph G = (E,M,s,r, I, O), where
(1) E is a set of events,
(2) M is a set of messages. It is a binary relation on E (a subset of E $\times$ E), referred to as a set of directed edges,
(3) s and r are, respectively, unique send–message and unique receive–message events, s,r $\in$ E,
(4) I and O are, respectively, input and output messages.

Definition 2: An *Event/Message Path (EMP)* (Jorgensen & ERickson, 1994) is a sequence of event executions communicated by messages. An *Atomic System Function (ASF)* is an input message, followed by a set of *EMP*s, and terminated by an output message.
An *EMP* starts with an event and ends when it reaches an event which does not issue any messages of its own. An *ASF* is an elemental function visible at the system level. As such, *ASF*s constitute the point at which integration and system testing meet, which results in a more seamless flow between these two forms of testing.

From an abstract point of view, a distributed computation can be described by the types and relative order of events occurring in each process. Let E_i denote the set of events occurring in process P_i, and let $E = E_1 \cup .... \cup E_n$ denote the set of all events of the distributed computation. These event sets are evolving dynamically during the computation; they can be obtained by collecting traces issued by the running processes. As we assume that each P_i is strictly sequential, the events in E_i are totally ordered by the sequence of their occurrence. Thus, it is convenient to index the events of a process P_i in the order in which they occur: $E_i = \{e_{i1}, e_{i2}, e_{i3},...\}$. We will refer to this occurrence order as the standard enumeration of E_i (Schwarz & Mattern, 1994).

The execution behaviour of a distributed computation is non–deterministic. Because of indeterminacy, it is difficult to know the possible execution behaviours of distributed software, to exactly identify the execution behaviour to be tested and to control the software execution for testing a specific execution behaviour. Based on the analysis of execution behaviour of distributed software, a message–flow–graph–based model is not suited for modeling the execution behaviour of distributed software. Therefore, a Distributed Message Flow Graph (*DMFG*) is proposed.

Definition 3: A *Distributed message flow graph (DMFG) of a distributed software DS* is a pair (D,W), where D is a set of *MFG*, D = {Gp_1, Gp_2,..., Gp_m } where *MFG* Gp_i corresponds to process i and W set of communication edges, W = { $c_{i,j}$,..., $c_{m,n}$ } in a distributed software. $c_{i,j}$ is a subset of $E_i \times E_j$, where E_i in Gp_i , E_j in Gp_j and $Gp_i \neq Gp_j$. A *Distributed Event/Message Path (DEMP)* is a sequence of event executions communicates by messages in the same process or between different processes. An *Distributed Atomic System Function (DASF)* is an input message, followed by a set of *DEMP*s, and terminated by an output message.

Example: A distributed database system for a Grade Report. Consider a grade report database system that has four relations (shown in Figure 3) distributed over two different processes which communicate with each other by message exchange.

STUDENT

student id	student name	
	first name	surname
945216775	Huey–Der	Chu
:	:	:

COURSE

course id	course name	teacher id
CS2010	Data Base	N4508
:	:	:

TEACHER

teacher id	teacher name
N4508	Michael Lee
:	:

GRADE

student id	course id	score
945216775	CS2010	B
:	:	:

Figure 3: A distributed database system for a grade report

Query: Given a Student name and Course name to get the grade report of Figure 4.

A Grade Report for Huey–Der Chu		
Course id	Course name	Grade
CS2010	Data Base	85
CS2015	Algorithm	80
:	:	:

Figure 4: A Grade Report

The *DMFG* is used for describing the behaviour shown in Figure 5:

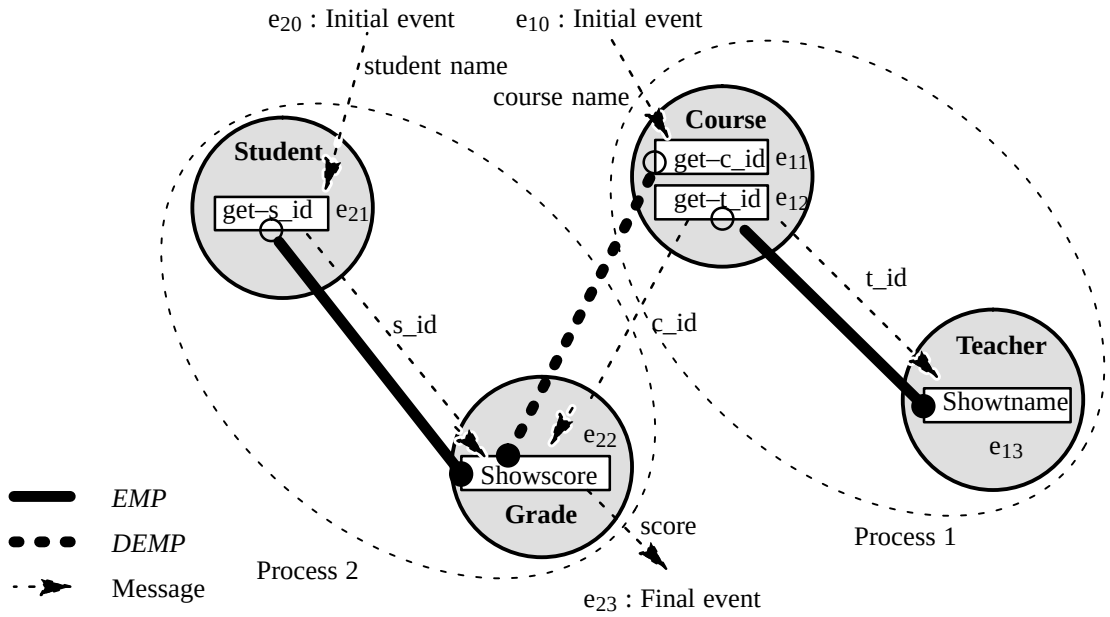


Figure 5: A *DMFG* for a computation for a grade report

In the get–c_id event, we can get the course id for the given course name from the COURSE relation and pass this course id to the show–score event. In the get–s_id event, we also can get the student id for the given student name from the STUDENT relation and pass it to the show–score simultaneously. When receiving both the course id and the student id, we can get the value of score from the Grade relation.

In a *DMFG*, it is sometimes impossible to say that one of two events occurred first. The relation 'happened before' is therefore only a partial order of events in this graph (Lamport, 1978). On the contrary, events on a *MFG* are totally ordered, so this order can easily be determined, since it is possible to use the same clock to determine the time at which each event occurs. The 'happened before' relation defined over E in *DMFG* determines the causal order of those events. We can formalize this relation by defining the causality relation as follows:

Definition 4: In *DMFG*, the *causality* relation $\longrightarrow \subseteq E \times E$ is the smallest transitive relation satisfying:
(1) If $e_{ki}, e_{kj} \in E_k$ occur in the same process P_k, and i < j in *DEMP*, then $e_{ki} \longrightarrow e_{kj}$.
(2) If $e_{ki} \in s_k$ is a send event and $e_{kj} \in r_k$ is the corresponding receive event in the same P_k, then $e_{ki} \longrightarrow e_{kj}$.
(3) If $(e_{ki}, e_{lj}) \in W$, then $e_{ki} \longrightarrow e_{lj}$, where $e_{ki} \in E_k$ and $e_{lj} \in E_l$, $E_k \neq E_l$.
Any pair of events not related by the causal order are logically concurrent and cannot affect each other. The determination of an ordering of events in a distributed system can be described in a system of causal timestamps based on partially ordered logical clocks (Fidge, 1991; Lamport, 1978; Schwarz & Mattern, 1994). When a receive event is executed, the logical clock in the clock vector is updated to be greater than both the previous local value and the logical clock of the incoming message. The recording of causal timestamps is useful for detecting whether or not the determination of an ordering of events in distributed systems.

While causal timestamps allow us to determine the relative order of any pair of events, they cannot be used to determine if there are any intervening events (Cherirton & Skeen, 1993). This means that messages can be delivered in an order that violates causality. As an improvement, we extend the —> relationship to include messages. Following definition 4 of the causality relation —> of events, we define the causality relation of two messages as follows:

Definition 5: In *DMFG*, the *causal message ordering* —> $\subseteq (M \cup W) \times (M \cup W)$ is the smallest transitive relation satisfying:

(1) If $m_i, m_j \in M_k$ occur in the same process P_k, and $i < j$ in *DEMP*, then m_i —> m_j.

(2) If a process receives m_i prior to m_j, then m_i —> m_j.

Causal message ordering guarantees that the order of delivery of messages does not violate causality in systems of communicating processes. Typically, messages are delayed at the receiver until all causally preceding messages are delivered. Specifically, if two the same input messages are sending to execute the same *DASF* and the sending of one input message happens before the sending of another input message, then the output message corresponding to the first input message is delivered before the second output message at all processes in the *DASF*.

3.2 The SMAD Tree

Input is constructed from data of different characteristics that are called input attributes. Associated with each input attribute is a syntax structure. The structure can be decomposed into a lower level substructure and so on, until further decomposition is not possible. The lowest level substructure is called a basic element. If the basic element is numerical then the lower bound and the upper bound of the element are given under the element. The overall structure is a tree. The tree can be arranged as a linear list with the structure preserved by a set of symbols called the tree symbols. The list is called a symbolic input attributed decomposition (SIAD) tree. It is used to represent the hierarchical and ‘‘network” relation between input elements and incorporate rules into the tree for using the inputs.

To guide testers in testing distributed software, the tool, the SMAD tree, is presented. Extending this concept of the SIAD tree, we can specify all possible delivered messages between events by means of the ‘‘Symbolic Message Attribute Decomposition” (SMAD) tree. It combines with classification and syntactic structure to specify all delivered messages in the *DMFG*. In the upper level of the SMAD tree, we classify all delivered messages into three types of message: input message, intermediate message and output message. Each type of message has a syntactic subtree describing the characteristics of messages with a time domain so that it can be determined whether messages were delivered in an order consistent with the potential causal dependencies between messages. The structure of the SMAD tree for the grade report system is shown in Figure 6:

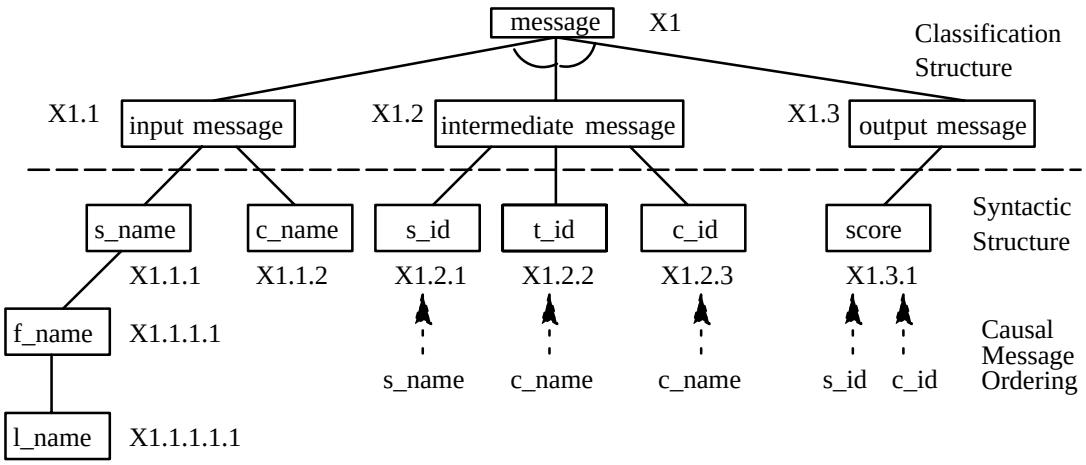


Figure 6: A tree structure of the SMAD tree

The SMAD tree is used to define test cases, which consist of an input message plus a sequence of intermediate message corresponding to messages in *DEMP*s, to resolve any non–deterministic choices that are possible during software execution, e.g., exchange of messages between processes. In other words, there are two uses of the SMAD tree: one is to describe abstract syntax of test data (including temporal aspects); another one is that one SMAD tree is instantiated for each test, to hold data occurring during the test.

The simple SMAD tree for messages in this *DMFG* is shown in Figure 7. A detailed SMAD tree depends on the decompositions of messages.

index	symbol	tree element	rule
1	X1	message	
2	X1.1	input message	
3	X1.1.1	student_name, K1 bytes	1,7
4	X1.1.1.1	first name, K2 bytes	2,4
5	X1.1.1.1.1	last name, K3 bytes	3,4
6	X1.1.2	course_name	4,8,9
7	X1.2	intermediate message	
8	X1.2.1	student_id	5,10
9	X1.2.2	teacher_id	5
10	X1.2.3	course_id	5,10
11	X1.3	output message	
12	X1.3.1	score, K4 bytes	4,6

(a)

rule index	rule description	subrule index
1	K1 = K2 + K3	1,2
2	K2	1
3	K3	2
4	Excluding characters + * / ’ ” < > & $; :...	
5	Including characters 0 1 2 3 4 5 ...	
6	K4	3,4
7	Happened before student_id	
8	Happened before course_id	
9	Happened before teacher_id	
10	Happened before score	

(b)

subrule index	subrule description	remark
1	$1 \leq K2 \leq 20$	length of first name
2	$2 \leq K3 \leq 15$	length of last name
3	$1 \leq K4 \leq 3$	length of score
4	K4 is an integer	

(c)

Figure 7: A simple SMAD tree for a grade report database system

In Figure 7 (a), the tree has four columns: the index, the symbol, the element and the rule index. The indices are for sampling use. The symbols preserve the tree structure of the tree elements in Figure 6. A tree element is a node in Figure 6 with some explanation of the node. A rule index points to a rule that governs the use of the tree element in constructing an input unit. The rules governing the use of tree elements in the SMAD tree can be listed as shown in Figure 7 (b). A rule index is used to identify the rule to be used and the rule description is the rule of interest. The sub–rule index indicates a sub–rule to supplement the use of the rule, as shown in Figure 7 (c).

3.3 Statistical Analysis

Testing a piece of software is equivalent to finding the defect rate of the product unit population generated by the software. The defect rate is defined as the ratio of the number of product units that are defective to the total number of product units that the software has generated. The total number of product units, denoted by N, of any non–trivial piece of software ranges from extremely large to infinite, but can still be treated as an object of statistical interest. Although impossible in practice, it can be conceptually assumed that all N units have been produced and analyzed. Each of them can be classified as defective or non–defective. If there are D units that are defective, then the product unit population defect rate, denoted by θ, is $\theta = D/N$. Since it is impossible to obtain all N units, the best approach is to estimate by means of statistical sampling. If the population is conceptually shuffled, it provides a basis for applying the principle of binomial distribution sampling. The application of the distribution often arises when sampling from a finite population consisting of a finite number of units with replacement, or from an infinite population consisting of an infinite number of units with or without replacement. The probability of getting x defectives in a sample of n units taken from a population having a defect rate of θ is given by the binomial distribution:

$$b(x) = \binom{n}{x} \theta^x (1-\theta)^{n-x}$$

The mean and variance of the distribution are given by:

$$\mu = n\theta$$
$$\sigma^2 = n\theta(1-\theta)$$

A sample of n units is taken randomly from the population. If it contains d defective units, then the sample defect rate, denoted by θ^0, is $\theta^0 = d/n$. If n is large enough, then the rate θ^0 can be used to estimate the product unit population defective rate θ. These two major testing issues are discussed in the following sections.

How good the software is after testing

The defect rate of the population can then be estimated from d. The estimate may be expressed in an interval called *100c%* confidence interval, where $0 \le c \le 1$. An approximation of the *100c%* confidence interval of the population defective rate may be computed by:

$$\left[\theta^0 - t_{n-1,\alpha/2} \sqrt{\frac{\theta^0(1-\theta^0)}{n}} \;,\; \theta^0 + t_{n-1,\alpha/2} \sqrt{\frac{\theta^0(1-\theta^0)}{n}} \right] \qquad (1)$$

where $t_{n-1,\alpha/2}$ is called the value of the *Student t–distribution* at $n-1$ degrees of freedom and $\alpha = 1 - c$ is called a risk factor (In statistics, a binomial distribution can be approximated by a normal distribution). Formula (1) can be used to estimate the mean of the product unit population, denoted by μ. Once the value of μ is estimated, the product unit population defect rate θ can be computed by $\mu = n\theta$. If the value of θ is acceptable, then the product unit population is acceptable. The piece of software is acceptable only when the product unit population is acceptable. Therefore, the estimated product unit population defect rate θ can be viewed as the software quality index.

When to stop testing

The accuracy of the estimates depends on the sample size. In general, the larger the size, the more accurate the

estimate. The value of n may be computed by the formula:

$$n = \frac{z^2(1-\theta)}{\alpha^2\theta} \qquad (2)$$

where α is the desired accuracy factor such that $|\theta - \theta^0| = \alpha\theta$, and z is the value of $z_{\alpha/2}$, which is the number of standard deviations in the normal distribution such that the area to its right under the normal curve is $\alpha/2$. The value of $z_{\alpha/2}$ is the same as $t_{n-1,\alpha/2}$ if n is large, e.g., $n \geq 30$. Since the population defect rate θ is unknown, the determination of n requires dynamic adjustment during sampling. An adjustment procedure, which is iterative in nature, is given as follows:

Step1: Take an initial sample of a small size, n_0 units (e.g., 50) from a software product population by executing n_0 input units.

Step2: Let θ_0^0 be the defect rate of the sample of size n_0,

Step3: Compute the sample size n_{i+1} by formula (4.2) as follows:

$$n_{i+1} = \frac{z^2(1-\theta_i^0)}{\alpha^2\theta_i^0}$$

where θ_i^0 is the cumulative defect rate of the cumulative sample units n_i already taken after the ith iteration, for $i = 0,1,2,...,$

Step4: If $n_{i+1} > n_i$, then take $(n_{i+1} - n_i)$ additional units and repeat Step3 and Step4.

Step5: Else stop. The total number of sample units taken is sufficient.

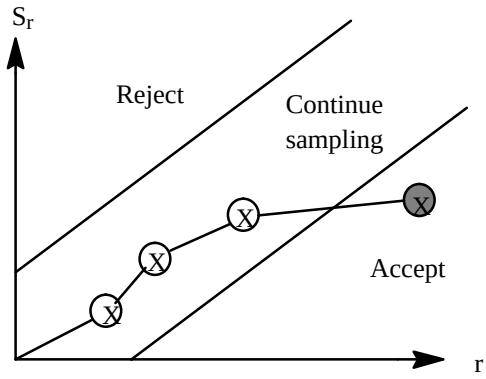
The final sample defect rate is then used to estimate μ and σ^2. In any factory it is almost impossible to produce a defect–free product lot: therefore, the conformance of product quality is usually measured by the defect rate being less than an acceptable number, e.g., $\theta < 0.01$. With a statistical sampling method, a confidence level of 98% certainity can be imposed on the final value of the estimated defect rate.

Quality analysis

If the software output is defined in terms of the ‘‘product unit”, then the output is a collection of product units called the output population of the software. For any non–trivial software, the population contains a very large number of units. The goal of software testing is to find certain characteristics of the population such as the ratio of the number of defective units in the population to the total number of units in the population. The ratio may be called the defect rate of the population and may be imposed on the software as the software quality index.

The sampling processing procedure discussed in previous sub–section represents the drawing of a product unit at random from a binomial distribution. If the number of defective product units in the sample is less than the tolerable number of defectives determined by the selected sampling plan, then the software can be delivered to users as acceptable. Otherwise, the developer should improve the quality by correcting the errors found during the test. The quality statement defines software quality that is equivalent to p% of the output population being non–defective (the acceptance level). The result of the iterating sampling process, sample n, will be dynamically saved into a sample size file for providing an information to the test data generator. The values of confidence interval also is computed and will be saved into a file for the range of defect rate for supporting the evaluation of software quality by the test results validator.

To analyse the failure data collected during the statistical testing a reliability model is need. The model is based on a control chart with three regions, reject, continue and accept, as shown in Figure 8.



S_r: the cumulative number of defectives in the sample of r units

Figure 8: A control chart for statistical analysis

The defect rate is plotted in the chart. As long as the plots fall in the continue region, the testing has to continue. If the plot falls in the rejection region, the software reliability is so bad that it has to be rejected and re–engineered. If the plots fall in the acceptance region, the software can be accepted based on the required quality statement with given confidence and the testing can be stopped.

3.4 Test Data Generator of Messages

The process of automated test data generation follows as the following steps:

Step1: Generate the number of test data *M* for each sample by random number seed,

Step2: The construction of test data using the input message part of the SMAD tree can be accomplished as follows:

2.1 Let *K* be the number of elements in the SMAD tree. Each element in the tree is indexed by a number ranging from 1 to *K*. A random number selected from [1,*K*] is produced by using a random number generator.

2.2 The element with its index equal to the random number is selected.

2.3 If the element has a parent in the SMAD tree, then go backtracking to select it.

Step3: A total of *M* elements will be randomly sampled from tree for designing test data.

For example, there are 6 elements in the input message of the SIAD tree of Figure 7. A test data includes one student name and several course names. The student name is generated from the index 5 of the SMAD tree. Two course names are to be chosen for a sample using random number generation producing 6. According to this process, the test data can be generated as Table 1:

Table 1: A test data is drawn from SIAD tree

Index	Tree Symbol	Tree Element	Remarks
4	X1.1.1.1	Huey–Der	Descriptive element
5	X1.1.1.1.1	Chu	Sampled element
6	X1.1.2	Database	Sampled element
6	X1.1.2	Algorithm	Sampled element

The ‘happened before’ rule in SMAD tree can be used for examining the causal message ordering. A partial order of all possible messages between events can be built based on the ‘happened before’ rule in SMAD tree. For example, there are six messages, student_name, course_name, student_id, teacher_id, course_id and score, in SMAD tree for a grade report database system as shown in Figure 7. The rules 7, 8, 9, 10 are therefore only a partial order of messages in this system. According to definition 4.5, Figure 9 shows a space–time diagram for the causality relation.

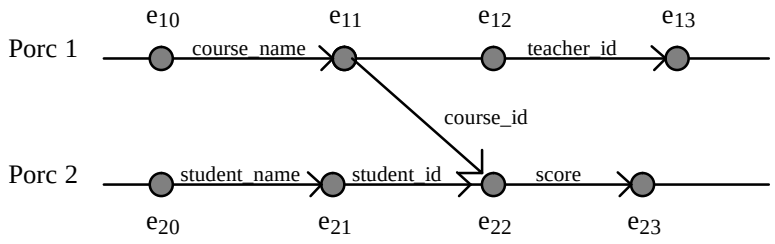


Figure 9: A causality relation for messages

3.5 Test Results Validator

The output message part in the SMAD tree can be used as a tool for describing the expected result which satisfies the user’s requirement and as a basis for analyzing the output messages automatically, particularly in non–numerical applications such as an interpreter and updating a data base.

Validate the syntactic structure

The result of executing an input by the software can be classified into two categories: defective and non–defective. Each product unit must be carefully analyzed for its conformance to the software requirements in order to reach the classification. The outcome of the analysis leads to classifying the output into either of the categories which, in turn, results in the acceptance or rejection of the software. Any unfair bias can increase the producer’s risk of having good software rejected or can increase the user’s risk of accepting poor software.

Test results can be inspected by manual, semi–manual or automatic means, which depend on software applications. A SMAD tree can be used as a tool for describing the expected result which satisfies the user’s requirement and as a basis for analysing the product unit automatically, particularly in non–numerical applications such as an interpreter and updating a data base. It is a data structure containing a record for each output element with fields for the attributes of the output element.

Based on the SMAD tree, the process of automated test result analysis is shown in Figure 10:

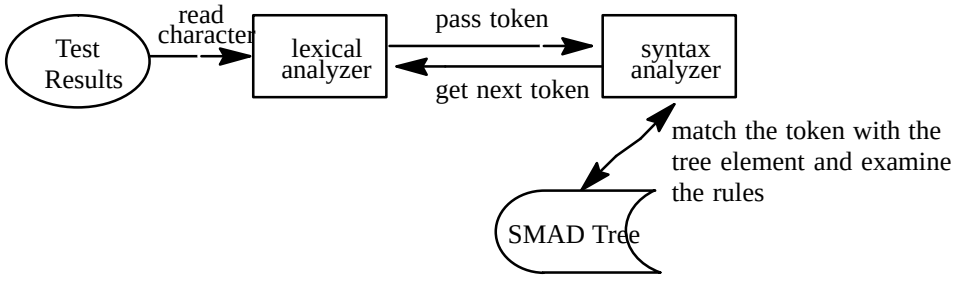


Figure 10: The process of test result analysis

The lexical analyzer is the first phase of inspection. It main task is to read the characters of test results and produce as output a sequence of tokens. In this process, the syntax analyser obtains a string of tokens from the lexical analyser, as shown in Figure 4.14, and verifies that the string is defective or nondefective by matching the token with the tree element and examining the rules in the SMAD tree. According to the different types of software applications, the algorithm of inspection based on its SMAD tree can be separately designed. The main advantage of using the SMAD tree here is that we do not need a test oracle to compute expected results. The SMAD tree can be used directly for automatic inspection whether or not the results produced by the software are correct.

Examine the causal message ordering

Because of the existence of non–deterministic behaviour, it is generally impossible to test all distinct execution behaviours of distributed software by proper selection of test cases and reproduce previous test results by repeating execution with the same input. The causal message ordering between messages is a fundamentally new approach to the analysis and control of execution behaviour of client/server applications. Based on the 'happened before' rule in the SMAD tree, we can receive an accurate representation of the message orderings and derive all possible totally ordered interleavings. As a result, the technique greatly reduces the number of tests required. It is never necessary to perform the same computation more than once to see whether different message orderings (interleavings) are possible. However, we need to test the causal message ordering to guarantee that order of delivery of messages does not violate causality in systems of communicating processes.

We can examine the execution of different computation paths which derive from different test data or from the same test data (repeated execution) to examine the casual message ordering between messages. Specifically, if two input messages are sending to execute the same *DASF* and the sending of one input message happens before the sending of another input message, then the output message corresponding first input message should be delivered before the second output message at all processes in the *DASF*. Figure 11 describes this behaviour in *DMFG*.

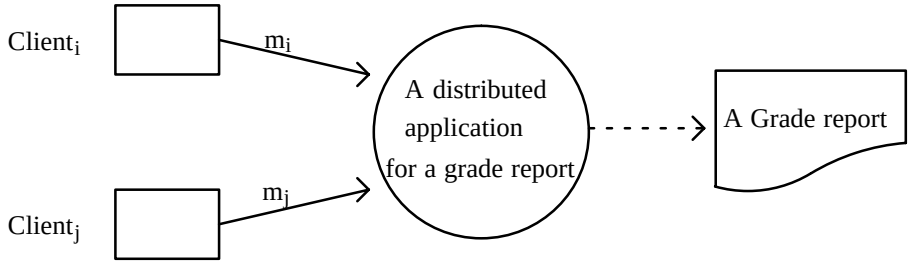
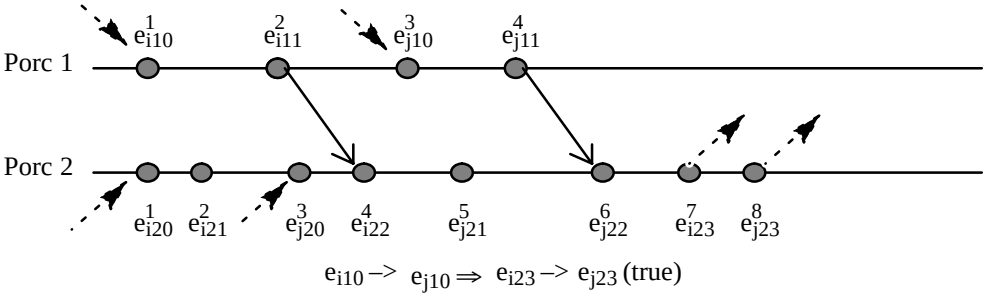


Figure 11: Examine a *DASF* with two clients in a distributed software

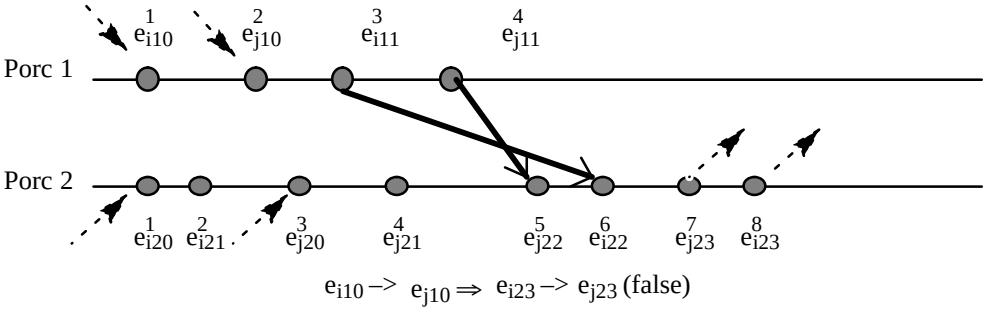
Consider a *DASF* on a distributed computation with receiving two input messages m_i and m_j. We can test the output messages corresponding m_i and m_j using the causal message ordering to ensure the causal consistency of processes in this client/server application.

Figure 12 shows a space–time diagram for the events in the computation with two different input messages in two different cases. There are two input messages m_i and m_j corresponding two different execution behaviours: $DASF_i$ for client$_i$ and $DASF_j$ for client$_j$, where

$DASF_i$: e_{i10} –> e_{i20} –> e_{i11} –> e_{i21} –> e_{i22} –> e_{i23}
$DASF_j$: e_{j10} –> e_{j20} –> e_{j11} –> e_{j21} –> e_{j22} –> $e_{j23.}$



(a) Case 1: causal message ordering is consistent



(b) Case 2: causal message ordering is inconsistent

Figure 12: A space–time diagram of a distributed computation

We can observe the execution behaviour from this space–time diagram. In case 1 in (a), We can test that the causal message ordering is consistent. However, when we examine case 2 in (b) , we can see that it violates causality, because the message for $client_j$ at e_{22} should arrive later then the message for $client_i$ according to my expectation. In this case, student *i* will get a grade report with student *j*'s grades.

4 Case Study : A Simple Banking Application

4.1 Problem Description

A banking application is an embedded software system which is commonly seen inside or outside banks to drive the machine hardware and to communicate with the bank's central banking database. This application accepts customers requests and produces cash, account information, database updates and so on. In this chapter, a Simple Banking Application (SBA) will be designed as a 3–tier client/server application as shown in Figure 13 within a banking enterprise, more specifically a corporate and distributed database collection for the personal data of customers, the balance status of customers, the password data and account type data. The corporation seeks to assimilate their data sources into one virtual data store and access it through a common interface.

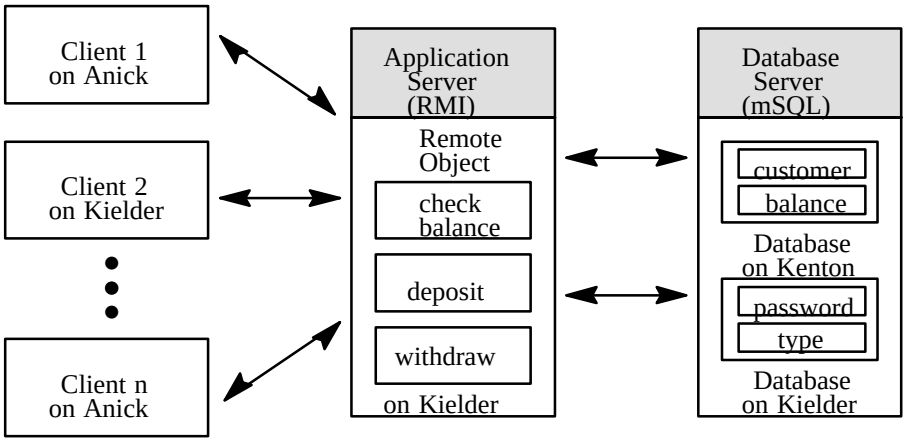


Figure 13: Three–tier System Structure

There are four business activities at this application: check balance, deposit money, withdraw money and print the statement. One of transactions is as shown in Figure 14.

```
    Input : account id, account type and password
    ---------------------------------------------
    Business activities:
            1. Check the balance
            2. Deposit 500 pounds
            3. Check the balance
            4. Withdraw 250 pounds
            5. print the statement
    ---------------------------------------------
    Output: balance or a banking statement
```

Figure 14: A transaction of the banking application

This standard transaction will accept customer requests (checking, depositing, withdrawing and printing) after the customer has input the account id, the account type and the correct password on the *Client* site. SBA will retrieve the balance from the database on the *Database Server* site, process the request on the *Application Server* site and save the balance back to the database. It also will produce the balance or print a banking statement to the customer.

4.2 The implementation of an automated software testing

According to the following test requirements, an integrated test environment is designed for the simple banking client/server application and implemented under Java Development Kit (JDK).

- To set up test requirements, including the functional requirements and quality requirements,
- To execute automated testing until it has been sufficiently tested (when to stop testing),
- To re–execute the input units which have been tested (regression testing),
- To execute the testing first for only one client and later for several clients,
- To test all business activities which should be traversed at least once (test coverage)
- To produce the test execution report, the test failure report and the test quality report.

The environment of the automated software testing consists of eight modules as shown in Figure 15.

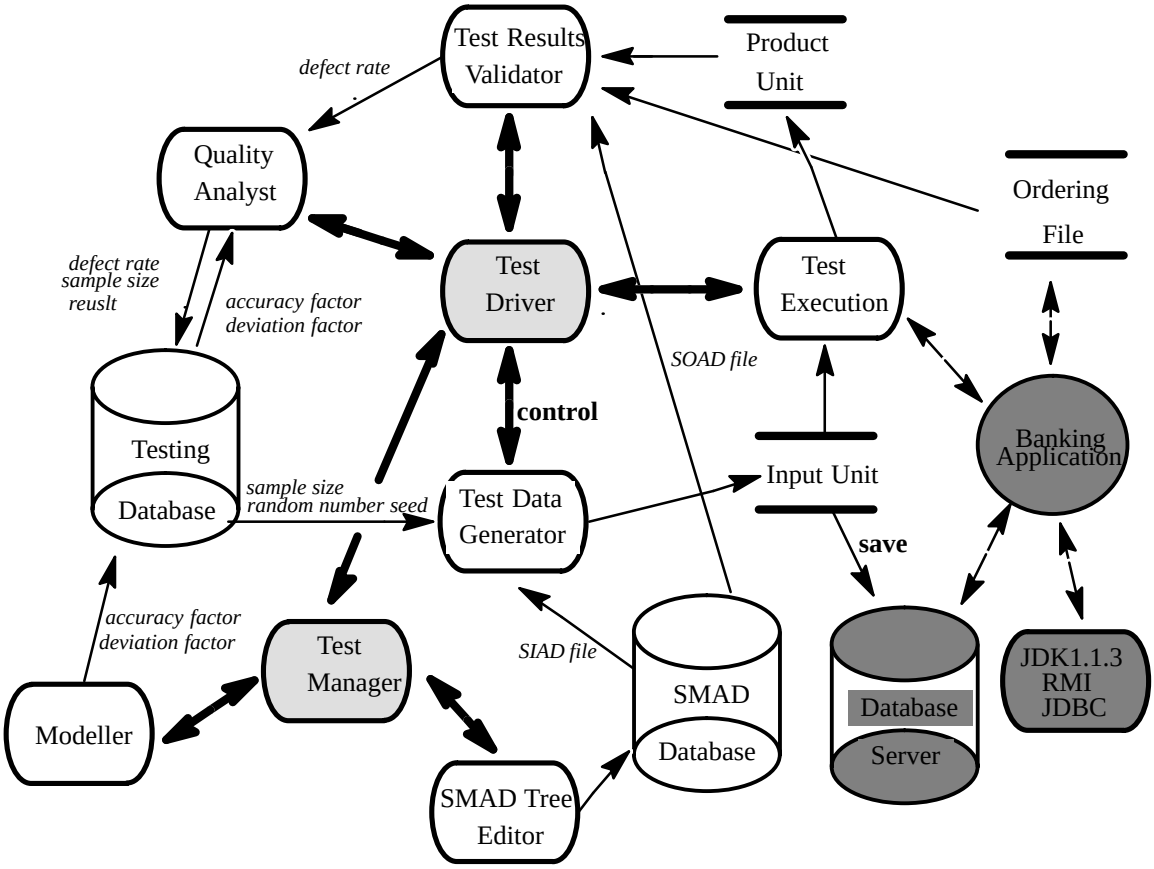


Figure 15: An Integrated Test Environment for the Simple Banking Application

Test Manager

The Test Manager receives a command from the tester and communicates with the functional module to execute the action and achieve the test requirements. It executes two main tasks: data management and control management.

1. Data management: in this implementation, the test manager maintains two databases using mSQL, Testing and SMAD. The Testing database saves the values of the accuracy factor, the deviation factor and the initial sample size from the Modeller and the values of the defect rate, the sample size and the testing result from the Quality Analyst. It provides the values of the accuracy factor and the deviation factor to the Quality Analyst for the dynamic sampling process

and the values of the sample size and the random number seed to the Test Data Generator for generating test data based on the SIAD tree file in the SMAD database. Three dynamic files, the input unit file, the product unit file and the testing ordering file, will be produced during the testing process. The contents of these files will be seen through the Test Manager.

2. Control Management: the Test Manager controls three main functional modules: the Modeller, the SMAD Tree Editor and the Test Driver. The Modeller is used for receiving the test plan such as test requirements and test methods from the user, creating the test plan documentation and saving some values for the Testing database. The

documentation provides support for test planning to the test driver as well as the SMAD tree editor for specifying messages among events. The SMAD Tree Editor is used to create the SMAD tree file that can be used to describe the abstract syntax of the test cases as well as to trace data occurring during the test. The SMAD database provides the structure to the Test Data Generator for generating the input unit and to the Quality Analyst to inspect the product unit. The Test Driver executes the main task of testing which is described in more detail in the next section.

Test Driver

The Test Driver sets up the test execution environment for the simple banking application, which involves valid testing with Test Coverage, valid testing using the statistical approach and invalid testing. In the valid testing with the statistical approach, the Test Driver sets up the values of the initial sample size, the accuracy factor and the deviation factor, initiating the Test Data Generator to generate an input unit, sending it to the Test Execution to execute the application and getting the product unit and delivering it to the Test Results Validator. When the sample size is satisfied from the Dynamic Sampling Process, the Test Driver passes the latest value of the defect rate to the Statistical Inference Process for estimating the confidence interval of the mean and variance of the population. This is used to determine the acceptability of the application. The activities of the Test Driver are shown in Figure 16.

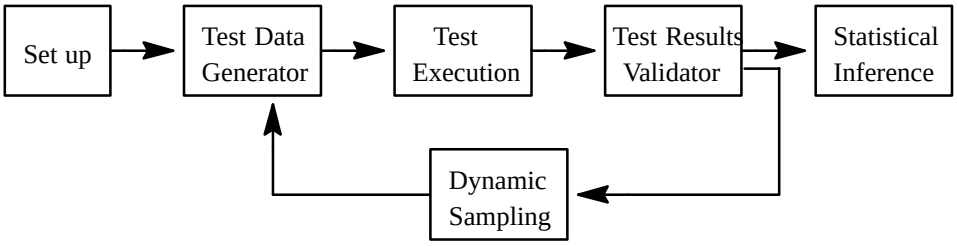


Figure 16: The activities of Test Driver

5 Experimental Results and Discussion

The simple banking application presented in previous section was implemented and tested on the statistics–based integrated test environment. All source codes for this implementation can be downloaded at *http://www.casq.org/site/banking/* which is under the web site for Chinese Association for Software Quality (CASQ) constructed and maintained by Huey–Der Chu 1998. This section describes how the automatic test of the application was conducted based on one client site and one server site and how the manual test of the application was conducted based on two client sites and one server site are examined and comments on the results obtained.

5.1 Automatic Testing on One Client Site

In our experience to run the test on this application on one client site, resulted in me not finding any defect on the test. In order to demonstrate the sampling process with statistical testing, we added a random number generator into the Test Results Validator. If the number is less then 10, the Test Results Validator will return ‘fail’ value.

The acceptance criteria for the sampling method in this application are: a sample of *n* units is to be taken randomly from the product unit population such that the sample defect rate θ^0 and the population defect rate θ differ with an accuracy factor of 0.1, that is $|\theta - \theta^0| = 0.1\ \theta$ and $\theta < 0.01$. In this implementation, given the probability is 0.95, the standard deviation factor *z* is found to be 1.96 from (Cho, 1988). As discussed in the Section 6.2.4, in this implementation the sampling plan implements the formula

$$n = \frac{z^2(1-\theta)}{\alpha^2\theta}$$

with z= 1.96 and $\alpha = 0.1$. The results of sampling are as shown in Figure 17.

Iteration	n (i+1)	n (i)	n (i+1) – n(i)	Defective Rate
1	1751	100	1651	0.18
2	2492	1751	741	0.1336
3	2449	2492	–43	0.1356
	Sample size = 2492		Defective rate = 0.1356	

Figure 17: An iterative sampling test results in testing

The sampling process stops at iteration 3 with a sample size of 2492 units and a sample defective rate of 0.1356. In other words, at iteration 1 the sample size is determined by

$$n_1 = \frac{1.96^2(1-0.18)}{01 \times 0.1 \times 0.08} = 1751$$

at iteration 2:

$$n_2 = \frac{1.96^2(1-0.1336)}{01 \times 0.1 \times 0.1336} = 2492$$

and at iteration 3:

$$n_3 = \frac{1.96^2(1-0.1356)}{01 \times 0.1 \times 0.1356} = 2449$$

Since $n_3 = 2492$ is less than $n_2 = 2449$, the total number of units sampled at iteration 2, the sampling process stops.

The 95–percent confidence interval of the mean of the population (for z= 1.96) can be estimated by:

$$\left[n\theta^0 - t_{n-1,\alpha/2} \frac{s}{\sqrt{n}} , n\theta^0 + t_{n-1,\alpha/2} \frac{s}{\sqrt{n}} \right]$$

where $n\theta^0 = 2492 \times 0.1356 = 388$ and

$$s = \sqrt{n\theta^0(1-\theta^0)} = \sqrt{2492 \times 0.1356 \times (1-0.1356)} = 17.09$$

$$t_{n-1,\alpha/2} = t_{2492,0.01/2} = 2.576 \text{ from Appendix 4 of (Cho, 1988)}$$

$$\sqrt{n} = \sqrt{2492} = 49.92$$

Thus, the range of population mean is found to be:

[337.118, 338.882]

The defective rate of the output population of the routine is estimated from this mean. Therefore, the 95–percent confidence interval of the defective rate is:

$$\left[\frac{337.118}{2492}, \frac{338.882}{2492} \right] = [0.1350, 0.1360]$$

In other words, the test results documented in the testing document show that the estimated product unit defective rate at the 95–percent confidence level is from 0.1350 to 0.1360. The software acceptance and test requirements documented in the requirements specification state that to be accepted, the software must have a product unit population defective rate of $\theta < 0.01$. Clearly, the value between 0.1350 and 0.1360 > 0.01, and therefore, the software product population does not meet the acceptance criteria. In this situation, the software developer should conclude that the application is not ready for delivery to the user. Further development is required to reduce the defective rate and improve the quality of the application.

5.2 Manual Testing on Two Client Sites

At this stage of implementation, we can only automatically run the tests on one client site, therefore, we tried to send test data files on two client sites to run the tests by hand. Two different types are given as following:

Two different input with two different DASFs

There are two test data files test1.in and test2.in corresponding two different execution paths: $DASF_a$ run on Anick.ncl.ac.uk and $DASF_k$ run on Kielder.ncl.ac.uk. Figure 18 shows the relation among test data files, test results files and the trace file on server site when the application under tests.

Two the same input with the same DASFs

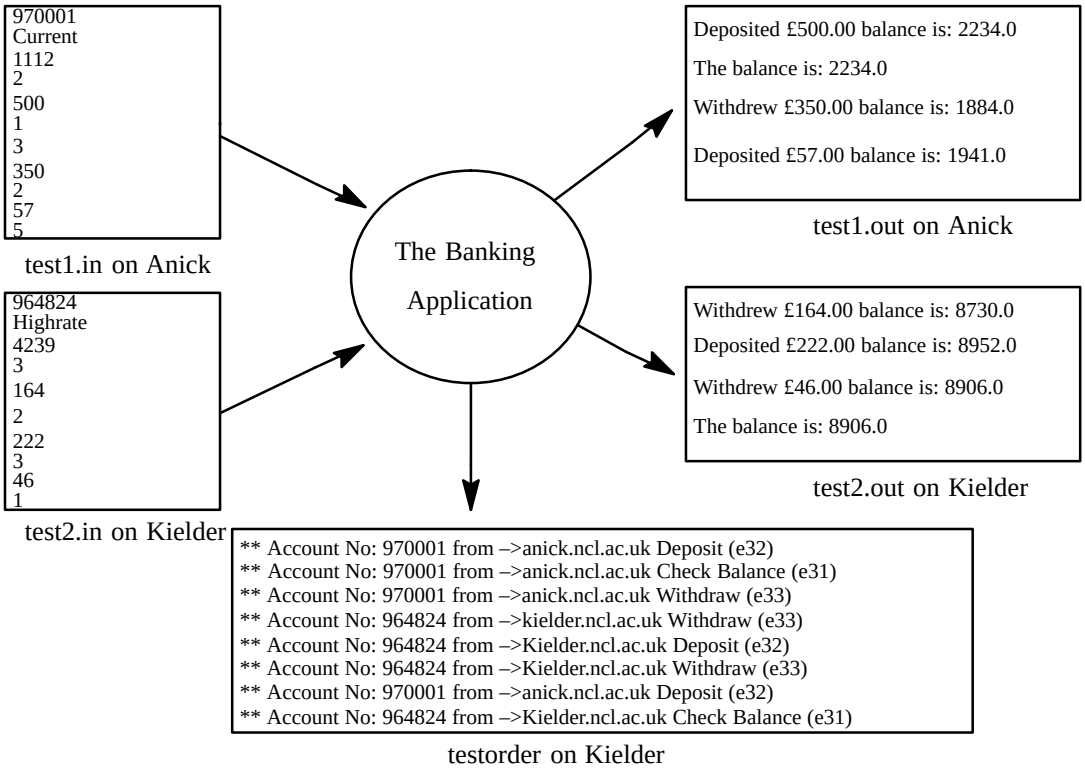


Figure 18: The application under test with two different input

In this case, two the same test data files test1.in and test1.in are corresponding the same execution paths: $DASF_a$ run on Anick.ncl.ac.uk and $DASF_k$ run on Kielder.ncl.ac.uk. Figure 19 shows the relation among test data files, test results files and the trace file on server site when the application under tests.

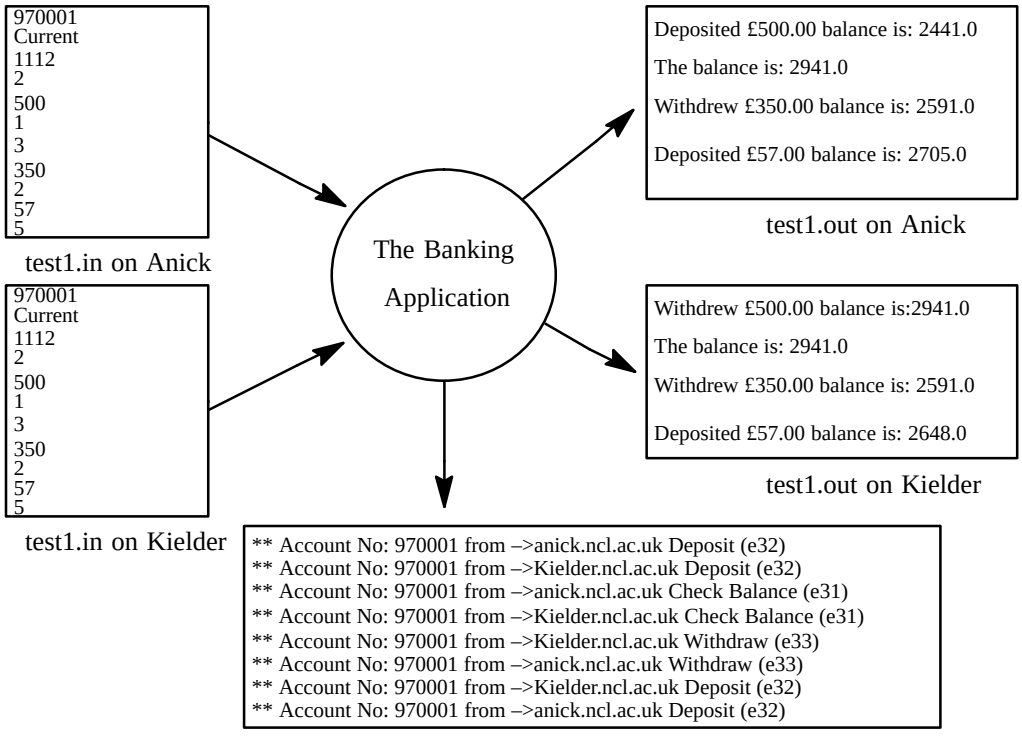


Figure 19: The application under test with two the same input

The test results are not satisfied, because the expect balance on Anick is 2298 and on Kielder is 2355. This

application didn't consider the problem of current access control which caused that the causal message ordering was inconsistent as shown in Figure 20.

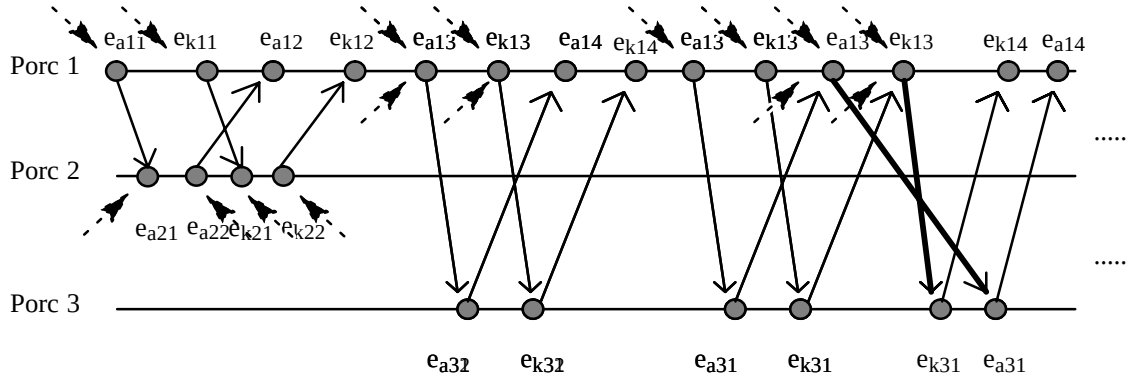


Figure 20: A space–time diagram for showing the causal message ordering

5.3 Comment

- Current automated testing tools focus on the two–tier client/server applications. However, the Gartner Group found 80 percent were planning for multi–tier (at least three–tier) client/server application (Mooney & Chadwick, 1998). In this implementation, a simple three–tier banking application was implemented, which was big enough to address middleware testing issues such as Java RMI and JDBC.
- In this implementation, we found that there are three basic premises using my approach to the automated testing: firstly, the modeling must precisely catch the behaviors of distributed applications, secondly, the requirements specification should be really defined and finally the testing tools must integrate well.

- According to the different type of software applications, we can use a number of different types of SIAD trees (a detailed description of these trees is given in Cho). Therefore, we need to design different Test Data Generators for each application. In other words, there is currently not a general test data generator which could be adapted for all applications.

- At this stage of implementation, we can only automatically run the tests on one client site and inspect the tracing file on the server site using a manual approach because the file can not be sent back by itself.

- According to the sampling method in this implementation, a large number of test data are needed to achieve high quality. In other words, the larger the size, the more accurate the estimate. However, the cost of sampling is almost negligible compared with that of a 100–percent inspection. In fact, if a population contains millions and millions of units, sampling is the only practical way to determine the defective rate of the population.

- The cost of testing with this approach was higher, because there was more front–end test planning in the work in developing the SMAD tree, however, this is effectively balanced by less test operation, since testing can be automatically achieved and the quality index can be estimated.

- In this implementation, we constructed invalid test data using invalid testing component to test whether the

application could detect erroneous test data. In this work, the error message of the input data was displayed in the executing record file (product unit), which satisfies our expectation.

● In this implementation, the same test data can not automatically be sent to two or more client sites at the same time, therefore, we tried to copy a test data file and send it to another client site. When we ran the tests on two client sites simultaneously by manual test execution, a problem might arise when the database is accessed with the same account id without the consideration of mutual exclusive access. Therefore, we need to consider how to generate the test data and broadcast it to multiple client sites to find out a solution to this problem.

6 Conclusion

In this paper a graph model suited for testing purposes is proposed for modeling the execution behaviour of the distributed computation. To guide testers in testing client/server application, the SMAD tree which specifies all possible delivered messages is presented. The SMAD tree is used to define test cases, which consist of an input message plus a sequence of intermediate message, to identify the execution behaviour to be tested. Based on the SMAD tree, we develop a framework which not only can generate the input messages and a sequence of intermediate message pairs (in/out events) with their time domain, but can inspect the test results, both with respect to their syntactic structure and the causal message ordering under repeated executions.

The early the testers incorporate an automated test approach into the development cycle, the greater the return on the investment. In this framework for automated testing, changing the layouts does not need to be acknowledged during test execution since the SMAD tree is static. Any change for the message layouts of the application can be done by the SMAD tree before the test execution. In other words, some testing activities such as modelling and specifying messages by SMAD tree can be done early. Moreover, the interaction behaviour between client and server sites can be examined by the causal message ordering.

The implementation of the simple banking application written using Java Remote Method Invocation (RMI) and Java DataBase Connectivity (JDBC) shows the testing process of fitting it into the framework. This application is simple, however, it is big enough to address middleware testing issues such as Java RMI and JDBC which is a new area in client/server testing.

Reference

Beizer, B. (1990) *Software Testing Techniques* (Second ed.). (Van Nostrand Reinhold, New York).

Cheriton, D. R., & Skeen, D. (1993) Understanding the Limitations of Causally and Totally Ordered Communication, in *14th ACM Symposium on Operating Systems Principles*, 44–57.

Cho, C.K. (1988) *Quality Programming – Development and Testing Software wih Statistical Qualty Contro*l (John Wiley & Sons, New York).

Chu, H.D., Dobson, J. and Liu, I.C. (1997) FAST: a framework for automating statistics–based testing, *Software Quality Journal*, **6**(1), 13–36.

Curritt, P.A., Dyer, M. and Mills, H.D. (1986) Certifying the Reliability of Software, *IEEE Transactions on Software Engineering*, **SE–12**(1), 3–11.

Deck, M. (1996) Cleanroom practice: a theme and variations, in *Proceedings of the 9th International Software Quality Week* (Software Research Institute, San Francisco).

Dyer, M. (1992) *The Cleanroom Approach to Quality Software Development* (John Wiley & Sons, New York).

Fidge, C. (1991) Logical Time in Distributed Computing Systems, *IEEE Computer*, **24**(8), 28–33.

Ince, D. (1987) The automatic generation of test data, *Computer Journa*l, **30**(1), 63–69.

Lamport, L. (1978) Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM*, **21**(7), 558–565.

Jorgensen P.C. & Erickson, C. (1994) Object–Oriented Integration Testing, *Communication of the ACM*, **37**(9), 30–38.

Mooney, K. and Chadwick, D. (1998) Overcoming the Challengs of Testing Client/Server Applications, Available at <*http://www.rational.com/support/techpapers/challenges/* >.

Myers, G.J. (1978) *The Art of Software Testing* (John Wiley & Sons, New York).

Norman, S. (1993) *Software Testing Tools* (Ovum Ltd, London).

Pettichord, B. (1996) Success with Test Automation. In *9th International Software Quality Week (QW'96)*, San Francisco, CA.

Quinn, S.R. and Sitaram, M. (1996) Shrink–wrapped and custom tools ease the testing of client/server applications , Byte, September 1997, 97–102.

Schwarz, R. & Mattern, F. (1994) Detecting Causal Relationships in Distributed Computations: in search of the holy grail, *Distributed Computing*, **7**(3).

Thévenod–Fosse, P. and Waeselynck, H. (1991) An investigation of statistical software testing, *Journal of Software Testing, Verification and Reliability*, **1**(2), 5–25.

Thévenod–Fosse, P., Waeselynck, H. and Crouzet, Y. (1995). Software Statistical Testing. In *Predictably Dependable Computing Systems* (Springer, London), 253–272.

Whittaker, J.A. and Thomason, M.G. (1994) A Markov Chain Model for Statistical Software Testing, *IEEE Transcations on Software Engineering*, **SE–20**(10), 812–824.

Zallar, K. (1997) Automated Software Testing – A Perspective, In *10th International Software Quality Week (QW'97)*, San Francisco, CA.

The Automated Build Verify Test: A Valuable Time Saver

View Slides

Bruce Kovalsky
Quality Assurance Manager
Fidelity Technology Solutions
11400 SE 8th Street, Suite 215
Bellevue, WA 98004
Bruce.Kovalsky@fmr.com

Fidelity Technology Solutions™

A Fidelity Investments Company

Abstract:
Development and implementation of an automated Build Verify Test (BVT, or also known as a Acceptance Test or Smoke Test) saves Test teams valuable time by automatically determining if a new application build is solid enough in passing basic functionality to release to testing. The BVT has proven invaluable to many organizations as a method to provide early detection of major program defects as well as give confidence to the Test team once a build is first released by Development.

Keywords:
Automated Testing, Defect Detection, Verification and Validation, Software Builds

About the Author:
Bruce Kovalsky has been a Quality Assurance Manager for Fidelity since 1996, and has been leading QA and Test teams at various Seattle-area companies (including IBM) for the past 8 years. After receiving a Bachelor's Degree in Computer Science from the University of California at Berkeley, he spent 8 years developing software for the Aerospace industry. Bruce then began focusing his career on Quality Assurance and Testing. He first used an automated testing tool in 1992 - managing to automate a VAX/VMS application with a capture-replay tool that used binary scripts. Since then, he has successfully used Windows script-based automated test tools (such as Visual Test and SQA Robot) to automate several products. He taught a Microsoft Test course for Software Testing Laboratories in 1995.

Overview

When testing any software application, there are many standard features that the Test team should exercise every build, because of their frequency of use or importance to the user. Since it can be time-consuming (and also get monotonous) performing these manually, and to ensure it always gets done, every software testing organization should attempt to create an automatic mechanism to perform these repetitive tasks.

Soon after a program's feature set has been frozen and initially implemented, Test personnel can begin development of an automated Build Verify Test (BVT) using any Automated Software Quality tool of choice (e.g., SQA Robot, Visual Test, QA Partner, WinRunner). Also known as an Acceptance Test or Smoke Test, this test will be run many times over the life of an application.

The purpose of the BVT is to broadly exercise the important features of the application to an appropriate level, so that after running it successfully (usually in 30 minutes or less, depending on the program's complexity) the Test team is confident they can begin formal testing knowing that the major application functionality works correctly. Sometimes, serious new defects or regressions are found during the running of the BVT, which alerts the Development team to fix the problem quickly before the build is released to the Test Team again.

The BVT becomes most useful in the later stages of a software development lifecycle, when an application gets closer to "release candidate" quality. This is when early knowledge of the presence of any serious defect *before* an entire Test team starts testing a build is extremely valuable.

Determining what to put in an automated BVT

Generally, the BVT contains a subset of what might be developed for more detailed automated functional and regression testing. This does not necessarily mean the BVT is short, trivial or unimportant. In fact, a good BVT performs a comprehensive set of tests that cover a necessary and sufficient set of an application's functionality.

Before spending large amounts of time developing a BVT, make sure the application's user interface (and data file format, if applicable) is well defined, and won't be subjected to significant changes. Needing to change the BVT to match any new GUI or file format could be very time-consuming and not cost-effective.

When thinking about what to include in a BVT, you should take a Breadth-first approach, as opposed to a Depth-first approach. The idea is to touch every important feature that the application has, initially at a high level. During every test cycle, be conscious of the tradeoff between depth and breadth of testing [Kaner 1]. If you rely on the BVT to handle the breadth side, you can concentrate on more depth testing via manual testing and other automated regression testing efforts.

As part of your test planning process, you should be creating test documentation to help organize, specify, and communicate the testing project. One of the core components that Kaner [1] advocates is the Function List – a list of the top-level (user-visible) functions, visible subfunctions, and inputs. Working with the appropriate resources, you should first create a draft of the Function List for your application, in a matrix or document structure.

Next, I recommend holding a meeting (early in the development lifecycle) with the leads of the Development, QA/Test and Product Management groups for the product that you are developing the BVT for. In this meeting, present the draft function list, and get a consensus on the content and completeness of the Function List. Then, assign priorities (regarding the importance

of the item to be tested) to each of lowest level items with respect to the user. Any type of priority weighting scale can be used. I use the following priority system:

1: Critical - Test in every build
2: Important - Test in most builds
3: Routine - Test occasionally
4: Minor – Test before release

From the priorities that the group decides on, an outline of the BVT can then be designed. Generally, you should try to include testing of Priority 1 and 2 items into a BVT, with inclusion of Priority 3 and 4 items when convenient. The following example (Figure 1) shows an excerpt of a function list I developed for a financial management application.

Sample Function List with Priorities

Module	Function	SubArea	Input Data	Priority
Portfolios	Search	Accounts	General/Specific Bank/Region/Office	1
			Single/Multiple Investment Officers	1
			Account Officer selected/not selected	1
		Clear All		3
		Open	Existing *.sch list file	4
		Save	To new *.sch list file	4
		Assets/Positions	Ticker/Asset #	1
			Category/Class/Asset Major/Industry Major	1
			Sector	1
			Asset Minor	1
			Industry Minor	1
		Search for	Holders/Non-Holders	1
	Holdings	Search	Account #	1
Assets	Search	Search All	Asset Number/Alpha Name/Ticker	1
		First 20	Attributes/Equities/Fixed Income/MBS/CMO	1
		List	Select Search Results	1
		Clear		3
	Display Editor	Display	Selected Elements/To Display	2

Figure 1

I also believe in using the “80/20 Principle” to determine what to best include in a BVT, and to determine what to test in an application in general. In his book on the subject, *The 80/20 Principle, Secrets of Achieving More with Less,* Koch [2] discusses how highly effective people recognize that in many different situations, 80 percent of results flow from just 20 percent of efforts. By recognizing this imbalance, you can maximize productivity by concentrating on the key causes, inputs or effort aimed at producing the results, outputs or rewards.

In applying this principle to software, most users of applications use only 20 percent of the features 80 percent of the time. Given this, it makes sense to determine what features are used 80 percent of the time, and make sure to include these features in the BVT. Also, in many software applications, 80 percent of the defects are found in about 20 percent of the features. It’s a good idea to routinely review the defect database to determine where the most defects are occurring, and concentrate testing resources (both automated and manual) on these areas.

Figure 2 shows a suggested process for developing and using a BVT.

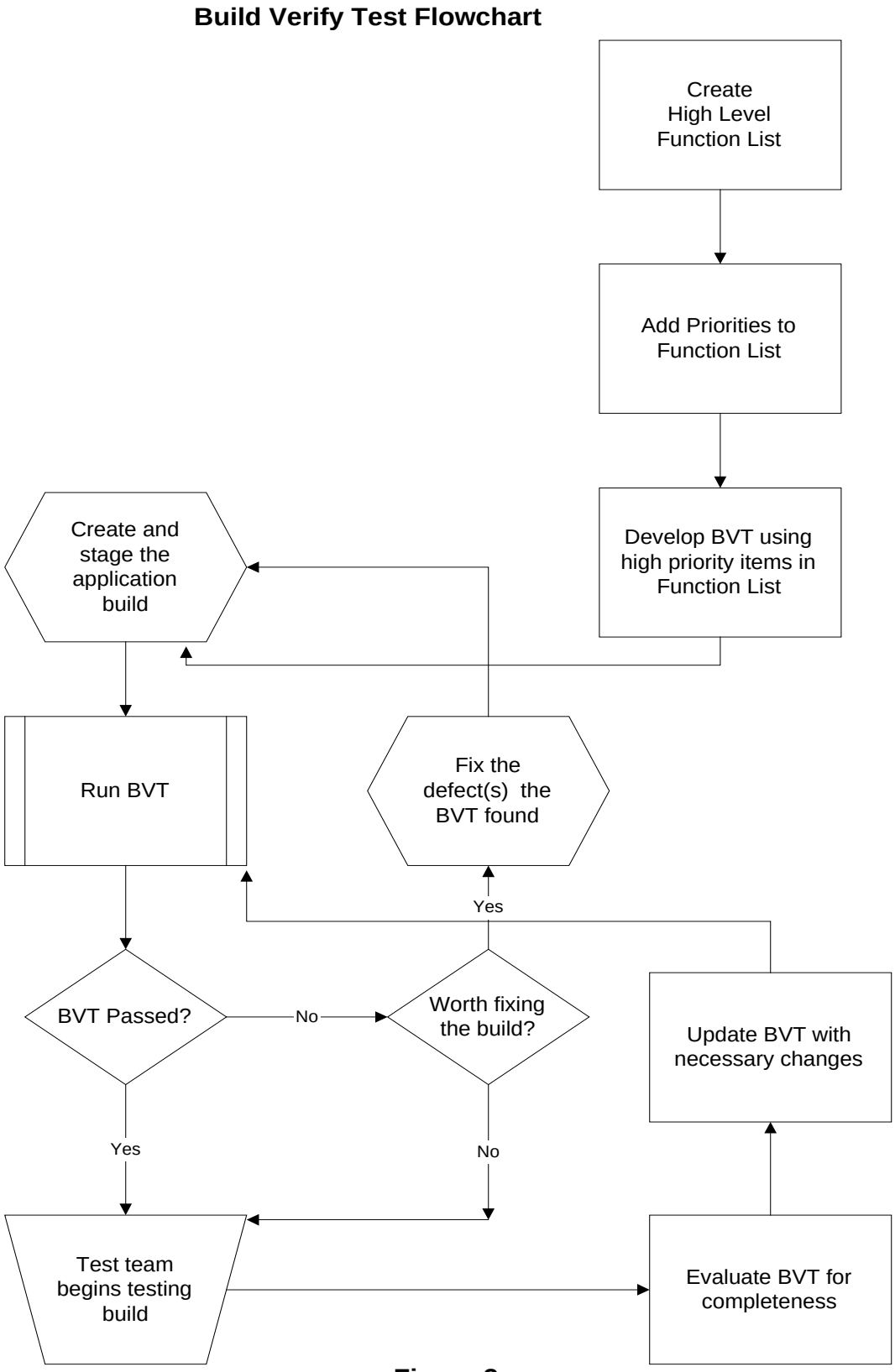


Figure 2

Tips when developing the BVT

- Design the BVT to run unattended, if possible.

- It should have a single Pass or Fail result at the end, based on whether all individual tests passed as well. This result should be written to a disk file and/or displayed on the screen to be reviewed after the BVT is complete.

- Be observant to finding possible defects when developing scripts for the BVT, since you need to verify a correct result for each test item. When you do find a defect, if it blocks other tests, then try to create a workaround or comment it out, and come back to it when it is fixed.

- Test automation developed for the BVT should be treated as code, and typical software development principles apply. Bach [3] reminds us that Testware is Software, and careful thought and structured design is required in creating the architecture.

- Centralizing all configuration parameters into one place is another of Bach's [3] suggestions that applies to the BVT. This can allow you to change various settings without having to recompile the script code, such as:

    - Where the application executable is launched from
    - What application usernames and passwords are, if applicable
    - Whether or not to log navigational steps and actions during the test
    - Which directories to read input from and write output to

- Develop reusable routines for commonly used actions, and place them at the highest appropriate level of include file, so they can be accessed globally or locally.

Determining when (and by whom) the BVT should be run

Once you have a BVT in place, the next step is to determine when to run it in the software development lifecycle. This will depend on how your Development, Configuration Management, and QA/Test teams are organized in your company. Depending on the company's organization, there are 3 different places where you can run the BVT:

By Whoever creates the builds

Depending on the organization, the application build process can be someone in the Development department, a separate Configuration Management (CM) role, or in the Test/QA department.

In this scenario, the BVT should be run immediately after the application is built, staged and test installed. An efficient technique I have seen used by Build personnel is to have another script or program read the BVT result automatically, and if a Pass result was obtained by the BVT, the script automatically kicks off a process to copy the build to a formal staging area and then notify Test personnel via email.

By a Tester as soon as build comes out

If the builds are done outside of the Test organization, once an application build becomes available for testing (ideally on a network location) a designated Test person can install the application, and start the BVT. Once the BVT finishes successfully, they can notify the Test team that the test cycle can begin.

By a Developer after checking in changes

If the BVT is designed flexibly enough (e.g. to run the application from a configurable directory location), each developer can also run the BVT after a significant code check in, to reinforce unit testing or to verify a special build. Even if the BVT is run at this stage, it should also be run after the formal build is completed to ensure testing is performed on an integrated version.

Checking BVT Results

If the BVT runs successfully to completion (a "Pass" result), notify the Test team either personally (or use the automatic technique described earlier) to begin testing the build. However, if the BVT does not reach a Pass result, proper diagnosis should be undertaken.

Analyzing Failures

When failures are found by the BVT, it's important that the BVT is designed flexibly enough that quick decisions can be made regarding the impact of the failure. There are generally 3 types of failures that may occur:

1. BVT Test Failures – Thi s type of failure occurs when an individual test is logged a "Fail", either by an expected result not matching an actual result, or by failure to produce a result at all. When this type of failure occurs, investigate the nature of the result.

In some cases, the "failure" may be simply a new expected actual result, such as a GUI change or updated functionality change. In this case, update the "Master" result to be the new expected result so that the test will pass the next run.

If you determine the failure appears to be an actual defect, try to reproduce the steps manually. If it's a reproducible defect of serious or fatal nature, or if it blocks other application functionality, you should assess and determine the impact to the testing schedule. If appropriate, contact Development personnel immediately, have the defect fixed ASAP, and run the BVT again.

2. Unexpected Application Errors – This is when the BVT script cannot continue because of an exception, such as a General Protection Fault (GPF) or an unexpected application dialog. Most modern automation tools offer some type of error recovery capability, such as Visual Test's Suite Manager's capability to look for windows that contain a particular caption [Arnold 4], or SQA Robot's method of sending an Escape key to attempt to dismiss unexpected dialogs. This can allow you to continue running the BVT without interruption, if possible.

3. Script Run-time Errors – This occurs when a script cannot run because of some condition during the running of the script not allowing it to continue, such as trying to delete a file that doesn't exist, or trying to access a non-existent array element. In this case, an error handler should be used to trap the error, and attempt to recover from the automation tool's failure to continue running a script. Arnold [4] discusses the importance of using flexible error handling early in script development, so that the error handling is consistent in all script development.

Maintaining Changes to the BVT

Once an initial BVT has been created and is in place, it should not be forgotten. The BVT should periodically be evaluated to ensure it's continuing to do the job. Keep the BVT updated as changes in the application occur, so that the BVT remains up-to-date and doesn't stop running. This may mean handling simple GUI changes, such as new or renamed controls, or functional changes, such as a new feature or function.

After high-priority defects are fixed and closed, sometimes it's a good idea to add the automated regression of these bugs to the BVT, so you can be confident they stay closed. This can simplify your regression testing a great deal.

Using the BVT for other uses

Once a basic BVT has been created, with a minimum of additional modifications, it can be extended to perform other automated testing functions, at a fraction of the cost of tools that are specifically designed to perform these tasks.

Stress Testing

By using a simple loop construct (e.g. FOR i = 1 to n…NEXT in Basic), and logging current memory conditions, you can make the BVT into a Stress Test. By running the BVT for several iterations such that it runs for long periods of time (e.g. overnight or over a weekend), you can determine whether your application is causing a memory leak. Through review of the BVT output log, you should be able to tell where the memory leak is coming from, and rerun a subset of the BVT (or additional test automation) to isolate the specific offending function.

Different Operating Systems track memory conditions using different measures. In Windows 3.x, variables available include Memory Load, Total Physical Free Mb, and Virtual Free Mb. In Windows 95, the Resource Meter enables you to track System Resources, User Resources, and GDI Resources. In Windows NT, the Task Manager keeps track of the Commit Charge.

Performance Testing

By adding Elapsed Time measurements to timing-critical application functions, the BVT can also be used for performance testing purposes. It can be used to compare the timing of running certain program functions across many different measurements:

- Different machine configurations (e.g., Pentium 133 vs. Pentium II 266)
- Different operating systems (e.g., Windows 95 vs. Windows NT)
- Incremental program versions (e.g., Version 1.1 vs. 1.2)

A Baseline for Regression Testing

If the BVT is the first piece of test automation developed, and was designed in a structured, modular way, you should be able to use it as a starting point for a fully automated regression test suite. Depending on the depth of the automated regression testing to be developed for the entire application, sometimes it makes sense to either continue adding more tests to the BVT to make it increasingly more comprehensive, or create a separate regression testing suite using the BVT as the initial baseline structure.

BVT Return on Investment

Now that you are convinced that a BVT is a good thing to do, what is the return on investment (ROI)? According to Kaner [5], the cost to create, verify, and minimally document an automated test is usually between 3 and 10 times what it costs to do it manually. In my experience, the cost of developing a BVT will vary, depending on such factors as application program complexity, automation tool(s) used, and skill of the personnel developing the test automation.

However, the cost of a BVT is offset by greater savings if manual testing is used for the same testing activities. Figure 3 shows a table with an example of the amount of time saved by having a BVT find a defect compared to manual testing methods. The example assumes the BVT finds a serious defect after 15 minutes of running the application.

Aggregate Hours Saved
if defect discovered by BVT vs. manual testing

Time until bug found	3 testers	7 testers	10 testers	25 testers
1 hour	2.2	5.2	7.5	18.8
4 hours	9.8	22.8	32.5	81.3
1 day	21.8	54.3	77.5	193.8
2 days	47.3	110.3	157.5	393.8

Figure 3

If we multiply the actual number of hours saved per build in the chart above by the number of builds that an application has during its life, you can see how quickly a BVT can save your organization time by early defect detection.

In my experience, I have found that the BVT finds an error worth fixing on average about every 3 to 5 builds, and although it's hard to predict how much time was saved based on the Test team size, I estimate 25-30 hours saved each time a defect was found.

In addition, there is an immeasurable "confidence factor" that takes place when the Test team knows that the BVT has passed successfully.

Summary

Development and implementation of an automated BVT is well worth the time spent to improve the QA and testing process, and provides a first indicator of application reliability once released to Test.

Conversely, projects without an effective BVT in place can easily have Test teams aggregately spend dozens (to potentially hundreds) of hours manually testing an application build, only to send it back to Development when a major defect is found, hurting the project schedule and negatively affecting morale.

You can effectively use the time saved with a BVT to perform deeper-level testing of the application, producing a better quality product.

References

[1] Kaner, Cem, Falk, Jack, and Nguyen, Hung Quoc. *Testing Computer Software, Second Edition*. Van Nostrand Reinhold, 1993.

[2] Koch, Richard. *The 80/20 Principle, Secrets of Achieving More with Less*. Doubleday, 1998.

[3] Bach, James. *Useful Features of a Test Automation System*. Software Testing Laboratories, 1996.

[4] Arnold, Thomas. *Software Testing with Visual Test 4.0*. IDG Books Worldwide, 1996.

[5] Kamer, Cem. *Improving the Maintainability of Automated Test Suites*. Presented at Quality Week, 1997.

Strategic Planning Process - How to Plan Improvement

Mary Sakry and Neil Potter
The Process Group
P.O. Box 700012
Dallas, TX 75370-0012
Tel. 972-418-9541
E-mail: help@processgroup.com
Web: http://www.processgroup.com

ABSTRACT

Effective process improvement must help an organization achieve its business goals. This strategic planning process ensures that there is a tie between improvement activities and these goals. Several companies have used this process to successfully plan their process improvement.

This workshop will describe the steps necessary to develop a strategic-level process improvement plan.

The starting point of an effective process improvement program is to conduct a formal or informal process assessment to determine the strengths and weaknesses of the organization. After the problem identification phase, a high-level (strategic) process improvement plan is created. This is followed by a detailed tactical action plan. This article describes the steps to create a strategic plan.

The overall flow of the strategic planning process is:

- Select business goal to focus on
- Select assessment findings (or problem areas) that support the business goal
- Develop subgoals for each problem area
- Prepare for tactical planning

KEY WORDS

process, improvement, goal, vision

BIOGRAPHICAL SKETCH - MARY SAKRY & NEIL POTTER

Mary Sakry is co-founder of The Process Group, a company that consults in software engineering process improvement. She has 23 years of experience in software development, project management and software process improvement. For 15 years she was a Project Manager and Software Engineer within Texas Instruments (T.I.) in Austin, TX. In 1989 she worked on the Corporate Software Engineering Process Group within T.I. to lead software process assessments across T.I. worldwide. The last two years of T.I. were spent consulting and educating software developers and managers on software project planning, risk management, estimation, inspection and subcontract management. Mary is an SEI authorized lead assessor for CBA-IPI process assessments. She has an M.B.A. Business Management from St. Edwards University and B.S. Computer Science from the University of Minnesota.

Neil Potter is co-founder of The Process Group, a company that consults in software engineering process improvement. He has 13 years of experience in software design, engineering and process management. For six years Neil was a Software Design Engineer in Texas Instruments, Dallas, developing Electronic Design Automation software. The last two years at T.I. he was a manager of a Software Engineering Process Group performing consulting within T.I. in America, England and India. Consulting included software project planning, risk management, estimation and inspection. Neil is an SEI authorized lead assessor for CBA-IPI process assessments. He has a B.Sc Computer Science from the University of Essex in England.

Strategic Planning Process - How to Plan Improvement

The starting point of an effective process improvement program is to conduct a formal or informal process assessment to determine the strengths and weaknesses of the organization. After the problem identification phase, a high-level (strategic)improvement plan is created. This is followed by a detailed tactical action plan.

The overall flow of the strategic planning process is:

- Select a business goal to focus on
- Select assessment findings (or problem areas) that support the business goal
- Develop subgoals for each problem area
- Prepare for tactical planning

In detail, the steps to develop a strategic plan are:

1. State the business goals of the organization being improved:

The business goals are used to determine which assessment finding to address first, and to keep the team focused on a clear result. If the improvement effort does not help the business it may fail due to a lack of management support. Using the business goals makes the tie clear.

If you don't have a clear picture of the business goals you might need to ask the managers of the organization.

2. Select one business goal as a focus:

Select an important organizational goal. Importance can be determined by using such criteria as: urgency, impact on the organization or the presence of significant weaknesses. For example you might choose to work on the following goals: Y2K compliance, reduced costs, and reduced time to market.

3. Select a strategic planning team:

Look at the business goal and problems you are trying to solve and choose people that have a vested interest. Include people, when appropriate, from management, development and marketing.

4. For the selected business goal:

a) If a process assessment has been conducted, determine which assessment findings (or parts of an assessment finding) would have the greatest impact on the goal

b) If a process assessment has not been conducted, brainstorm and discuss the problems that are preventing you from achieving the goal and set priorities.

5. For each weakness (assessment finding or problem)**:**

a) Each member of the planning team gives his/her interpretation of the weakness. The team validates its understanding and discusses and resolves any differences.

b) Determine **subgoals** (over the next 18-24 months):

» How should the organization look or behave when this weakness has been addressed? These subgoals (established for each weakness) should be in line with the overall business goal.

The creation of an image of how you want the organization to look after an assessment is the most important part of strategic planning. Initially you may come up with images that seem unbelievable or ones that you don't feel you have any control over. These will be refined in step "d".

A good analogy is road building. The business goal describes where the road ends. The subgoals, based on problem statements, state the pot holes that need to be filled in order to achieve the business goal.

c) State **why** this subgoal is important to achieve (the benefits)

It is important for improvement goals to be compelling. For each subgoal statement, state why the organization wants this to be achieved. This is the motive. If you cannot develop a compelling argument for achieving a subgoal you may have found an area not to waste time on for now.

d) Check that each subgoal is SMART: **S**imply stated and specific. **M**easurable, stated **A**s-if-now and achievable (within your control or influence)**. R**ealistic and believable. **T**imed (eventually with a date) and toward what you want (stated in the positive).

6. Set priorities for each subgoal statement:

It is important to focus on a few areas and not try to solve everything at once. Priorities can be set by determining the relative benefit and cost of each goal.

- On a 1-10 scale, rate the **relative benefit** the subgoal would have for the organization (1 = low, 5 = medium, 10 = high)
- On a 1-10 scale rate the **relative cost** of the subgoal to implement (1 = low, 5 = medium, 10 = high). "Cost" can be money or effort.
- Determine the **priority** (benefit / cost)
- Any interdependencies? Categorize into **phases**, 1st, 2nd, 3rd?

Look to see if there are any natural sequences for the subgoals and then sort the complete list by phase (use three phases for simplicity). For example, when working on planning, a subgoal in phase one may be to identify a project planning technique and pilot it. A subgoal in phase two may be to plan one real project and collect data. A subgoal in phase three would be to deploy the technique to all project teams. Start work on those subgoals in phase 1 with the highest priority.

7. Determine when the goal and progress should be reviewed:

If the goals are not frequently reviewed they are likely to get out of sight and out of mind, and action towards their accomplishment will fade. Frequent review will maintain a burning desire and consciousness for the goal so that the organization stays focused.

8. Do a preliminary brainstorm of tactical ideas:

It is useful to brainstorm ideas at this stage for the action plan. Not all the ideas will be used. If you have a process goal of the CMM or ISO9001, pick out items that help address the problems and achieve the subgoals. The quickest way to achieve any process model or standard is to find some practical use for each element. The selection criteria for where to start looking are the business goals and problem areas of the organization.

Summary

The intent of this process is to ensure that the improvement activities focus on the business goals of the organization. This is accomplished by starting with a business goal and then identifying the improvement steps required to achieve it.

Example - A Portion of a Strategic Plan

Business Goal: Time to market for product development is six months on 31 December 1998

#	Assessment Finding (problem statement)	Subgoal description (Antithesis of Problem Statement)	Why do we want to achieve this goal?	Relative benefit of goal (1-10)	Relative cost of goal (1-10)	Priority (Benefit/ Cost)	Phase 1,2,3
1.	Planning is ad hoc.	Planning is structured, i.e., a process is used.	Better planning will reduce rework, costly surprises and increase market share.	7	5	1.4	2
2.		New project leads are trained in planning.	Training will help avoid the same mistakes over time by keeping our skill level consistent.	6	7	0.9	2
3.	Estimation does not comprehend all the necessary tasks.	All necessary tasks are defined and factored into each estimate.	We must ship when we say we are going to with a product that works. We need credibility in schedule prediction.	8	2	4	1
4.		An estimation process for new development.	A process will make estimation more predictable and easier to learn. We can record and disseminate the best practices.	8	5	1.6	2
5.	Risk assessment is limited to known problems.	Risk assessment extends to unknown likely problems (true risks).	We are often surprised by problems that occur, we do not look ahead and see what is likely to occur. We need to manage our time better managing risk.	6	5	1.2	2
6.		Risk data are incorporated into the project schedule.	Risk data are easy to ignore and forget unless it is incorporated into the plan.	6	3	2	3

Bibliography

1. Robbins, Anthony., *Outcome, Purpose-driven, Action planning method.* See www.tonyrobbins.com.
2. James, Tad., *Creating Your Future,* Audio cassette series,1992.

THE PROCESS GROUP

Strategic Planning Process - How to Plan Improvement

The Process Group
P.O. Box 700012
Dallas, TX 75370-0012
Tel. 972-418-9541 • Fax. 972-618-6283
E-mail: help@processgroup.com
Web: http://www.processgroup.com

THE PROCESS GROUP

Focus on a Business Goal and Determine Related Weaknesses

1. **State the business goals of the organization** (or the project)
2. **Select one business goal as a focus**
3. **Select a strategic planning team**
   e.g., people that:
   - own the business goal
   - have expertise to help achieve the goal
   - have a vested interest in the goal

THE PROCESS GROUP

Business Goals + Assessment Findings

1998 Business Goals

- Customer satisfaction 4/5 on 31 December
- Development cycle time is 6 months on 31 December
- Employee satisfaction 4/5 on 31 December
- Profit is $100K/employee/year on 31 December

Planning Findings

- Planning is ad hoc
- Estimation does not comprehend all the necessary tasks
- Risk assessment is limited to known problems
- Schedule and functionality are preset before the project starts and there is no negotiation method
- No accounting method for all hours worked whether paid or not makes historical data flawed

THE PROCESS GROUP

Match Problems with Goals

4. For the selected business goal:

a) If a process assessment has been done, determine which **assessment findings** (or parts), when addressed, would have the greatest impact on the goal.

b) If a process assessment has not been done, brainstorm on the **problems** preventing you achieving the goal.

Set priorities.

THE PROCESS GROUP

Example

Business Goal:

Development cycle time is 6 months on 31 December 1998

Match problems with goal →

Assessment Finding (Problem statement)
Planning is ad hoc.
Estimation does not comprehend all the necessary tasks.

THE PROCESS GROUP

Establish Desired Outcome (subgoal)

5. **For each weakness** (assessment finding or problem)**:**
   - Each team member gives his/her interpretation of the finding, discusses and resolves differences.
   - Determine **subgoals** (over the next 18-24 months):
     - » How should the organization **look** when this finding has been addressed?
   - State **why** this goal is important to achieve.
   - Check each subgoal is SMART:
     - » **S**imply stated and specific. **M**easurable,
     - » stated **A**s-if-now and achievable (within your control or influence).
     - » **R**ealistic and believable.
     - » **T**imed (eventually with a date) and toward what you want (stated in the positive).

THE PROCESS GROUP

Example

Assessment Finding (Problem statement)	**Subgoal** (Antithesis of problem)	**Why do we want to achieve this goal?**
Planning is ad hoc.	Planning is structured, i.e., a process is used.	Better planning will reduce rework, costly surprises and increase market share.
	New project leads are trained in planning.	Training will help avoid the same mistakes over time by keeping our skill level consistent.
Estimation does not comprehend all the necessary tasks.	All necessary tasks are defined and factored into each estimate.	We must ship when we say we are going to with a product that works. We need credibility in schedule

THE PROCESS GROUP

Set Priorities for Each Subgoal

6. **Prioritize** each **subgoal** statement (1 = low, 10 = high):
   - » On a 1-10 scale, rate the relative **benefit** the subgoal would have for the organization
   - » On a 1-10 scale, rate the relative **cost/effort** of the subgoal to implement
   - » Determine **priority** (benefit / cost)
   - » Any interdependencies? Categorize into **phases**, 1st, 2nd, 3rd?

THE PROCESS GROUP

Example

Subgoal (Antithesis of problem statement)	**Relative benefit of goal** (1-10)	**Relative cost of goal** (1-10)	**Priority** (Benefit /Cost)	**Phase 1,2,3**
Planning is structured, i.e., a process is used.	7	5	1.4	2
New project leads are trained in planning.	6	7	0.9	2
All necessary tasks are defined and factored into each estimate.	8	2	4	1

THE PROCESS GROUP

Establish Review Points

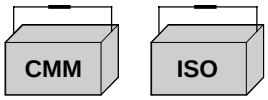
7. **Determine when the goal and progress should be reviewed**
   - Verify we are heading in the right direction
   - Review points would be added to the tactical (action) plan, for example:
     - » end of action plan creation
     - » after initial solution analysis
     - » after initial solution development
     - » after first pilot
     - » after each phase

THE PROCESS GROUP

Brainstorm Tactical Ideas

8. Do a preliminary brainstorm of tactical ideas

- For selected **subgoal** statements:
  - » Brainstorm **high-level tasks** to address finding and achieve subgoal
    - What is preventing us from achieving this subgoal?
    - Who can we model who has already achieved this?
    - What skills, resources and similar solutions exist that can be used?
    - What in the process standard / model being used can help us do this?



THE PROCESS GROUP

Example

Subgoal (Antithesis of problem statement)	**Priority (Benefit/ Cost)**	**Phase 1,2,3**
Planning is structured, i.e., a process is used.	1.4	2
New project leads are trained in planning.	0.9	2
All necessary tasks are defined and factored into each estimate.	4	1

Tactical Ideas

- **Interview project managers to get comprehensive task list**
- **Develop detailed checklist**
- **Develop tailoring guidelines for the checklist**
- **Inspect checklists**
- **Put checklists into scheduling tool**
- **Pilot on projects A+B**
- **Determine lessons learned**

THE PROCESS GROUP

Hand to Working Group

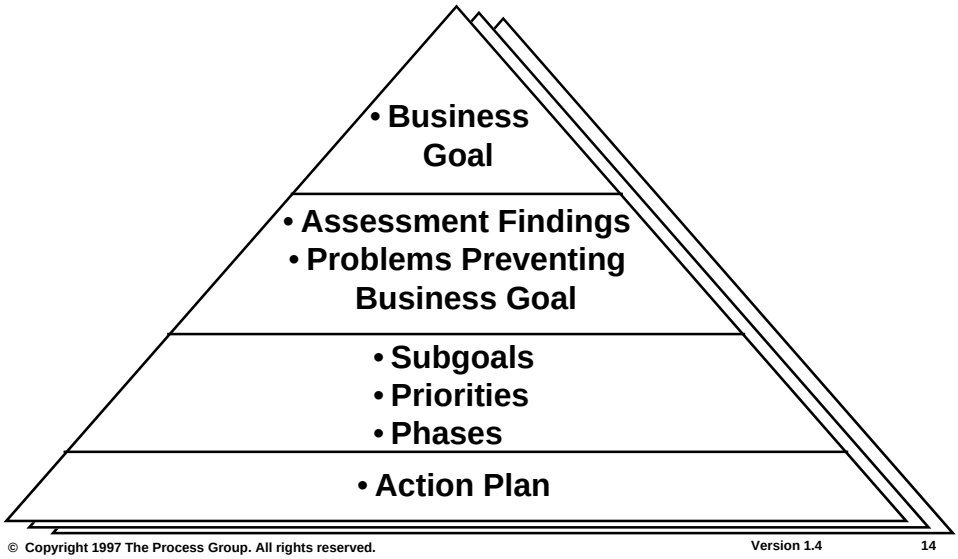
- Assign responsibility for the subgoal statement and hand preliminary tactical ideas to working group
- The tactical ideas will be reviewed, refined and put into an action plan

Assessment team, sponsor, guests...

Working group

THE PROCESS GROUP

Summary



THE
PROCESS
GROUP

References

1. **Robbins, Anthony., *Outcome, Purpose-driven, Action Planning Method.* See www.tonyrobbins.com.**
2. **James, Tad., *Creating Your Future.* Audio cassette series,1992.**

EXAMPLE 1
SOFTWARE PROCESS IMPROVEMENT
STRATEGIC PLANNING PROCESS

Business Goal:

Development cycle time is 6 months on 31 December 1998

#	**Assessment Finding** (Problem statement)	**Subgoal** (Antithesis of problem statement)	**Why do we want to achieve this goal?**	**Relative benefit of goal** (1-10)	**Relative cost of goal** (1-10)	**Priority** (Benefit/ Cost)	**Phase 1,2,3**
1.	Software project planning is ad hoc.	Planning is structured, i.e., a process is used.	Better planning will reduce rework, costly surprises and increase market share.	7	5	1.4	2
2.		New project leads are trained in planning.	Training will help avoid the same mistakes over time by keeping our skill level consistent.	6	7	0.9	2
3.	Software estimation does not comprehend all the necessary tasks.	All necessary tasks are defined and factored into each estimate.	We must ship when we say we are going to with a product that works. We need credibility in schedule prediction.	8	2	4	1
4.		An estimation process is used for new development.	A process will make estimation more predictable and easier to learn. We can record and disseminate the best practices.	8	5	1.6	2

5.	Risk assessment is limited to known problems.	Risk assessment extends to unknown likely problems (true risks).	We are often surprised by problems that occur, we do not look ahead and see what is likely to occur. We need to manage our time better managing risk.	6	5	1.2	2
6.		Risk data is incorporated into the project plan and schedule.	Risk data is easy to ignore and forget unless it is incorporated into the plan	6	3	2	3
7.	Schedule and functionality are preset before the project starts and there is no negotiation method.	Schedules are variable before the project starts and there is a method of negotiating.	We need to get products to the market place when we say we will. Fixing this is the source of our problems.	8	8	1	2
8.		Functionality is variable before the project starts and there is a method of negotiating.	To avoid the situation of getting locked into a specification that is not a true reflection of what the customer wants.	8	6	1.3	1
9.	No accounting method for all hours worked whether paid or not makes historical data flawed.	There is a method of accounting for all hours worked whether paid or not. Historical data in the database includes uncompensated overtime.	Our estimates are always wrong. We cannot predict how much overtime will be needed, and the organization gets burnt out.	6	5	1.2	3

EXAMPLE 1 (continued)
SOFTWARE PROCESS IMPROVEMENT
TACTICAL PLAN FOR 2 GOALS

#	**Assessment Finding** (problem statement)	**Goal**	**Why**	**Actions**	**Pri**	**Phase**	**Cost or Time**	**Who**	**Date**
3.	Software estimation does not comprehend all the necessary tasks.	All necessary tasks are defined and factored into each estimate.	Credibility in schedule prediction.	Select 2 pilot projects.	1	1			
				Brainstorm all tasks for projects in 1995.	2	1			
				Produce task list template.	3	1			
				Inspect template.	4	1			
				Determine special cases where tasks are not required in overall schedule.	5	1			
				Produce 2-page document with generic list and guidelines.	6	1			
				Pilot in 2 estimation sessions.	7	1			
				Revise solution from feedback.	8	1			
				Present results so far to organization.	9	1			
				Add to estimation process kickoff meeting.	1	2			
				Develop training class section for project planning class.	2	2			
				Review class solution with organization.					
				Use checklist with 2 additional projects.	1	3			
				Develop policy.	2	3			

4.		An estimation process for new developme nt.	Estimatio n with more predict-ability.	Select 2 projects to pilot.	1	1			
				Read Boehm+SEI refs. for processes.	2	1			
				Select 2 solutions to evaluate for 4 weeks with both pilots.	3	1			
				Run pilot sessions	4	1			
				Communicate results to PI steering committee.	5	1			
				Replan deployment.	5	1			
				Make/buy training.	1	2			
				Select 3 projects to deploy solution.	2	2			
				Lessons learned + revise deployment plan.	3	2			
				Establish policy for when process is to be used.	1	3			
				Pilot policy for 2 months.	2	3			
				Lessons learned + revise deployment plan for policy.	4	3			

EXAMPLE 2
SOFTWARE PROCESS IMPROVEMENT
STRATEGIC PLANNING PROCESS

Business Goal:

31 December 1998: Products contain zero major defects at release time.

#	**Assessment Finding** (Problem statement)	**Subgoal** (Antithesis of problem statement)	**Why do we want to achieve this goal?**	**Relative benefit of goal** (1-10)	**Relative cost of goal** (1-10)	**Priority** (Benefit/ Cost)	**Phase 1,2,3**
	Doc. of guidelines and criteria for process tailoring of the org standard process needs more detail.	Process tailoring is such that PL can define project processes within 5% of project time, verified and enforced.	Increase productivity of the PL. Increase odds of success. Increase profitability.	6	2	3	2
		Improvements to standard processes are incorporated in projects within a month where applicable.	Projects benefit quicker.	4	6	0.7	3
		PL accepts the impact of his/her tailoring decisions. Project process meets project goals.	Improved awareness and management of risks.	8	3	2.7	2
		PL tailors project processes.	Increase ownership and accountability.	5	4	1.3	2
		Mandatory steps are explicit.	Reduced risk.	5	2	2.5	1

		Make the already defined process definitions understandable in 2 readings/5 days.	Easy training. Increase odds of use. Higher acceptability.	7	5	1.4	2
		Process steps used, omitted and risks and impact are visible.	Management comfort and reporting. Learning mechanism. Organizational productivity. Process improvement.	5	3	1.67	2
		Project can reuse processes of other projects.	Increase PL productivity. Increase overall comfort level.	7	6	1.2	3
		Build tailoring guidelines.	Ease of use.	8	6	1.3	1
	Training on guidelines and tailoring of org. processes should be formalized.	Effective training material and program (e.g., case study) to meet goal 1). 95% of PL's will use these guidelines.	Easy training, Increase odds of use. Higher acceptability.	7	7	1	2
	Current online document-ation system should be strengthened with improved search and retrieval facility.	Define/formali ze learning and feedback mechanism.	Organizational productivity and quality improvement.	7	7	1	3

EXAMPLE 3
SOFTWARE PROCESS IMPROVEMENT
STRATEGIC PLANNING PROCESS
(Client supplied format)

Business Goal:

31 December 1998: SEI Level 2.

ID	Goal	KPA Goal X-Ref	Why do we want to achieve this goal?	Relative benefit of goal	Relative cost of goal	Priority (Benefit /Cost)	Phase 1,2,3
	REOUIREMENTS MANAGEMENT						
RM-A	Requirements are documented, agreed to (by producing & receiving groups) and baselined per a defined process. • Baselined requirements are used as the basis for plans and commitments • Changes to baselined requirements are controlled to allow all groups within the organization to assess the full impact of the change before related commitments are revised.	RM 1	Documented requirements provide: • the initial point of synchronization within the organization on the scope of the project or program. • an established baseline against which ABC INC. can identify, quantify, and manage changes.	10	3	3.3	1
RM-B	Customer input & feedback activities are part of the requirements acquisition process.		Improves our ability to satisfy customers: • ABC INC. understands the needs of customers better. • Misunderstandings between ABC INC. & our customers are reduced. • Communication with customers is increased.	8	4	2	1

RM-C	A common understanding of requirements: >between ABC INC. and customers >flow & decomposition throughout ABC INC.. • The organization understands the level of requirements needed by each group within the organization to plan their work and to develop and deliver product. • The organization understands how and when in the life cycle each level of requirements is acquired or derived.	RM 2	Throughout the life cycle, requirements (as they are decomposed) form a basis for estimating, planning, performing, and tracking the work of the various participating groups.	10	6	1.5	1
RM-D	Demonstrate traceability of requirements from product objectives through system, HW & SW requirements specifications into and through all relevant work products. Requirements sources: • Product objectives • Customer Specified	RM 2	Traceability through appropriate work products provides checkpoints that enable us to ensure completeness of what is being "built" and to verify that what the customer asks for is what they receive	8	7	1	2
	SW PROJECT PLANNING; SW TRACKING & OVERSIGHT						
SPP-A	Scope and content is clearly understood & communicated to all affected entities (internal & external) as part of program & project planning	SPP G2& 3	Customer satisfaction is more likely if expectations are realistic.	8	4	2	1
SPP-B	Each project or program uses a defined & documented software planning process.	SSP1	Provides organization with a common understanding of what is included or is not included in the planning of each project or program	10	3	3.3	1

SPP-C	Planning addresses areas such as: • Risks • SQA • SCM • Training • Maintenance • External Dependencies Plans also include tasks and milestones (at specific points in the project/program life cycle) for re-evaluating initial assumptions, scope, risks, etc., and for re-estimating the impact of changes or deviations from initial plans to overall project or program commitments Project/program schedules reflect these tasks & milestones	SSP	When schedules are more realistic and accurate, schedule and cost overruns can be minimized or eliminated.	10	2	5	1
SPP-D	Software estimates are documented and used in planning & tracking the project NOTE: PIT combined this with SPP-F	SSP 1	•	10	1	10	1
SPP-F	Project plans and schedules also include activities (& estimates) for: • Requirements gathering; • SW engineering support of IT, tech pubs, customer support, account teams; • Support provided by the Development organization for proposals, product mgt, etc. • Maintenance & release support; • Training; • Project or program post-mortems		More realistic and accurate schedules Reduces over allocation of key resources. • Provides a more objective basis for measuring progress. • Estimates & actuals can be used in planning future projects	10	3	3.3	1

SPP-E	All internal groups participating in a program or project are involved in the commitment and planning activities and phases.		Results in more realistic schedules. When schedules are more realistic and accurate, schedule and cost overruns can be minimized or eliminated.	8	2	4	1
SPP-G	The skill set & organizational responsibilities of each resource are used along with estimates of work to be done to derive more realistic schedules	SSP1	Results in more realistic schedules. When schedules are more realistic and accurate, schedule and cost overruns can be minimized or eliminated.	6	6	1	1
SPP-H	People understand the full scope of their assignments before agreeing to them	PTO 3	When people and their managers have the same understanding of what is being committed to, we increase the probability of successfully meeting commitments	8	2	4	1
SPP-I	When initial commitments are communicated (internally and externally), they are always accompanied with a clearly defined and well understood statement of scope and the assumptions upon which the commitment is based.	SPP G2	Enables ABC INC. to determine & communicate when commitments change thus enabling us to maintain realistic customer expectations	8	2	4	1
SPP-J	(Internal) Realistic negotiation of scope, features and dates occurs as part of the commitment process	SPP G2& 3	Re-negotiation enables an organization to fully understand the impact of changes from the technical, program/project management and business aspects	8	4	2	2
SPP-K	(Internal) Changes to requirements or to initial program or project assumptions are used as a basis for renegotiation of program or project plans and schedules. • Replanning occurs when these changes are detected	PTO 1&2	Establish a plan to manage and track the project against. Have the ability to determine current status of projects.	8	2	4	2

SPP-L	Organization has integrated view of all current project plans. Project & program planning includes • synchronization with customer plans • synchronization with all other ABC INC. product plans and roadmaps	SPP G2&3	• Reduces overcommitment of resources • Reduces schedule & cost overruns • Increases probability of meeting commitments	8	3	2.6	2
SPP-M	(External) Realistic negotiation of scope, features and dates occurs as part of the commitment process	SPP G2&3	Re-negotiation enables an organization to fully understand the impact of changes from the technical, program/project management and business aspects	6	6	1	3
SPP-N	(External) Changes to requirements or to initial program or project assumptions are used as a basis for renegotiation of program or project plans and schedules. • Replanning occurs when these changes are detected	PTO 1&2	Establish a plan to manage and track the project against. Have the ability to determine current status of projects.	6	4	1.5	3
SPP-O	(External) The organization also plans for concurrence & synchronization with customers & their plans		Improved customer relations	8	4	2	3
SPP-P	Plans that are produced are actually used by the producing organization to manage their group's work.	SSP & PTO	One purpose of a plan is to guide the execution of a project or program. Plans should be the primary tools that the organization uses for tracking progress (and changes).	10	2	5	2
SPP-Q	Software managers understand and manage critical path activities	PTO 1	Enables the organization to know when we're approaching or in trouble	8	3	2.6	2

SPP-S	Organization re-evaluates initial commitments or plans on a periodic basis or as soon as a deviation from plan is detected. (ie., Org culture: Red flags, raised as soon as known, are good!)	PTO 2	The earlier a red flag is raised, the sooner it can be addressed and the potentially negative impacts can be minimized.	8	2	4	2

A Phased Approach to the PSP

Jeffrey S. Holmes
Motorola Cellular Infrastructure Group
Fort Worth, Texas 76137

Bonnie E. Melhart
Texas Christian University
Fort Worth, Texas 76129

Abstract

Completing the exercises for learning Humphrey's PSP is a monumental task. After the developer has reached this milestone, what next? Implementing the PSP into an industrial setting can be very frustrating. This paper suggests ways to reduce the amount of frustration experienced by developers as they apply the practices of the PSP in their daily work activities. A process for introducing the PSP into a developer's work and results from using this process are described.

Author's Biographical Sketches

Mr. Holmes has worked as a software engineer for 10 years. His previous affiliations include General Dynamics, Howell Instruments, and Canmax Retail Systems. He is currently part of the Motorola Cellular Infrastructure Group working on embedded software for CDMA cellular infrastructure equipment. He has served as an internal CMM assessor and is trained to perform Motorola assessment. He has earned a B.S. in Computer Science from Texas A&M University in 1988 and a Master of Software Engineering from TCU in 1998.

Dr. Melhart is an Associate Professor of Computer Science at TCU. She has been a consultant to the Office of Personnel Management of DoD, Motorola, and Systems World, Inc. in various software engineering capacities. Her areas of research include Software Quality Assurance, Software Safety and Reliability, and Computer-Based Systems Engineering. She has earned the B.A. and M.S. degrees in Mathematics and the M.S. and Ph.D. degrees in Information and Computer Science. Dr. Melhart belongs to Sigma Xi, ACM, and IEEE, of which she is a senior member. She serves as vice-chair of the Technical Committee on Engineering of Computer-Based Systems of the IEEE Computer Society.

1 Introduction

The software development industry is inundated with demands for higher quality products in record setting time. Companies are pursuing many avenues in an attempt to compete in this high pressure market. A substantial portion of the effort to improve software project quality and timeliness has focused on the organizational software process. The most notable standard for organizational process is the Software Engineering Institute's (SEI) Capability Maturity Model (CMM)[1]. The CMM provides a framework for an organization to create defined and repeatable steps to develop software. Once the organization has defined the software process, the organization is constantly evaluating and making improvements to the process.

An element that seems almost forgotten in the organizational software effort is the software developers themselves. For companies to continue to build on their organization process successes and advance to the upper SEI levels, the organization will depend on the software developers to carry out the organization's processes. Without disciplined software developers to execute and improve upon the company's processes, the organization will not achieve the full benefits of the defined software process. The Personal Software Process (PSP), developed by Watts Humphrey, provides a method to address the software process improvement effort at the software developers' level [2]. The PSP is not an alternative to the CMM; rather it is a mechanism to achieve the desired maturity level.

The use of PSP practices within a software development organization is reminiscent of manufacturing practices employed in Japanese assembly lines. Instead of adding inspectors to their assembly lines, the Japanese implemented techniques to improve the quality of work produced by the assembly line workers. In a similar manner, the PSP reduces the need for additional testers in the organization, as the software developers themselves learn where defects occur and take steps to eliminate further defects.

The organization's software efforts initially define the software process. For the software process efforts to actually benefit the company, the software developers must execute the process. The largest hurdle in establishing the software process may be convincing the developers that the activities are worthwhile. The practices of the PSP demonstrate to the individual developers the benefits of following a defined software development process.

The PSP strives to instill a set of disciplines into the developer's work habits that will make the developer more effective. In particular, work practices are targeted to improve time management, software quality, project estimations, and project planning. Example disciplines are:

- Recording time spent on each phase of the project.
- Recording project code size.
- Recording defect data.
- Performing time estimates based on historical data.
- Performing size estimates based on historical data.
- Performing defect estimates based on historical data.
- Evaluating process to determine areas for improvement.

With the PSP disciplines incorporated into the developer's daily work activities, the organization will benefit from complete and accurate metrics, and accurate project estimations. In addition, the PSP practices reinforce the use of software engineering techniques such as inspections, defect prevention, and software reuse.

This paper promotes a process for incorporating the practices of the PSP into a software development organization. The process was developed while utilizing the PSP practices to develop software for Motorola's Cellular Infrastructure Group in Fort Worth, Texas. The paper also includes a proposed method for training an organization's software developers on the PSP. In addition, the paper identifies future PSP activities including the status of a CASE tool to support the PSP.

2 Putting the PSP into Practice.

Humphrey has developed a set of exercises to train a developer in the PSP. The completion of the PSP exercises provides valuable experience for the software developer; however, the use of PSP practices in industry provides the real test. The introduction of PSP activities into an industry work environment can be very difficult. Real experiences are described here in an attempt to ease the transition to a disciplined personal process. This section will provide an overview of the work environment and projects to which the PSP was applied, a description of the PSP implementation process being developed, and the results of using the PSP on actual software development activities.

The PSP experiences related here occurred at the Motorola Cellular Infrastructure Group (CIG) in Fort Worth, Texas. The facility was self-assessed at SEI CMM Level 2 in April 1996 and SEI CMM Level 3 in December of 1996. This complies with Humphrey's recommendation that prior to performing the PSP, the organization must be at least SEI CMM Level 2 [2]. Five real-time embedded projects and one non-embedded project using the C language were completed. Only one developer on the projects was following the PSP practices.

The initial effort to introduce the PSP into work habits attempted to employ most of the practices at one time. This proved to be very frustrating due to the unfamiliarity with the organizational software process that was being followed on the project and with the product being developed. The project schedule was extremely aggressive and, by adding multiple other tasks, it was very difficult to actually perform the PSP tasks. This initial frustration inspired the phased PSP approach discussed in this paper.

2.1 Phased PSP Introduction Process

PSP practices were phased into the six projects, not unlike the way the PSP is learned. Table 1 shows the progression of PSP practices implemented in the projects. The introduction process is a guideline for institutionalizing the PSP into the work habits of developers who are members of a multi-person development team. The introduction process attempts to phase-in the PSP practices over time, with minimal interference to the project. The length of time that is required to complete the introduction of the PSP depends on the size of the projects. Because of the variance in projects, it could conceivably take years to fully implement the PSP into all work habits.

The PSP Introduction Process has four phases in which the PSP activities are introduced. New PSP activities are added at each phase until all of the PSP practices are being followed in the fourth phase. Phase 1 allows the individual to get in the habit of recording the project time and defect information, while performing their established development process. Phase 2 adds the use of the size estimations in a planning phase as described by the PSP. Phase 3 incorporates the use of task and schedule planning. Phase 4 is the optimization phase where the PSP data is continuously analyzed and process improvements steps are taken.

The introduction of the PSP can be very frustrating to developers. Experienced developers have established habits of performing their jobs. The incorporation of PSP activities disrupts the routine with some very tedious new activities. By implementing PSP in phases, the disruption to the developer's work habits is minimized. The phased introduction of the PSP allows the developer to gradually incorporate the PSP practices into their daily routine and, hopefully, increases the probability that the PSP introduction will be successful.

The development of a database of historical data is the baseline for the PSP estimation techniques. The phasing in of the PSP practices postpones the requirements for size, time, and defect estimates until a baseline of historical data has been created. This practice allows the developer's initial estimates to have better accuracy than if the estimates were based solely on the data accumulated from the PSP exercises.

A word about size measurements is in order, since these are given in lines of code (LOC), a measurement open to interpretation. As Humphrey suggests, the LOC measurement is typically needed to deal with a program's packaging, to measure, evaluate, or predict work progress, or to assess the program's quality. The latter is the main justification for its use in the PSP [2]. LOC measurements used here are for new, changed, deleted, or modified code of the projects described.

Table 1: PSP Implementation Process

Phase No.	Purpose	To guide in implementing PSP
	Entry Criteria	• Have completed minimum 10 exercises • Have a coding standard • Have new project to begin on • Have tools to collect data (optional)
1	Data Recording (PSP0 and PSP0.1)	• Use development life-cycle phases established by exercise 10A. • Religiously record time spent on project and non-project tasks. • Record defect information (injected/removed) • Record size of project when completed. • Determine process optimizations and record. • Complete 2-3 projects in this phase (At least 2 projects of 250+ LOC each)
2	Estimation (PSP1.1)	• At start of new project, calculate estimated size, time and defects using data from Phase 1 projects and exercises 8A-10A. • Continue recording data as in Phase 1. • Perform steps of PSP1.1. • Determine process optimizations and record. • Complete 2 projects in this phase.
3	Planning (PSP1.1 and PSP2.0)	• At start of new project, calculate size, time, and defect estimates using data from Phase 1 and Phase 2 projects. • Perform PSP2.0. • Do personal reviews prior to compiling code and performing peer reviews. • Continue recording data. • Roll up data. • Determine process optimizations and record. • Complete 2 projects in this phase.
4	Optimization (PSP2.1 and PSP3.0)	• Continue learned practices from previous phases. • Add COQ calculations. (PSP2.1) • Modify process as appropriate based on data and experience.

2.2 Results from Phasing in the PSP

This section details personal experience with phasing the PSP into a software developer's work habits. The following paragraphs describe experiences in PSP implementation during each of the phases. A total of six projects have been completed thus far and the first three phases of the implementation process have been completed. For completeness, a discussion of phase 4 is also provided.

Phase 1 - Data Recording

The first project was an enhancement to existing code on an unfamiliar system. The base code size of the project was approximately 500 KLOC of C code. The size of the team involved in the first enhancement was 7 developers. The project entailed 20 KLOC of changes, with data recording practiced on 5370 LOC. This project lasted for 10 calendar months. The project was characterized by the typical project planning shortfalls. The initial size and time estimates were, at best, very poor guesses. During the design phases, the project underwent two major re-architectures. These factors played a large part in the decision to only implement PSP data recording activities on the first project.

The second project was much smaller and was performed by three developers. Project two was 1500 LOC total with the PSP used for a part that involved 670 LOC. The project lasted 2 calendar months. Although not large, the project was required to meet a tight development schedule. Thus, the practice of only recording PSP data was continued. The data for projects 1 and 2 is summarized in Table 2.

Table 2: Projects 1 and 2 Results

Project	Size (LOC)		Time (Hours)		Defects	
	Estimated	Actual	Estimated	Actual	Estimated	Actual
1	N/A	5370	N/A	866	N/A	574
2	N/A	670	N/A	188	N/A	40

Time data was recorded using an in-house time tracking tool. The tool allows the developer to select the current activity phase and a timer counts the minutes spent in the activity. The use of this tool proved to be very convenient. It eliminated the need for tracking the information on paper and provided an easy venue for compiling the time data. Code size was measured using another in-house tool. The in-house tool was chosen over the line counting tool developed for Humphrey's exercise 2A (see [2]) due to the enhanced capabilities of the in-house tool.

The defect data that was recorded on the project was extracted from the department's defect tracking database. The defect data included defects detected in peer reviews, as well as in testing. The detailed defect tracking system of the PSP was not employed during this phase. The organization's defect tracking database was thought to be sufficient for recording defect information. Experience has revealed that the PSP defect tracking system provides a level of granularity not achieved using the organization's defect system. The recommendation is to follow the PSP defect tracking system detailed in [2].

Phase 2 - Estimation

Projects 3 and 4 were the first to utilize the PSP estimation techniques. As the recording of work time became ingrained as a work habit, the individual process was modified to more closely resemble the PSP phases. The detailed PSP defect tracking practices were incorporated for these projects. In addition, these two projects added the size and defect estimation techniques of the PSP. Because the discipline of recording project data was already institutionalized, this was a natural extension to the process. There was a strong desire to add these steps just to utilize the data that had previously been collected.

Projects 3 and 4 involved modifications to an existing system. The size of the base system was approximately 56 KLOC. The data in Table 3 shows the actual versus estimated values for projects 3 and 4. The accuracy of estimates for projects 3 and 4 can be credited to the size of the projects. These project sizes are comparable to the exercises performed when learning the PSP. The similar sizes enhanced the PROBE estimations ([2]) and resulted in a very nice set of estimates. The significant difference between estimated and actual defects for project 4 is attributed to the simplicity of the project. The project involved adding a software alarm to the existing software.

Project 4 took longer than expected because of extra time spent in design and compile. Although the infrastructure for adding the alarm was in place, the design strove to produce a generic implementation that would allow easy modification in the future. The compile time took 1.5 hours longer than projected due the unfamiliarity with the project's build environment.

Table 3: Projects 3 and 4 Results

Project	Size (LOC)		Time (Hours)		Defects	
	Estimated	Actual	Estimated	Actual	Estimated	Actual
3	219	213	42.7	26.9	24	28
4	70	89	13.2	15.6	19	7

Phase 3 - Planning

Project 5 was new development for a next generation product. A total of 1460 LOC were created to implement the assigned functionality for the new product. The personal process used on this project was extended to implement rigid personal code reviews, and task and schedule planning. Once again, the historical data collected on the previous projects provided incentive to further optimize the process. Previous data showed the number of defects that had been caught by peer review and thus influenced the decision to increase the effort spent on personal code reviews. In addition, there was a strong desire to apply the time estimates for project completion to calendar time. Unfortunately, the task and schedule planning revealed a weakness in task scheduling. The estimation of project time was fairly accurate; applying this time to calendar dates proved to be very inaccurate. The revelation that the PSP provided was that the project estimates were accurate, but tasks outside of this project caused the delay in meeting the calendar date. Project 5 data is shown in Table 4.

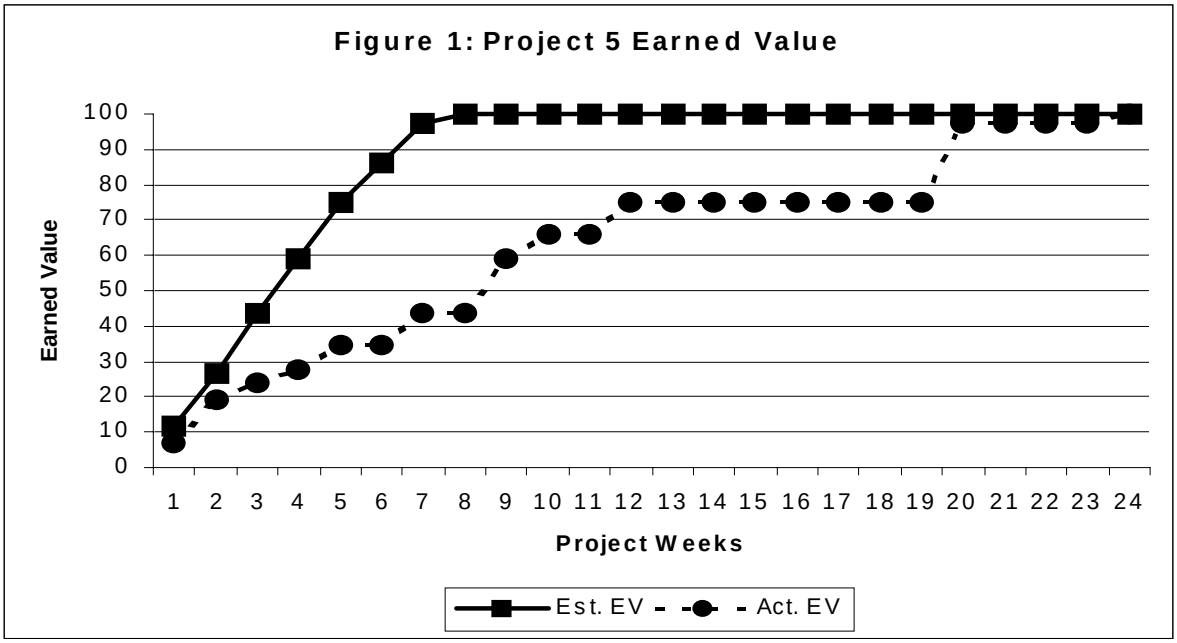
Table 4: Project 5 Results

Project	Size (LOC)		Time (Hours)		Defects	
	Estimated	Actual	Estimated	Actual	Estimated	Actual
5	820	1460	164	329.5	94	157

Earned value tracking was used to evaluate the project status throughout the development life cycle. This system of tracking progress on a project uses a common scale to judge the percentage of the project that is complete. As subtasks are completed, their predetermined contribution to the total effort is added to the earned value [2]. In our projects, each task was assigned an earned value by calculating the percentage of total time spent on the task in previous projects. The earned value tracking revealed that the project schedule was in jeopardy after three weeks. There was hesitation to report this information due partially to a lack of confidence in the data and a tendency to revert back to old habits to catch up by working harder and longer hours.

Project 5 experienced a schedule slippage of 16 weeks from the PSP forecasted completion data. The slippage was due in part to 5 weeks being diverted to other project tasks and 2 weeks because of test system availability. This diversion is illustrated in Figure 1 between weeks 12 and 19. Even if the identified 7 weeks were not factored into the project schedule, the project would still have been approximately 5 weeks late due to the low project size estimate.

The accuracy in estimation for projects 3 and 4 should have influenced the use of the cyclic format of PSP. In Humphrey's PSP exercise 3.0, projects are broken into "cycles" of approximately 250 LOC [2]. This practice was not followed for project 5. The use of cycles might have improved the code size, project time, and defect estimates.



On this 5th project, a dedicated effort was also made to perform thorough personal code reviews prior to allowing the code to go through the peer review process. (The PSP practice of performing a personal code review prior to compiling the code was not followed, however.) A comparison of the data on peer reviews performed prior to incorporating personal code reviews and data on peer reviews performed on code that had undergone personal code review revealed an interesting trend. The peer review process consists of classifying defects as major or minor. Major defects are those that cause undesirable results from the software. Minors consist largely of style and cosmetic defects. The ratio of major to minor defects was cut in half after incorporating personal code reviews. Prior to personal code inspections, for every 1 major defect found there were 4 minor defects recorded. After using personal code inspections, for every 1 major found there were 2 minor defects identified.

Table 5: Defect Fix Times

Phase	Total Defects	Average Time to Fix (minutes)
Design Review	45	1.4
Code Review	44	2.7
Compile	41	6.4
Unit Test	15	12.6

System Test	12	33.8

Personal reviews proved extremely valuable during project 5. Defect data recorded for this project presents a strong case for the value of finding defects during personal reviews. The fix time for defects found during the project are shown in table 5. As the data shows, defects caught in personal reviews are fixed in a dramatically smaller amount of time. The shorter fix times and the change in defect ratio provide substantial evidence of the value of personal reviews. At the least, the use of personal reviews should be incorporated as a courtesy to the other members of the development team that serve in the peer reviews. With fewer minor defects, the inspectors can concentrate on finding the major defects in the code.

Project 6 differed from the previous five projects. This project involved designing a stand-alone utility that could be invoked from the command line. There were several significant things experienced during this project. The most important involved the use of personal code reviews. One goal during this project was to utilize personal code reviews as described in [2], i.e., reviews performed prior to compiling the code. Of the 60 defects found during project 6, 34 (56%) were found prior to compiling the code. Data for Project 6 is shown in Table 6.

Table 6: Project 6 Results

Project	Size (LOC)		Time (Hours)		Defects	
	Estimated	Actual	Estimated	Actual	Estimated	Actual
6	139	204	25.7	16.2	44	60

Another item of significance involved the coding rate. The actual coding rate (12.6 LOC/hour) was double the projected coding rate (5.4 LOC/hour) for project 6. First inclination is to attribute this obvious improvement in productivity to reuse or developer experience. But these factors were minimized because there was very little reuse in project 6, and the application was of a unique nature. Rather, the most important factor affecting this productivity was the requirements. The requirements for the utility were easily understood and were not changed. This eliminated the costly practice of reworking design and code due to a requirement's change.

The projected completion date for project 6 was met, an improvement over project 5, but the small time needed for the project does not permit confidence in estimating actual project completion dates. As more projects are completed, confidence will grow.

Phase 4 - Optimization

The next software development project undertaken will be in phase 4 of the implementation process. During this project, an anticipation of renewed confidence in all of the data should provide incentive to use the data in reporting status to the project lead.

The optimization phase hinges on the successful implementation of the other phases. The entrance criteria for phase 4 include incorporation of all aspects of the PSP into previous projects. As the developer's work habits become more disciplined and the amount of data collected grows, the weak areas in the developer's process will become apparent. The process will then be altered to compensate for these weak areas.

Using the data collected from the previous 6 projects, several areas of emphasis have been identified. The use of personal code reviews will be continued and utilized before compiling the code. For larger projects, the cyclic methodology will be employed. The cyclic strategy allows for development in cycles that include the detailed design, test, implementation, and testing of small parts of the product at once. New functions are built and tested on the solid foundation of previous ones in earlier cycles [2].

During the optimization phase, concerns such as reuse and defect prevention can be addressed at the personal level. The data compiled on the previous projects provides the mechanism for addressing these issues. The historical data will also be used to generate control limits for the different phases of the project. Phase postmortems will be employed to examine the data collected during each phase versus the established control limits.

The optimization phase takes Quantitative Process Management to the developer level. The developer analyzes historical project data and makes personal process adjustments accordingly. Likewise, the developer performs the functions of Software Quality Management by estimating, tracking, and reacting to the defect data recorded as the project phases are executed.

3 An Approach for Learning the PSP

Humphrey has developed a series of ten exercises to introduce the PSP to the software engineer [2]. Currently, there are two developers at Motorola CIG Fort Worth who have completed the exercises and a third who has started. One developer completed the exercises as part of an elected class project for a graduate software engineering course at Texas Christian University and has since served as a mentor to the second developer in Fort Worth to complete the exercises. Although these individuals were basically self-directed and completed the PSP exercises, Humphrey discourages learning the PSP in a solo manner because of the low success rate [3]. The interest in the PSP at Motorola has prompted the creation of an approach for teaching the PSP to software developers.

Based on the experiences learned in CIG Fort Worth while working through the exercises in a self-directed manner, a proposed process for teaching the exercises was created. This process will be used at Motorola's CIG in Fort Worth, Texas to train other developers on the practices of the PSP. The process was designed to have a minimal impact on the developer's current project assignments and the developer's personal time.

The program consists of a mentoring program where PSP trained developers guide other developers as they complete the PSP exercises. A mentor monitors a developer's progress versus the prescribed schedule and provides advice in clarifying any questions regarding the PSP. In addition, the mentor serves to encourage the developer in completing the exercises in a timely manner. Although an incentive program has not been defined, developers completing the program should be monetarily rewarded and receive a prize of some sort for completing the program.

The initial implementation of PSP practices began in January of 1996. Being the only PSP developer on a multi-person team was very frustrating. Some members of the team and management viewed much of the PSP activity as non-value added. This non-acceptance proved to be quite an obstacle in following through on the commitment to the PSP. There have been many times during the past two years when PSP efforts were almost abandoned altogether. After a presentation on the PSP at Motorola's Software Engineering Symposium, other Motorola employees who were performing the PSP were identified. The enthusiasm and encouragement from them provided the initiative to regroup and maintain the focus on PSP activities. This type of support is invaluable in successfully implementing the PSP. Use of a mentor should provide this much-needed support.

The proposed learning process attempts to spread the PSP work over approximately 8 hours of work per week. Of the 8 hours, the developers are allowed 4 hours of their regular 40 hour week to perform the PSP exercises. In addition, the developers will spend 4 hours of their own time to satisfy the 8 hour per week requirement. Ideally, developers from the same team would be trained together in the PSP. This may make the transition into their actual work projects smoother.

The proposed activities for each week of the training course are shown in Table 7. The tasks for each week were determined by examining the data from the two Motorola engineers who have completed the PSP tasks. Based on the time recorded to perform each task, the tasks have been evenly distributed over a twelve-week period.

4 Taking the Next PSP Step

The presence of one or two PSP developers in an organization is definitely beneficial. The ultimate goal is to have an entire project's staff following the PSP practices. The two largest hurdles in accomplishing this goal are training the developers and having the developers follow the PSP. By overcoming these two obstacles, an

organization can make large strides in optimizing their software process through the efforts of disciplined software developers.

Table 7: PSP Course Schedule

Week	Read Chapter	PSP Tasks
1	1 and 2	Perform exercise 1A.
2	3 and 4	Write R1, R2, and complete 2A.
3	5	Complete 3A and write R3.
4		Complete 4A.
5	6	Complete 5A.
6	7 and 8	Complete 6A and write R4.
7	9	Complete 7A.
8	10	Complete 8A.
9		Complete 9A.
10	11	Begin 10A.
11	12	Complete 10A and write R5.
12	13 and 14	Perform postmortem on course.

The issue of training involves both money and time. The proposed phased introduction and process for training full-time software developers on a part-time basis should be employed. As the popularity of the PSP increases, training accessibility will increase. Currently, training is available through Motorola University, the SEI, and many universities. Motorola NAPSD PPG has experienced significant success in developing a joint training program with Embry-Riddle Aeronautical University. [4] One thing to consider when examining possible training programs should be whether the program allows adequate time to complete the exercises. The completion of the PSP exercises may take 80 to 100 hours themselves.

The hurdle of getting the developers to use the PSP practices stems mainly from data collection and statistical calculations. The tedious nature of data collection and statistical analysis required by the PSP practices creates the need for a software tool for performing these tasks. Such a tool is being developed as part of a graduate school project at Texas Christian University. The tool is designed to provide a simple interface for the developer to record time, defect, and process improvement data. The tool will maintain a database of the developer's project information. In addition, the tool will utilize the information in the developer's database to perform the statistical calculations used for producing estimations, earned value planning, and the various PSP reports. In addition, the tool will provide the capability for a manager to view PSP data at the project level. The tool is scheduled for completion in 1998 and will be piloted at Motorola CIG Fort Worth.

5 Summary and Conclusions

The methods of the PSP are very powerful for optimizing the practices of the individual software developer. The desire to optimize one's software development skills provides the impetus for enduring the detailed PSP practices. The approaches discussed in this paper strive to reduce the magnitude of the learning task and of implementing the PSP by phasing the PSP methods into the developer's work habits.

The idea of taking software process issues to the personal level is paramount in overcoming the "process-for-process-sake" stigma currently associated with software process efforts. Developers complain about the collection of metrics that are not used. The institutionalization of PSP practices creates an awareness of the importance of metrics to each developer. The process followed and data collected is controlled by the individual developer. If the developer deems certain data or processes useless, the personal process can be altered to better suit their needs. This ownership of the process is paramount to the successful utilization of the organizational software process efforts. It is through this utilization that the organization will continue its maturity to levels 4 and 5 of the CMM [1].

The PSP metrics take on new meaning to the developer because the data provides insights into the developer's personal work habits. These insights are instrumental in demonstrating to software developers the weak and strong areas of their process. With this knowledge, a developer knows where to concentrate process improvement efforts.

The practices of the PSP are not magical. There are no earth-shaking revelations for the software development process. However, the PSP practices combine many software engineering methods and incorporate them at the personal level. In addition, the PSP provides a system for tracking the software development process to allow quick identification and effective process improvement.

The organizational ability to generate high quality software in a predictable time frame relies heavily on the software developers. The developers must have a commitment to meet and exceed the quality and time goals established for a project. In addition to this commitment, the developers must have an established personal process to guide their work. The PSP provides a methodology for establishing and optimizing the developer's personal process. This paper provides a methodology for introducing the PSP into actual project work. With this information and the commitment of the software developers, an organization can build their software process efforts upon the shoulders of disciplined software professionals.

6 References

[1] M. C. Paulk, Bill Curtis, and M. B. Chrisis, *Capability Maturity Model for Software, Version 1.1*, Software Engineering Institute Technical Report, CMU/SEI-93-TR, February 24, 1993.

[2] W. S. Humphrey, *A Discipline for Software Engineering*, Reading, MA: Addison-Wesley, 1995.

[3] W. S. Humphrey, "Using a Defined and Measured Personal Software Process," *IEEE Software*, May 1996.

[4] Macke, New, Pang, Nieto, Wirth, Khajenoori, Hirmanpour, "Personal Quantitative Process Improvement Through the Use of the PSP," *Proceedings of the Motorola Software Engineering Symposium*, June 1996.

Successful Implementation of ISO 9001/TickIT in a Software Development Company
Technical Paper

Cecilia M. Yourshaw
Director of Quality
VTLS Inc.
1701 Kraft Dr.
Blacksburg, VA 24060
yourshawc@vtls.com
(540) 557-1200

ABSTRACT

.VTLS Inc. is an international software development company with corporate headquarters in Blacksburg, VA, and distribution partners and sales offices in 18 other countries. The company designs, develops and supports automated solutions for libraries and information centers worldwide. In July 1996, VTLS began the process of creating a quality system that would not only result in ISO 9001 registration just over 12 months later, but also would dramatically improve the quality of the products and services provided to customers.

The quality system in use at VTLS is already showing results. The design of the new product currently under development is better documented through the use of controlled design specifications resulting in less program revisions. Product testing is more structured and consistent with the development and use of test plans by an independent testing group. Finally, and most importantly, feedback from the customers is extremely positive following release of the latest product that was the first to pass through the new quality system. At VTLS, ISO 9000 is the first step down the road to improved quality.

BIOGRAPHY

Prior to joining VTLS Inc. in 1996, Cecilia had 10 years of quality management experience in a large manufacturing organization, including leading their successful ISO 9002 certification effort. In her current position as Director of Quality, she developed and implemented a quality system in the software development arena that resulted in ISO 9001/TickIT registration. In addition to managing the quality assurance functions, she is also responsible for personnel recruitment and development. She graduated with an M.B.A. from Va. Tech in 1982. She is a senior member of ASQ, holds a CQA and is currently serving as Chair of the Radford-Roanoke Section.

INTRODUCTION

In today's global economy, it is extremely difficult, if not impossible, to compete without a focus on quality. This emphasis on quality intensified in the early 1980's with the popularity of Total Quality Management that emphasized the need to embrace quality as an organizational goal. Typically, this included increasing employee awareness of quality principles and objectives, empowering employees to become active participants in the process, recognizing employees for their quality efforts and making customer satisfaction the highest priority for the organization. In response to this increased focus on quality, the International Organization of Standardization (ISO) created the ISO 9000 series of quality standards in 1987. Whereas the focus of TQM is very broad, ISO 9000 is more tailored toward the ability of the supplier to provide quality products and services. This typically includes each step in the product lifecycle from negotiation of customer requirements through product servicing, ensuring that each step is performed in a controlled manner resulting in a product that meets the customer's quality requirements. Although the reviews are mixed on the success of the Total Quality Management efforts in businesses today, there is no question that quality continues to be a key driver for successful companies. The popularity of the ISO 9000 standard after ten years is a testament to its continued importance and validity within the quality arena.

VTLS Inc. is an international software development company with corporate headquarters in Blacksburg, Va., and distribution partners and sales offices in 18 other countries. The company, which employs approximately 100 people with expertise in library science and software engineering, designs, develops and supports automated solutions for libraries and information centers at more than 430 sites in 38 countries.

In July 1996, VTLS management decided to create a quality system that would not only result in ISO 9001 registration, but also would dramatically improve the quality of the products and services provided to customers. The following factors were essential in making this decision:

1. Customer satisfaction surveys showed the need to improve both product and service quality.
2. More and more customers, particularly internationally, required the ISO 9000 certification.
3. Becoming the first library automation vendor to achieve ISO 9001 registration under the TickIT scheme would provide a competitive advantage in the industry.
4. Morale of the employees would improve given better documented procedures and training.
5. The software development lifecycle would be better planned and controlled.
6. Formalized testing for products prior to release would improve product quality through earlier detection of defects.

DEVELOPMENT

The first and arguably most important step VTLS took in beginning this process was to hire a full-time Quality Manager with experience in both ISO 9000 and quality systems development. This individual was given the responsibility, coupled with the authority, to create, implement and maintain the quality system. Additionally, realizing the magnitude and importance of the assignment, this position was placed at an executive level on the organizational chart reporting directly to the President.

The Quality Manager's first task was to perform a gap analysis, which compared the actual operations of VTLS to the ISO 9001 criteria. This proved to be challenging since very few of the existing processes were documented which resulted in a great deal of inconsistency between departments and individuals. Nonetheless, the gap analysis provided a starting point for the implementation phase, including showing the magnitude of the effort required. See Table 1 for a summary of the results of the gap analysis.

ISO Element	Documented Process Exists and Complies with Standard	Process Exists and Complies with Standard but is not Documented	No Process Exists
4.1 Management Responsibility			√
4.2 Quality System			√
4.3 Contract Review	√		
4.4 Design Control			√
4.5 Document & Data Control			√
4.6 Purchasing			√
4.7 Control of Customer-Supplied Product		√	
4.8 Product Identification and Traceability		√	
4.9 Process Control			√
4.10 Inspection and Testing			√
4.11 Control of Inspection, Measuring and Test Equipment		√	
4.12 Inspection and Test Status			√
4.13 Control of Nonconforming Product		√	
4.14 Corrective and Preventive Action			√
4.15 Handling, Storage, Packaging, Preservation, and Delivery		√	
4.16 Control of Quality Records			√
4.17 Internal Quality Audits			√
4.18 Training			√
4.19 Servicing		√	
4.20 Statistical Techniques			√

Table 1

While performing the gap analysis, the ISO 9000-3 Standard (Guidelines for the Application of ISO 9001 to the development, supply and maintenance of software) was used extensively to clarify the application of the standard to the software development process. Additionally, TickIT (Guide to software quality system construction and certification using ISO 9001) was used to provide insight into the requirements of the software development lifecycle. Both of these documents proved invaluable in providing guidance for creating a quality system in the software development organization.

Following the gap analysis, the next step taken was to select a registrar. This was done quite early in the process so that lead time would be sufficient to schedule a pre-assessment. Three of the leading ISO 9000 registrars were invited to make presentations concerning their certification process. Some of the guidelines used to evaluate the registrars were reputation, experience, accreditations held, and the prospective auditors' knowledge of software development practices. As a result of this evaluation, VTLS chose the British Standards Institution (BSI) to be its ISO registrar.

Since implementation of the quality system would affect every employee at VTLS, the Quality Manager held training sessions for all employees. The purpose of the training was threefold: 1) provide an understanding of what a quality system is, 2) provide an understanding of the ISO 9000 requirements, and 3) provide an understanding of the registration process and proposed schedule.

IMPLEMENTATION

Faced with a significant task and the goal of completing the process in one year, we decided to implement the quality system in three phases (See Table 2). Within each phase, the Quality Manager worked closely with the affected departments to address the deficiencies that existed between the current process and the standard. Based on the input provided by the employees closest to the process coupled with the Quality Manager's knowledge of the standard, the Quality Manager created the system level procedures and work instructions required by the ISO 9001 standard. Individuals within the departments reviewed each applicable procedure and work instruction to ensure that the expectations were understood and were achievable.

Phase I	Phase II	Phase III
4.1 Management Responsibility	4.3 Contract Review	4.6 Purchasing
4.2 Quality System	4.4 Design Control	4.7 Control of Customer-Supplied Product
4.5 Document & Data Control	4.8 Product Identification and Traceability	4.11 Control of Inspection, Measuring and Test Equipment
4.14 Corrective & Preventive Action	4.9 Process Control	4.18 Training
4.16 Control of Quality Records	4.10 Inspection and Testing	4.20 Statistical Techniques
4.17 Internal Quality Audits	4.12 Inspection and Test Status	
	4.13 Control of Nonconforming Product	
	4.15 Handling, Storage, Packaging, Preservation, & Delivery	
	4.19 Servicing	

Table 2

Phase I

Phase I established the underlying structure of the system. The critical elements of this phase were as follows:

- Establishing a means for formal management review of the quality system on a regular basis,
- Determining the method by which the quality system would be documented and controlled,
- Developing guidelines for the preparation of project plans,
- Establishing a formal system of complaint resolution, and
- Determining the method to be used for conducting internal quality audits.

In regard to the quality system, the traditional four tier approach to documentation was used.

1. The Quality Manual is the top-level document whose sections correspond to each section of the ISO 9001 standard. It provides an overview of the VTLS quality system.
2. The second tier is the System Level Procedures that also correspond to each ISO 9001 section of the standard and contain *inter*-departmental descriptions of the quality system including the purpose, scope, responsibility and general procedure.
3. The third tier is the Work Instructions, which are detailed *intra*-departmental procedures that describe the specific tasks involved in implementing activities specified in the system level procedure.
4. The fourth tier is the Quality Records, which are the forms, reports, logs and checksheets specified in the work instructions.

The biggest challenge in preparing the quality procedures was providing sufficient detail so that they were valuable reference documents but avoiding so much detail that individual creativity was stifled.

Due to the newness of the quality system, frequent management review was thought to be critical to its success. This review is conducted monthly by the Quality Manager and attended by the President and Vice-Presidents with an agenda that includes such items as:

- Product Defect Data
- Customer Complaints
- Cycle Time for Defect Resolution
- Audit Findings
- Additional Quality Issues

In addition to reviewing the cycle time and resolution of customer complaints at the monthly management review meeting, another method of identifying and solving internal system problems was created. The Corrective Action Request is submitted by employees to report system breakdowns either after or preferably before they occur. These requests are reviewed by the Quality Manager who makes the corrective action assignment and reviews the ensuing corrective action to ensure that the root cause of the problem was addressed.

The internal audit process was designed to be comprehensive and involve employees throughout VTLS. The Quality Manager developed an audit training class for the prospective auditors that included information on auditing principles, planning, performing, reporting and closing audits. The audit schedule is prepared based on the number and severity of findings reported in previous audits. Deficiency reports are generated to report audit findings and are not considered closed until corrective action is completed.

With respect to planning development activities, guidelines for project plans were created which included the following:

- Objective
  - What are the applicable quality factors?
- Organization
  - What will be the responsibilities for the team or individual?
  - What documents should be referenced (i.e., design specification, departmental procedures)?
  - What are the project risks?
  - How will the project risks be mitigated?
- Development Plan
  - What hardware or operating system will be supported?
  - What other VTLS products are involved?
  - What development skills are necessary?
  - What development tools will be used?
  - What programming language will be used?
  - What is the development method?
  - What coding standards are applicable?
  - How will code be reviewed?
- Schedule
  - What tasks will be involved?
  - Who will be assigned to each task?
  - When will each task start?
  - How long will each task take?
  - In what order do the tasks have to be done?
  - How will the schedule be kept up-to-date?
- Testing Plan
  - What types of tests will be done, i.e., Functional, System, Volume, etc.?
  - What test cases will be used?
  - What test databases will be used?
  - How will acceptance testing be performed?
  - What test environment will be used?
  - What test software will be used?
  - What are the pass/fail criteria?
  - How will testing be correlated with a requirements or design specification?
  - How will defects be reported?
  - How will documentation be reviewed?

Phase II

Phase II implemented the software development lifecycle. The key elements in this phase were as follows:

- Preparing the requirements and design specifications,
- Creating coding standards and implementing code review,
- Testing the product,
- Improving the process for delivery and installation of the product, and
- Establishing the role of customer services.

Requiring that design specifications be prepared, reviewed, and approved prior to the beginning of coding demanded tremendous commitment by the management of VTLS. Complicating the situation was that the Development group was midway through the design/development process, which was largely undocumented, of a major new product. The effort involved in preparing design specifications would slow down the production of a project that was already behind schedule. However, everybody realized that although productivity would decline in the short term, it was more important to focus on the long term benefits—that is, a product that better meets the user's requirements and is more stable due to fewer revisions.

To maintain consistency, design specification templates were created for both the functional and implementation levels that included the following:

Functional Design Specification

- Overview and objectives,
- Special environmental constraints,
- Document references,
- Glossary of abbreviations and terms,
- Description,
- Possible areas for future enhancement,
- Suggested user documentation and
- Maintenance.

Implementation Design Specification

- Overview and objectives,
- Document references,
- Glossary of abbreviations and terms,
- Environment constraints,
- Programming language,
- Programs affected and new programs,
- Database changes,
- Code structure,
- Workflow and
- Future enhancements.

When completed, the design specification is routed for approval and placed in the document control system where future revisions are controlled. Only when this step is completed is the programmer authorized to begin writing code in support of the design specification.

Another significant change to the software development process occurred within the implementation of the process control section of the standard. Coding standards were created for each programming language. The software engineers were required to use these standards as guidelines for the preparation of source code. Additionally, upon completion of a section of code, a senior software engineer reviews the code, ensuring the use of good programming fundamentals. This procedure means cleaner code, improved probability of reuse and fewer bugs.

Another significant step was taken in regard to testing the software. An independent Quality Control group was formed who reported to the Quality Manager. This group was responsible for reviewing the design specifications and then preparing test plans to validate the code. In addition to the unit testing performed by the programmers, Quality Control performs two types of testing. As a feature is added or a bug is fixed, Quality Control validates that particular change through progressive testing. Then, upon receipt of a frozen software release, Quality Control conducts regression testing to ensure that a prior change has not affected another area of the software. Additionally, prior to general release, the Quality Control group administers the beta testing, which occurs at customer sites.

The method of handling software deliveries and installations underwent a major change as well. Due to the amount of planning and coordination required to install a new site, a project management group was formed. The responsibility of the project coordinator within this group is to oversee the installation, including acting as a liaison for the customer and working with other VTLS departments to complete the necessary activities. Additionally, the project coordinator prepares an implementation plan for each site that typically contains:

- Hardware and software purchases,
- Pre-installation activities,
- Record loading/conversion information,
- Delivery, installation and training activities and
- Customer acceptance testing requirements.

Following acceptance of the installation, the customer is then assigned to a Customer Services team who handles future support that includes providing system and usability training for the site and tracking customer concerns or bugs until closed.

Phase III

Phase III completed the implementation by addressing the remaining supporting elements of the standard. These included:

- Setting up a document control system,
- Evaluating and inspecting vendors,
- Handling customer-supplied data/software/hardware,
- Managing product configuration,
- Establishing criteria for product release,
- Developing a training plan, and
- Implementing a measurement system.

The on-line document control system applies to the Quality Manual, System Level Procedures, Work Instructions, Forms, Design Specifications, Project Plans, Test Plans and User Documentation. Each document has an approval and distribution list associated with it. Upon revision of the document, a cover sheet is attached which summarizes the changes to the document and is routed to the specified approvers for review and approval. Following approval, the document is posted on the company's intranet for easy access and the distribution list is used to notify the affected users that the document has changed. Since the intranet simplifies the process of locating documents, it eliminates the problem of employees using out-of-date printed documents and forms.

Following the completion of the system level procedures, work instructions and forms, the Quality Manual was prepared by the Quality Manager. This document, which provided a summary of the VTLS quality system, became easy to prepare at this point since it was written by summarizing the system level procedures.

INITIAL ASSESSMENT

Upon completion of the Quality Manual, BSI conducted a 2 day pre-assessment. The main focus of the pre-assessment was to ensure that the quality procedures were adequate in addressing all elements of the ISO 9001 standard. The results of the pre-assessment were 8 findings that highlighted weaknesses in the areas of definition and preparation of design specifications, servicing and configuration management work instructions, and planning of design and development activities. All findings noted in the pre-assessment were addressed through modifications to the procedures and the process.

Training sessions were set up with each department to go over each work instruction in its entirety. The purpose of these sessions was not only to inform the employees of their new responsibilities but also to elicit any final questions or concerns about the procedures. As each training session was completed, the procedures were revised as necessary. Following the completion of all training sessions and subsequent procedure revisions, the procedures were routed for approval, and all employees were notified of their approval status. Typically, as the procedures were being generated within each phase, the employees began following them to identify areas that were problematic. In this way, by the time the procedures were actually approved, the employees were comfortable and complying with them.

The final step taken prior to the initial assessment was an internal quality audit of all sections of the ISO 9001 standard. The focus of this audit was to concentrate on each department's knowledge of and compliance with the procedures. The Quality Manager performed the audit over a 2 week period. Many of the findings reflected the immaturity of the quality system including incomplete contract review records, inadequate requirements specifications, uncontrolled project plans, and inadequate inspection of subcontractors. Following the review of all findings, and the resolution of all corrective actions resulting from the findings, the initial assessment by BSI was conducted.

The initial assessment consisted of a four day review of VTLS' quality system by a TickIT registered BSI auditor. As a result of the initial assessment, 12 findings were noted, at least half of which were easily corrected with minor procedure revisions. The auditor's comments were that the quality system was well conceived and that the intention and effort to implement the system was well-evident. Based on this positive assessment, the auditor recommended that VTLS receive certification to ISO 9001 following completion of all outstanding corrective actions from the findings generated during the initial assessment. On September 18, 1997, VTLS received the Certificate of Registration that certified that VTLS Inc. operates a quality management system that complies with the requirements of BS EN ISO 9001:1994 for the design, development, installation and maintenance of software for libraries and information centers worldwide. The total time required from initiation of the project to registration was just over 12 months.

CONCLUSION

The development and implementation of the quality system at VTLS was not without problems. However, due to the high level of commitment and support by management, each problem was solved and the process continued to move forward. In retrospect, some parts of the implementation were very successful while other parts could have been more effective if done differently.

Considering our experience, we would make the following recommendations to other companies that face a similar situation:

1. Make absolutely sure that senior management understands the magnitude of the project and is willing to devote adequate resources to it.
2. Depending on the size of the operation, assign at least one full-time resource to be given the responsibility and authority to complete the project. This individual should be at a fairly high level in the organization to have the most success in the shortest period of time.
3. Hold regular management reviews of the process to ensure that progress is being made and any outstanding issues are being addressed promptly.
4. Ensure that the procedures are prepared from the bottom up; that is, ensure that information is extracted from the individuals who are doing the work.
5. Train, train, train. Make absolutely certain that from the beginning of the process to the end of the process all employees are kept informed as to the status and their responsibilities within the process.
6. Break the project down into phases that are implemented gradually.
7. Schedule a pre-assessment with the selected registrar midway through the process. Not only does this help ensure that you are on the right track, but also it allows you to begin developing a relationship with the auditor to better understand their expectations.
8. Listen to employees' concerns about the process. If you attempt to force new procedures and processes on them without convincing them of their value and obtaining their buy-in, it will be an uphill struggle.
9. Crawl before you walk. When you begin writing procedures, the tendency is to try to develop the perfect system from the very beginning. What results is frustration because you will never complete the process. Concentrate on establishing a basic quality system that meets the standard first, and then you can begin to add improvements once this structure exists.
10. Make the quality system everyone's responsibility by having everyone participate at some time in an internal quality audit.

The quality system at VTLS has already improved our processes, productivity and customer satisfaction levels. The design of the new product is now documented through the use of controlled design specifications, which is resulting in a product that better meets the users' requirements. Productivity is improving now that programs are no longer being rewritten because of design changes that occur late in the process. The use of coding standards and code review is leading to cleaner code that is better documented and also is resulting in the detection of potential bugs much earlier in the process. Product testing is more structured and consistent with the development and use of test plans that are run by an independent testing group. Finally, and most importantly, VTLS customers are recognizing and commenting on the improvements they are seeing after receiving the first product release that went through the new quality system.

Although developing and implementing a quality system that complies with the ISO 9000 standard is challenging for any organization, in our experience it can be done successfully and will pay countless dividends when done right.

REFERENCES

American Society for Quality Control Standards Committee for American National Standards Committee Z-1 on Quality Assurance. *Quality Management and Quality Assurance Standards—Guidelines for the Application of ANSI/ISO/ASQC Q9001 to the Development, Supply, and Maintenance of Software* Milwaukee, Wisconsin: ASQC, 1995.

American Society for Quality Control Standards Committee for American National Standards Committee Z-1 on Quality Assurance. *Quality Systems—Model for Quality Assurance in Design, Development, Production, Installation, and Servicing* Revision of first edition (ANSI/ASQC Q91-1987). Milwaukee, Wisconsin: ASQC, 1996.

The TickIT Guide. Issue 3.0. London, England: DISC TickIT Office, 1995.

Title: A Problem-Based Approach to Software Process Improvement: A Case Study

Primary Contact Author: Johanna Rothman

Email address: jr@jrothman.com

Mailing address: 38 Bonad Rd., Arlington, MA 02174

Work phone: 781-641-4046

Home phone: 781-641-1957

Fax: 781-641-2764

http://www.jrothman.com

Key Words/Phrases: software process improvement, software engineering, software product development, program management, project management, concurrent engineering.

Author Biography:

Johanna Rothman is a speaker, trainer, and consultant on issues relating to high technology management and quality. She works with organizations to improve their product development practices resulting in increased productivity and quality. She has over twenty years experience in the software engineering and software management profession. Johanna is a member of the clinical faculty of The Gordon Institute at Tufts University.

Johanna has a MS in Systems (Software) Engineering from Boston University. She also has a BS in Computer Science and a BA in English Literature from the University of Vermont. She holds two ASQ certifications: Certified Quality Auditor and Certified Software Quality Engineer.

A Problem-Based Approach to Software Process Improvement: A Case Study

Abstract

Organizations struggle [1] with their process improvement efforts for a variety of reasons. Perhaps the most common struggle pattern is to take a long time developing a general understanding of their processes and then trying to define all possible alternatives in the product development process. This pattern leads to large, unmanageable, unreadable, and incomplete [3] process documentation. The long process development duration tends to make the technical staff suspicious of the process itself. Add the unreadable process development documentation and the result is that the staff is generally reluctant to accept the end result.

This paper is a case study of one organization that minimized the struggle by taking a different approach to the development of their product development process. Management believed that the key to product development success was to plan for success, while understanding the current problems. They were convinced that product development quality and timeliness comes about when the process is clear, easy to understand, and leads to the desired end result. Planning for failure was not to be part of the process definition and improvement effort—each project needs to plan for its own risks. But management wanted to make it clear that reasonable people, following a reasonable process, could produce products successfully.

The process group took an approach that incorporated: solving their specific problems, frequent testing, and rapid, evolutionary delivery of results. This paper describes the efforts of that steering committee to define a reasonable, but not comprehensive, product development process.

Introduction

The organization described in this paper was a twelve-year-old company, formed out of two startups. It created and sold graphics products. We'll call this organization "ExtendIt." ExtendIt employed about 150 people wolrdwide. The product development staff was split into two locations: about 50 people in the Boston area office, and about 20 people in a European office.

ExtendIt was in a typical chaotic state—most of senior management did not understand the software product development process. Engineering management didn't know where or how to begin, project management and product management were non-existent, and engineering processes were completely inadequate for product development and testing. Projects were planned for 4-8 months, but typically took 13-18 months. Even at the end of the extended development time, ship decisions were generally based on emotional reasons to ship, not objective reasons. For example, management made the decision to ship a major release because the developers were too tired to continue the 80-hour weeks, not because the project met the ship criteria. In fact, that particular project did not have all the expected features, so the developers continued to work long hours to get the features into the follow-on release.

Approach to Process Improvement

A new CEO started at ExtendIt and changed the product strategic vision and sales model. Based on the new goals, it was clear that the organization had to change how it developed products. It was not possible for this geographically dispersed Engineering organization to meet the new goals without changing their practices.

Senior management had already agreed to decouple releases from project development, which is a typical concurrent engineering approach to product development. This would be known as the "Release Train," a quarterly plan to ship products[1]. Projects at a certain point in their development would be eligible to be loaded on the train and be shipped. Projects would not be shipped unless ready. To meet the release train goals, ExtendIt formed small independent projects.

[1] Companies who have the need for parallel development of multiple releases use this concept. Although Sun has implemented this differently, the release train idea described in http://solaris.license.virginia.edu/sun_microsystems/workshop4.2_docs/teamware/solutions_guide/casestudy.doc.html#8868 is similar in concept.

A software engineering process group (SEPG) [2] was formed in May 1997, with the original plan that the process definition and design could be completed by the end of July 1997, a total of eight weeks. The SEPG consisted of Engineering management (the VP of Engineering, the Documentation Manager, Development, and Release Engineering Managers), the Director of Program Management, and an outside consultant, a total of seven people.

The initial roles of the people on the SEPG were:

- The VP Engineering was the facilitator between the SEPG and organization at large.
- The Documentation Manager served as the Chairperson of the SEPG and provided expertise about documentation processes.
- The two Development Managers and the Release Engineering Manager provided expertise about current processes and how they could be changed.
- The Program Management Director provided specific Engineering expertise and general organizational expertise about product development.
- The consultant provided planning and facilitation for the SEPG's meetings in addition to process and product development expertise during the process design.

Like many organizations, the SEPG planned to roll out the process definition and templates to the organization *à la* the Hole-in-the-Floor model of change [5] [2]. The rollout milestone was planned for August 1997. After the initial SEPG effort, Engineering management was to carry out ongoing process change.

This SEPG forgot one thing—change is not successfully rolled out to organizations [5]. People have to integrate the changes into their daily lives for the change to be successful. Although this SEPG did not anticipate this, changes were introduced and integrated into the organization in a most fortunate and successful way.

Problem Statement

The SEPG began by discussing what had to change. Using brainstorming, they identified 29 issues. Then they used affinity grouping to sort the 29 issues into 9 "buckets." [3] Each SEPG member cast three votes, and voted on their top three issues. They took the top 80% of the problems and threw away the lower 20%. The result of this analysis were the following six problem statements:

1. The Product Development Process was not documented. The process was not uniform among projects.
2. The Functional Specs/Design specs were not separated. Because the functional description and the design were intertwined, some parts of the system were not well defined and the test planning effort was insufficient.
3. Vague Marketing Requirement Documents (MRDs). Marketing told development how, not what to do.
4. Development's intake of market requirements are not well defined or controlled. This was really an organizational problem—getting a single point of contact for discussing issues.
5. Too many off-process interruptions. The Engineering staff was interrupted, or dragged off to work on other issues. There were no organization-wide rules about how to get consulting from others.
6. Managing to a schedule was a problem. People did not know how to manage their own time, or how to rank their activities.

Each SEPG member wrote six descriptive sentences describing each problem as it appeared to or affected each person. The SEPG called this their "6x6" matrix, for six sentences about each of six problems. Everyone's sentences were gathered into a concept matrix, with each major item on the left (the numbered items above), and the

[2] The Hole-in-the-Floor model of change: Some set of people upstairs develops the perfect system. The change plan consists of drilling a hole in the floor. The system is dropped through to the people below. Supposedly people instantly change to the new system. Unfortunately, people generally can not change without integration and practice.

[3] Affinity grouping is the activity of creating sets of similar ideas together under one theme. In this case, we wrote each the problem on a sticky note, silently organized the sticky notes into groups, and then named each group.

relevant issues on the right. The SEPG then grouped the problems into subcategories, to organize the issues. See Table 1 for a representative portion of the final set of problem statements.

Product Dev. Process not Documented	A: Central Reference Required	1. There is no "playbook" which can be given to all employees, so they know the process for developing software. 2. Missing the "what, when and who" for product development maintains our current (perhaps our past) operating procedures (i.e., controlled chaos) 3. Without a doc'd product development process, there's no real way to determine where we are in the development process (there's no “starts with" or "ends with" statement or entry or exit criteria). 4. …
	B: Common Terminology	5. Terminology is imprecise, e.g. "Alpha" means something, but not the same thing to different people. 6. …
	C: Phase Definition/Criteria	7. There is no current opportunity to define project success criteria. 8. There is no current opportunity to know when a project is complete. 9. There are no clear phases, with entry/exit criteria to know what is done and what is not. 10. …

Table 1: Portion of Concept Matrix describing problem statements

The final concept matrix has a generic problem statement, specific issue, and examples of how each issue affected the organization. The SEPG then made a critical decision—the SEPG decided to focus their work on just the six problem statements above: documenting the product development process; separation of functional and design specs; specific MRDs; how development took in requirements; managing interruptions; and managing to a schedule. This focus provided these main benefits:

SEPG modeled problem solving behavior	Not every decision was correct in hindsight, but the problems were discussed in context of the problems the SEPG was trying to solve. The decisions and the decision-making process were accessible to the organization.
SEPG practiced problem solving skills	The managers were on the SEPG. They had a chance to practice their problem solving skills in an environment of their peers, before trying them out on a project. This included practice using the traditional problem solving skills and tools, such as brainstorming, affinity grouping, and facilitating discussions of diverse ideas.
Focused SEPG work	ExtendIt was working towards a rational way of doing business, not towards public certification or assessment. Using the business as incentive for the process improvement activities was understandable by the management and technical staff.

Intermediate results

The VP of Engineering and some SEPG members felt very strongly that some aspects of product development could not be planned. The VP wanted the SEPG to take an approach to process definition that facilitated reasonable things for reasonable people to do. The SEPG would then incorporate management reviews into the process sufficient to inform management, and enable management to take appropriate steps. In addition the process documentation would give general problem-solving guidance. (On-line documents describing useful meeting techniques and useful project management techniques were part of the final deliverables.)

The SEPG approached the process definition work as if it were an Engineering project. The work started with a strawman five phase process:

Concept/ Requirements	Design/ Definition	Coding/ Implementation	Validation/ Verification	Manufacturing/ Ship

Starting from its charter, the SEPG initially refined its concept (Concept/Requirements phase). The SEPG took the time to define its requirements and an initial project plan, to clarify project completion and success criteria.

To clarify and define SEPG deliverables, the initial SEPG project plan used the phases above: Concept/Requirements; Design/Definition; Coding/Implementation; Validation/Verification; Manufacturing/Ship.

During the design and definition phase, the SEPG defined the functional specification and design. The SEPG made an initial cut at the phases, figured out what the necessary documents had to be, and where the review points were.

The implementation phase consisted of the detailed design of the process description, and generating the flow charts and words to describe it. To get early testing, and to get Engineering buy-in, the SEPG held focus groups to discuss each phase. Getting early Engineering input had these benefits:

1. The SEPG's work was visible to the organization. In fact, parts of the organization were able to "test" the process by using pieces of it on ongoing projects. Doing this early testing has some ramifications:
   - The SEPG could see if the people who were supposed to use the process would actually use it.
   - A number of issues arose during these focus groups. The discussion around these issues allowed the SEPG to change and simplify the process.
2. The SEPG was able to gain substantial experience in presenting the process to the organization. When the focus group was confused, the SEPG could test how the focus group understood different descriptions.
3. The SEPG walked the talk of "early and often" review and testing. By having their work held up for review and verification, it was easier for the Engineering staff to buy into frequent reviews and early testing.
4. Using an evolutionary process design meant the SEPG didn't have to get everything right the first time. The Engineering organization could see this, and see the relevance to their work.

At the end of the implementation phase, the five-phase product development process had evolved:

Concept/ Requirements	Design/ Definition	Coding/ Implementation	Validation/ Verification	Product Qualification

Disadvantages of this approach

There were two notable disadvantages of this approach. The SEPG worked very quickly, so it was hard for some people to integrate the changes to how they thought. Although the SEPG members did not have trouble with the concept of iteration, some had trouble with their ability to iterate their thoughts quickly. These SEPG members were thrown into "chaos" [5] with almost every meeting, and had a difficult time adjusting to the pace of change. Change can be painful to the people involved.

During the SEPG's work every member had to closely examine and change or give up closely held ideas about product development. Changing your mind about something when you do not have direct experience with its potential for success can be very hard. Some of the SEPG members were quite reluctant to change how they worked, even when they admitted their current patterns were not working.

For example, the SEPG intellectually understood that inserting a milestone at the beginning of the Coding/Implementation phase to verify the release criteria against each project's criteria made sense to everyone. Some SEPG members were concerned that these release criteria would be fixed too soon and would be non-negotiable. They were concerned that they would be forced to develop the wrong product. The rest of the SEPG, from experience, realized that clarifying release criteria before the code is finished, is one easy way to make sure that the product under development is the correct product. The reluctant SEPG members were concerned, because they had no experience with the success of release criteria. They knew their current methods were inadequate, but were reluctant to agree to something they had no direct knowledge of. As an SEPG, we agreed to mini-retrospectives during the first few projects, to check on this and other points in the process.

Some of the SEPG members also had trouble changing their meeting behavior. Some team members were stuck in legacy behavior, using the same assumptions that had created the problems. One assumption was that all decisions were open to more discussion and change after the decision was made. It was impossible to make progress when all decisions could be revisited at any time by anyone.

Consequently, the SEPG remained stuck in the storming phase of team development [4]. After discussing these problems with the SEPG chairperson, the consultant requested the VP Engineering attend some team meetings. The presence of the VP acted as an inhibitor to "business as usual," and allowed the team to make appropriate decisions and move forward. In the case of the SEPG's decision-making, the VP verbalized the SEPG's responsibilities and the time to deliver on those responsibilities.

Results of using the process

SEPG Results

The original dates were very aggressive (an eight week schedule), and were not met. Missing the original dates created these results:

- The SEPG was able practice iteratively replanning their schedule. This experience was directly applicable to normal Engineering projects.
- After the first milestone was missed, the SEPG was able to practice testing their work focus. Were they working on the most time critical and valuable item?
- The SEPG clarified their tradeoff decisions and decision-making process. They created a "Pending Bin" to place ideas and issues that were relevant to address, but not now.

All these issues emulated typical challenges of a product Engineering project—one would do them on a project. They were good practice. The SEPG gained understanding that their work was a process development process. The end result was not a saleable product, but it was a process where similar tools and ideas were useful.

Product development results

Initially, the Engineering staff was concerned about changes to how they were expected to do product development. At the initial overview presentation of the Release Train, the Engineering staff was confused by terminology and how to do what, because the specific changes to the process were not rolled out. The SEPG started its work after this initial presentation.

To get buy-in from the Engineering staff, the SEPG started focus groups to discuss the process steps and then the templates in group meetings. The SEPG chose one SEPG member to present each lifecycle phase to the focus group. The focus group would ask questions, and the designated SEPG member answered the questions. The rest of SEPG staff took notes about the presentation and the questions. When there were many questions, the SEPG generally redesigned the process to make it easier to understand, easier to implement, and more streamlined.

After the process was reviewed in the focus groups, the templates (plans and specification documents) were reviewed in focus groups. The SEPG used the same process: one SEPG member presented the material, and the focus group commented on the material.

By the end of the focus group activity, all the senior staff in Engineering had seen parts of the process and the templates. Because the Engineering staff helped create and review the process and the templates, the senior staff led the rest of the technical staff to adopt the process.

At the next general presentation, the overall process was discussed. The Engineering staff understood the process and the templates, because it was clear what they had to do and when.

Lessons Learned

ExtendIt learned a tremendous amount from these steps to process improvement: a process improvement process. They were able to avoid some typical process improvement problems:

Typical problem	Avoidance
Process improvement effort takes a long time.	Focus the SEPG's efforts on the problems that need to be solved now. The Engineering staff was not only willing to use the process; they demanded it on projects.

Define all possible steps in the process.	The process provides reasonable guidance and specific criteria for escalation to management. Project participants use the process as a guide. They use their judgement about how to deal with problems, until the problems need to be escalated to management.
"Big Honking Binder" syndrome: the size of the documentation overtakes the process definition.	One page process flows with one-page descriptions. One and two page document templates are part of the process definition. The whole process document is 20 pages.
Technical staff is suspicious of process development process and reluctant to adopt outcome.	Test the process with staff as it's developed.

Table 2: Lessons Learned by ExtendIt

Conclusions

This process improvement process was very effective. It consisted of first determining the problems that needed solving, then developing a process that illustrated the way to do the general case, and a set of problem solving skills. Before the SEPG even finished its work, at about 8 weeks after its formation, its members started to work differently. The SEPG thought about their deliverables to each other in a more complete way—how people could use what they developed, and the effects of their deliverables on other deliverables.

The biggest organizational change was that managers and the technical staff thought differently about how to do their work. They started to plan for the reasonable case, and created a risk assessment and management plan. This had the desired effects of creating simpler project plans, and pushing risk assessment into the organization.

A small process description seems to be adequate for the present for this organization. The process description contains 5 pages of flowcharts, about 4 pages of definitions, and about 5 pages of prose describing the process and general problem solving techniques. In addition, there are templates for each document Engineering has to produce.

ExtendIt has now been using this process for almost a year. It has successfully produced three quarterly release trains. The technical and management staff has tested the process, and for now, it works.

ExtendIt has had a difficult time escaping from its startup phase. The new CEO and senior management are determined to make the company a success. From a product development perspective, the organization can now deliver products on time and within budget, with the requested features. Using the Release Train to chunk the features into smaller independent projects, and by creating the expectation that the organization would deliver multiple products over the course of the year, ExtendIt is operationally poised to succeed.

Acknowledgements

In addition to the anonymous reviewers, I thank the following reviewers for their help and substantive comments: Don Gray, Brian Lawrence, Sue McGrath, Jerry Weinberg.

References

1. Hayes, Will and Dave Zubrow, *Moving on Up: Data and Experience Doing CMM-Based Software Process Improvement*, CMU/SEI-95-TR-008, 1995.
2. Humphrey, Watts, *Managing the Software Process*, Addison-Wesley, Reading MA, 1989.
3. Murphy’s Law. Specifically, “Whatever can go wrong will, at the worst possible time”.
4. Scholtes, Peter R. Joiner and Streibel, *The Team Handbook*, Joiner Associates, Madison, WI, 1996.
5. Weinberg, Gerald, *Quality Software Management: Volume 4: Anticipating Change*, Dorset House Publishing, New York, 1997.

View Slides

Eighth International Conference on Software Quality
Pacific Northwest Quality Conference
Technical Paper

Presented by
Karen Bishop-Stone
Testware Associates, Inc.
6313 Hillside Road
Edina, MN 55436

I've Been Asked to Review this Specification. Now What Do I Do?

Errors found during system level testing can be as much as 90 times more expensive to correct than those uncovered during the document creation process. Studies have shown that effective document reviews at the requirements, analysis, and design phases of the software development life cycle uncover as many as 80% of the defects within the system. The evidence speaks to the effectiveness of the outcomes but few software professionals are trained in actual techniques and methods for performing the task of reviewing a document.

A hundred pages of documentation for your review have just been dumped on your desk with a meeting schedule attached. Now what do you do? You can ignore the task assuming that the other team members will perform magic. Or you can read the document cover to cover and start day dreaming around page 8. Or you can use one of several preparation techniques, which will make your review task comprehensive and enjoyable.

Biography:

Karen Bishop-Stone has taught seminars on Software Testing and Quality Assurance Management internationally since 1980 and is a national conference lecturer on software life cycle testing methodologies. She has recently managed testing in eight states for a large, complex, federally mandated program. She has been certified as a Certified Software Quality Analyst with the American Society of Quality and as a Certified Software Test Engineer with the Quality Assurance Institute. Ms. Bishop-Stone is the principal owner of Testware Associates, Inc., a firm dedicated to the independent testing and quality management of software.

Eighth International Conference on Software Quality
Pacific Northwest Quality Conference
Technical Paper

Presented by
Karen Bishop-Stone
Testware Associates, Inc.

I've Been Asked to Review this Specification. Now What Do I Do?

A hundred pages of documentation for your review have just been dumped on your desk with a meeting schedule attached. Now what do you do? You can ignore the assignment assuming that the other team members will take care of the task. Or you can read the document cover to cover and start day dreaming around page eight. Or you can use one of several preparation techniques, which will make your review task enjoyable and comprehensive.

Errors found during system level testing can be as much as 90 times more expensive to correct than those uncovered during the document creation process. Studies have shown that effective document reviews at the requirements, analysis, design and code phases of the software development life cycle uncover as many as 80% of the defects within the system. The evidence speaks to the effectiveness of the outcomes but few software professionals are trained in actual techniques and methods for performing the task of reviewing a document.

The review of documentation is to insure that the message received is the same as the intended message sent. The reviewer attempts to understand as much about the system as possible while looking for inconsistencies, omissions, and unintended interactions within and across the system. When reviews are accepted as an effective form of testing, product rework is reduced, interfaces are improved and a testing perspective is established early in the project.

No matter what development methodology is used each phase of the software development life cycle produces some form of documentation which should be reviewed by either a formal or informal process. What ever format is used - technical review, walkthrough, or inspection - each team member is assigned a role from whose perspective the document is reviewed. This avoids re-creating work the author has already done and brings fresh insights to the specification. The effectiveness of the test is often determined by the selection of the team members and the roles they play.

Different preparation techniques are best applied on different documentation types. Although several of the techniques are used for test case design, the initial organization of the information is often an effective approach for reviewing the documentation. The outcomes of these efforts can then be carried forward to the testing phase. With these techniques it’s not so much the perfection of the outcome as it is the process used to reach the outcome.

The following is a brief description of various techniques used for review preparation:

Requirements traceability matrix is a method for tracking the relationships of the requirement with its solution, design, module code, and tests. The building of the initial requirements matrix will often surface issues within the document set.

- List the requirements as bullet items restated as a verb and direct object.
- Group the requirements into functions and subfunctions.
- Prioritize, from the user's perspective, all requirements as 1/3 high, 1/3 medium, 1/3 low
- Ask the following questions:
  - How will this requirement be manifested in the system?
  - How will the requirement be tested?
  - What will the user do if the requirement is not met?

Checklists are used to ensure the completeness, consistency, and accuracy of documents. These are used in all phases of the development cycle. Errors of omission are best identified with this technique.

- Start with a list of standards defined for the company/department/project.
- Compare each item on the checklist to the document.
- Consider the primary, secondary, and tertiary removed user of the product under development.

Data Reconciliation addresses the birth, life and death of a data item. It is a technique used to ascertain if enough information is available to determine how the item is used throughout the system and if the item is used consistently.

- Build a table of all data items identified in the document
- Describe the item's use.
- Sort the list by item, then by use
- Ask the following questions:
  - Do all items have a create function?
  - If created from multiple functions are the same edit rules applied?
  - Does each item have an output or is used in a calculation?
  - Does each item have a termination?

Equivalence Partitioning is another technique used to evaluate the effectiveness of the data descriptions and uses. Enough information should be included to allow the organization of the data into classifications.

- Group the data into classes representing valid and invalid sets.

Boundary Value Analysis is an extension of equivalence partitioning and is used to ensure that the data contains appropriate boundary limits.

- Group the data into classes representing valid and invalid sets.
- Examine the appropriateness of the expected result based upon the lower and upper limit defined for each item.

Complexity process models ensure that all branches of each decision flow contain information. Often documents describe the outcome of one side of a decision but ignore the other side of the decision flow.

- Build a model of the decision structure of the process
- Describe the results of each decision.

Plan on spending the scheduled meeting time length in preparation for the review. If the meeting is scheduled for two hours then spend at least two hours reviewing the documentation. The amount of documentation to be reviewed per meeting varies depending upon the type of document under review. The following estimates are to be closely followed. If the average time is always less then the reviewer is not focused enough on the document. If the reviewer consistently exceeds the time recommendations, they are going into too much detail. There is a point of diminishing returns when preparing for a review.

Document Preparation Time Estimates

Requirements	6-10 Pages per Hour
Technical Design	10-20 Pages per Hour
Code	125-200 Lines per Hour
Test Plans and Cases	10-20 Pages per Hour
User Documentation	10-20 Pages per Hour

Source: JPL

Once in the review session the reviewers follow basic conduct rules. The objective of this meeting is to identify document errors and concerns. It is not for the purpose of correcting the issues. All comments are to be directed toward the product not the producer. At this point in the process the document has become a product of the entire review team, not just that of the author.

A reliance on any single form of testing – black box vs. white box or static vs. dynamic will produce only a single type of software error. Therefore it is important to remember that a balance must be struck between document reviews and machine execution of the tests. And a balance is also important between structural types of test case design and external user tests. The success of the document review test is dependent on the effectiveness of the document review preparation.

IEEE/EIA 12207 AS THE FOUNDATION FOR ENTERPRISE SOFTWARE PROCESSES

James W. Moore
The MITRE Corporation
1820 Dolley Madison Blvd., W534
McLean, VA 22102, USA
Work Phone: +1.703.883.7396
Fax: +1.703.883.5432
James.W.Moore@ieee.org

Abstract

It is widely believed that the adoption of predefined processes aids the productivity and quality of software development. Furthermore, it is widely believed that the adoption of these processes at the enterprise level provides advantages of repeatability and organizational maturity that amplify the benefits. Unfortunately, the existing corpus of software engineering practice standards has been targeted for adoption at the project level rather than the enterprise level. A new standard, IEEE/EIA 12207, *Software Life Cycle Processes*, addresses this problem—it is intended as an integrating, organizing, strategic standard specifically directed to enterprise adoption and intended to form the foundation for the improvement of enterprise processes through the adoption of related standards.

KEYWORDS: Software engineering, standards, process, capability maturity, process improvement

Biography

Jim Moore is a twenty-nine-year veteran of software engineering and a ten-year veteran of software engineering standardization. With degrees from the University of North Carolina and Syracuse, he has worked in both the commercial and defense sectors for IBM and, now, The MITRE Corporation, where he is the corporate focal point for standardization activities. Currently, he serves as the chairman of the international standards committee for the Ada language, as a member of the Management Board of IEEE Software Engineering Standards Committee (SESC), as a member of the IEEE Standards Board, and as the Vice-Chair of the U. S. delegation to the international standards committee responsible for software engineering. He served for four years as a member of the Defense Department's Federal Advisory Board on Ada. He is a Senior Member of the IEEE and the IEEE Computer Society has recognized him as a Charter Member of their *Golden Core*. His book, *Software Engineering Standards: A User's Road Map*, was published this year by the IEEE Computer Society Press.

IEEE/EIA 12207 AS THE FOUNDATION FOR ENTERPRISE SOFTWARE PROCESSES

James W. Moore
The MITRE Corporation

1 Introduction

A project manager desiring to adopt a sound set of processes for software development faces a daunting task. A large number of important issues inevitably influence the definition of the needed software engineering processes. Some of these influences include:

- The *project management practices* of the enterprise and/or the customer.
- The *systems engineering methods* being used for the containing system.
- Specific *process requirements* levied by the customer.
- *Total Quality Management* practices imposed by the enterprise, by the customer, or by choice.
- *Governmental regulations* applicable to the software and/or system.
- *Safety, security, and privacy requirements.*
- *Best practices* selected by the development team.
- *Contractual requirements.*
- The organization's competitive need for satisfactory *capability evaluations.*
- Tooling and other *infrastructural constraints.*
- The *corporate initiatives* du jour.

Confronted with so many important sources of "help," it is little surprise that many project managers choose to "get on with the job" rather than pay attention to careful process definition. The result, of course, is that many software development projects utilize ad hoc processes implemented and modified on the fly by capable and not-so-capable developers.

The ultimate contribution to the project manager's problem would be to take away the problem entirely. We would hope that any given project could, with minimal customization, execute a set of processes already defined at the enterprise level. Besides the head start given to the project, this approach offers important advantages to the organization: repeatable execution, portability of staff, common infrastructural support capabilities, the potential for organizational improvement. One approach might be the application of voluntary standards for the practice of software engineering.

Software engineering standards cover a wide scope of subjects. Although they sometimes deal with supporting tools or product characteristics, the most familiar ones are process standards dealing with subjects like Configuration Management, Verification, Validation, and Quality Assurance. Unfortunately, the erstwhile user of these standards is presented with a huge problem in selecting the ones appropriate for usage because more than 315 different standards, guides, handbooks, technical reports and other normative documents are offered by nearly 50 different professional, sector, national, and international organizations [Magee97]. Some sources cite even higher numbers. The two most

important collections are provided by ISO/IEC JTC1/SC7[1] and the Software Engineering Standards Committee (SESC) of the IEEE Computer Society.

After more than a decade of focus on organizational process improvement, it is generally accepted that a software organization is best served by the adoption of a single, uniform set of processes repeatedly executed by all of its projects. Nevertheless, most software engineering standards have not yet caught up with this development. They continue to address the subject of process adoption by the individual project. Conformance is evaluated at the project level. Although there is nothing inherently wrong in this approach, it ignores both the investment and the benefit in adopting a sound and comprehensive set of enterprise processes. Needed is a new approach that gauges conformance by the enterprise-level adoption of suitable processes and their implementation through organizational procedures, practices, and infrastructure. In this view, the enterprise conforms to the standard process and the project conforms to the enterprise process. IEEE/EIA 12207 is a key standard for this use.

2 IEEE/EIA 12207, Software Life Cycle Processes

IEEE/EIA 12207, *Software Life Cycle Processes*, benefits from a heritage spanning a quarter-century of work on software process standards. Distant ancestors include the pioneering defense standards better known by number than by name—Mil-Std-1679 and DoD-Std-2167. More recently, a cross-DoD working group developed Mil-Std-498 to replace DoD-Std-7935A, NSA 1703 and DoD-Std-2167A. When DoD policy deprecated the use of defense-specific process standards, a joint working group of the IEEE and the Electronic Industries Association (EIA) converted much of the content of Mil-Std-498 into a commercial standard, EIA/IEEE J-Std-016, *Software Development—Acquirer/Supplier Agreement.*

Meanwhile, ISO/IEC JTC1/SC7 was working on the standard that eventually became ISO/IEC 12207. Although Mil-Std-498 and an existing IEEE process standard, 1074, *Developing Software Life Cycle Processes,* were inputs to the effort, the result is quite different from both of them.

The developers of the ISO/IEC 12207 standard succeeded in developing a standard remarkably different from its predecessors. It establishes a common framework for software throughout its life cycle from conception through retirement and addresses the organizational context of those software processes both from the technical viewpoint of the system and the business viewpoint of the enterprise. The standard is widely regarded as providing a basis for world trade in software services; adoption of the standard is completed or underway in most of the major countries of the world. Next to the ISO 9000 series, it is probably the most important information-technology-related standard ever written.

2.1 Key Concepts of ISO/IEC 12207

The 12207 standard made many important innovations. (Only a few are selected for treatment here.) Most importantly, 12207 is defined at the level of *process* rather than *procedure*. Rather than provide the step-by-step requirements characteristic of a procedure, it describes *continuing responsibilities* that must be achieved and maintained during the life of the process. The standard addresses the *functions* to be performed rather than *organizations* to execute them. (For example, the standard describes a quality assurance process; this does not imply that a conforming enterprise must establish a quality assurance department.) The standard describes the development, maintenance and operation of software within the context of the system, thus effectively establishing the *minimum system context* essential to software processes.

The processes of ISO/IEC 12207 are described in three categories:

---

1 Subcommittee 7 (responsible for software engineering) of Joint Technical Committee 1 (responsible for information technology) formed under a cooperative agreement of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC).

1. *Primary processes* are executed by parties who initiate or perform major roles in the software life cycle. They include the business roles of:

- Acquisition
- Supply

and the technical roles of:

- Development
- Operation
- Maintenance

2. *Supporting processes* contribute to the execution of other processes as an integral part with distinct goals. They include:

- Documentation
- Configuration management
- Quality assurance
- Verification
- Validation
- Joint review
- Audit
- Problem resolution

3. *Organizational processes* inherently exist outside the scope of the individual project but instances of them are employed by the project. They include:

- Management
- Infrastructure
- Improvement
- Training

From the viewpoint of this paper, ISO/IEC 12207 makes many important improvements, but has a key problem—it still addresses conformance at the project level. When the joint IEEE/EIA working group developed the industrial adaptation of 12207, it tackled that problem and added provisions for enterprise-level conformance. The resulting document, IEEE/EIA 12207, is now ready for use.

2.2 Key Improvements made in IEEE/EIA 12207

The joint working group responsible for the IEEE/EIA adoption of the international 12207 standard desired to make some important improvements while retaining a clear relationship to the international standard. They chose to achieve this goal by embedding the text of the international standard within the IEEE/EIA standard. IEEE/EIA 12207.0 (“Part 0” of the 12207 standard) is a “sandwich” with the complete text of ISO/IEC 12207 in the middle and additional material on the outside, as shown at the left side of Figure 1.

The IEEE/EIA project added two additional guidance parts to the standard. Part 1 provides guidance on life cycle data. It is cross-referenced to the provisions of Part 0. Part 2 provides guidance on the implementation of processes. It quotes the complete text of Part 0 and intersperses guidance notes.

The most important improvement in the IEEE/EIA standard is a set of alternatives for conformance with the standard. Rather than addressing only the project, four situations are addressed:

- *Enterprise*: The enterprise adopts the standard and implements policies, procedures and infrastructure to implement the provisions of the standard.
- *Project*: Either the project directly conforms to the standard or the standard conforms to enterprise processes that, in turn, conform to the standard.
- *Multi-supplier program*: A program adopts the standard and, viewed as a whole, conforms to the provisions of the standard regardless of whether individual suppliers qualify for conformance.
- *Regulatory program*: A program conforms to a tailored version of the standard provided by the regulator.

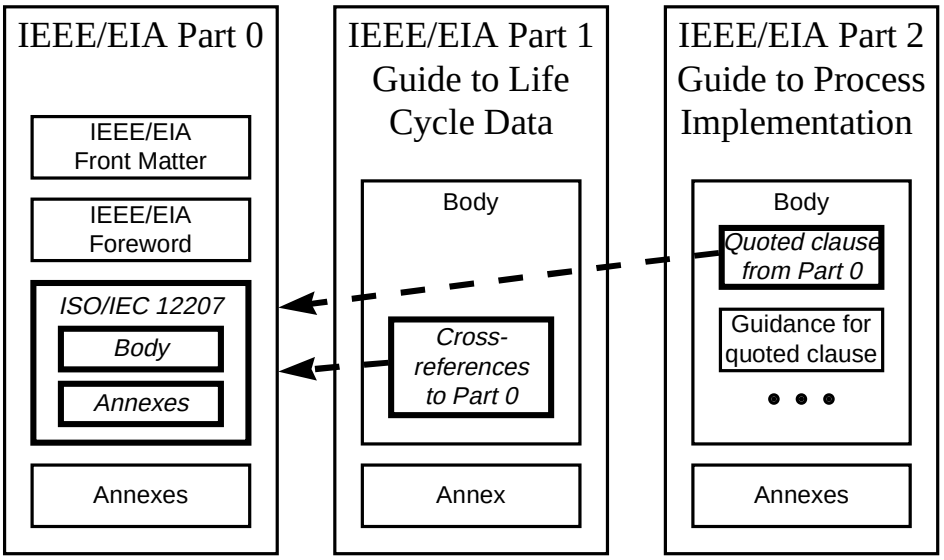


Figure 1. Structure of IEEE/EIA 12207

Part 1 also provides cross-references to other IEEE standards that may be helpful in implementing the provisions concerning data. For instance, a user might choose to adopt IEEE Std 1016, *Software Design Descriptions,* to detail the data provisions related to the information item for software item description. Working in the other direction, the SESC is currently supplementing each of the referenced IEEE standards with a content map describing the extent to which the standard satisfies the data provisions of Part 1; this work will be completed by year-end 1998. Within the next few years, the content of each of the standards will be revised so that it fully conforms to the relevant provisions of Part 1.

In overall terms, SESC has adopted policy designating 12207 as a strategic, integrating standard for its collection. All of the relevant standards of the SESC collection will be revised to improve their fit with 12207; in particular, many of them will serve to detail the processes of the 12207 framework. From the process viewpoint, IEEE/EIA 12207 will serve as a single entry point to all the standards of the IEEE software engineering collection.

3 Contextual Processes and Standards

One of the complications in adopting enterprise-level software processes is that software engineering does not exist in isolation from other information technologies. Apart from the obvious influences of computer science and the various application domains, software engineering exists in the context of more general disciplines such as *project management*, *systems engineering* and *quality management*. Furthermore, software engineering projects are profoundly affected by cross-cutting disciplines like *dependability* and *safety*. Implemented software engineering processes must be consistent with the processes demanded by these other disciplines.

The good news is that SC7 and IEEE SESC are taking the necessary steps to coordinate their collections with other important disciplines. One important example—quality management—is discussed here. (A more extensive treatment of these and other examples can be found in the guide to the SESC collection, [Moore97].)

The ISO 9000 series is the most successful example of an information-technology-related standard in the brief history of the information age. World-wide, more than 135,000 organizations claim conformance [Kiang97], more than 80 countries have adopted them as national standards [Peach94], and more than 50 countries actively participate

in developing the standards [Kiang97]. For an enterprise claiming or desiring ISO 9000 compliance, it makes no sense for their software engineering processes to remain unsupportive of that goal. Nevertheless, until recently, the relationship of software engineering to quality management was described by a guide, ISO 9000-3, whose relationship to the relevant standard, ISO 9001, was murky and whose references to software development were regarded by some as uninformed.

Fortunately, that problem has been solved by the recent publication of the 1997 revision to ISO 9000-3. That guide is now organized identically to ISO 9001; it quotes every relevant passage of 9001 and explains the passage in terms of the requirements levied upon software processes. Furthermore, each clause explicitly references the relevant clauses of ISO/IEC 12207, establishing a firm relationship between software and quality processes. In addition, the guide to the SESC Collection provides additional cross-references to IEEE standards that may be useful in detailing the relationships.

4 Adopting Enterprise Level Software Processes

4.1 Process versus Procedure

Before discussing the enterprise level adoption of software processes, it is important once again to emphasize the distinction between a process and a procedure. I found the most eloquent description of this distinction in an unpublished report of the Rand Corporation:

> A *procedure*...is a series of steps. When you have completed the steps, you have completed the procedure. This is the only guarantee of a procedure. It is unrelated to the ostensible objective of the procedure.
>
> A *process*, on the other hand, has principles that contribute to an objective. Everyone involved must understand and look at the contribution to the objective and use the principles to guide him rather than following them blindly as in a procedure.

The “principles” mentioned in this quotation are what I have termed “continuing responsibilities” elsewhere in this paper. The distinction between process and procedure is at the heart of the fundamental difference between 12207 and its ancestor process standards. The distinction is also fundamental to the framework for process adoption described below.

4.2 Basili Model for Process Descriptions

In performing his work on the “component factory,” Basili needed to provide appropriate process descriptions. He and his colleagues found [Basili92, Heineman94] that three different forms of description seem useful:

- The *Reference* view describes a process as a coherent, cohesive sets of activities that can sensibly be performed by a single *agent*. The 12207 standard fills this role in enterprise adoption.
- The *Contextual* view describes flow of control and data among the *agents*. For enterprise adoption, additional standards may be adopted to detail the processes of 12207 and to fill this role.
- The *Implementation* view maps the *agents* to the organization chart and selects policies, procedures and tools to implement the processes. For enterprise adoption, this role is filled by various corporate standards.

4.3 A Framework for Enterprise Process Adoption

Minding the distinction between process and procedure and applying the Basili model leads to the framework for process adoption depicted in Figure 2.

The first step is to select a life cycle process standard for enterprise adoption. Although IEEE/EIA 12207 is the obvious choice, there are alternatives that may make sense for specific industry sectors including space, nuclear, airborne and legacy defense systems.

The next step is to analyze the needs of the enterprise to identify contextual processes and requirements that may affect software engineering. In performing the analysis, one should look for organizational strengths to build upon as well as organizational weaknesses that should be remedied. One should consider:

- *Organizational imperatives*—for example, ISO 9000 conformance.
- *Organizational competencies*—for example, systems engineering or project management.
- *Industry sector characteristics*—for example, requirements for dependability or safety.

Each of the cited examples has a related body of standards (see [Moore97]) that should be considered when planning enterprise-level processes. Armed with the selected body of standards, one can now describe enterprise level processes at the reference level. In many cases, this description can be accomplished simply by referring to appropriate standards, perhaps with auxiliary explanation of areas where the selected standards may overlap or leave gaps.

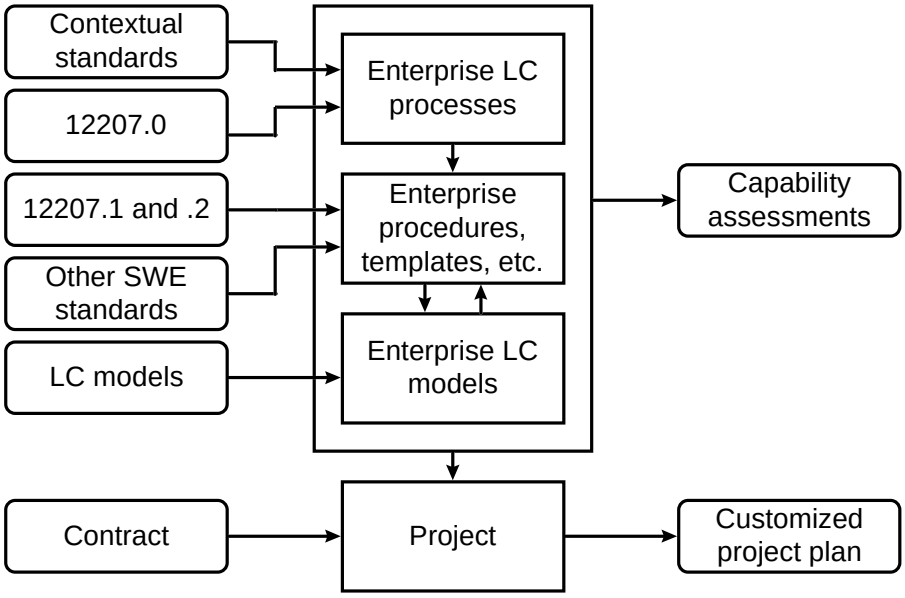


Figure 2. Overview of Enterprise Process Adoption.

The next step is to detail those processes, in part through the selection of additional standards. Parts 1 and 2 of IEEE/EIA 12207 can be helpful in this area. Furthermore, both of those parts reference additional standards that may be helpful. The guide to the SESC collection is intended to support this selection.

IEEE/EIA 12207 and most of the other standards are independent of any particular life cycle model, such as evolutionary development or waterfall. Mapping of enterprise processes to a selected life cycle model may be left to the individual project. Infrastructural support, though, is likely to be more effective if a set of models is selected and implemented at the enterprise level.

Given all of these resources, the real payoff comes from the implementation of procedures, templates, tools, training and other infrastructural support at the enterprise level. It is these resources that enable repeatable execution of processes, portability of staff and the resulting organizational improvement.

Finally, of course, enterprise policies must be established to motivate usage of the enterprise processes, to systematically monitor execution for desirable changes, and to complete the cycle of organizational improvement.

5 Summary

In the past, available software engineering standards have not been designed to support enterprise-level adoption of processes. The new IEEE/EIA 12207 standard breaks new ground by defining software life cycle processes as a set of continuing responsibilities and by permitting claims of conformance to be made at the enterprise level. Besides adopting 12207 as a strategic, integrating standard, the IEEE Software Engineering Standards Committee has fitted its collection of 44 standards into an overall framework and written a guide to describe the organization of the collection.

An organization seeking to implement enterprise level processes should supplement 12207 with other standards supporting contextual goals, such as quality management and dependability. The desired processes can be selectively detailed with additional standards and mapped to an inventory of life cycle models. The provision of additional procedures, templates, and tools enables an enterprise to provide a robust process capability to project managers.

List of References

[Basili92] Basili, Victor R., et al. "A Reference Architecture for the Component Factory." *ACM Transactions on Software Engineering and Methodology* Vol 1 (1992): 53-80.

[Heineman94] Heineman, G. T., et al. "Emerging Technologies that Support a Software Process Life Cycle." *IBM Systems Journal* Vol 33 (1994): 501-529.

[Kiang97] Kiang, David. "ISO 9000 Update." presentation to Society of Reliability Engineers, Ottawa, Canada, 25 Feb 1997.

[Magee97] Magee, Stan, and Leonard L. Tripp. *Guide to Software Engineering Standards and Specifications.* Boston, MA: Artech House, 1997.

[Moore97] Moore, James W. *Software Engineering: A User's Road Map*. Los Alamitos, CA: IEEE Computer Society Press, 1997.

[Peach94] Peach, Robert W., ed. *The ISO 9000 Handbook,* (2nd Ed.). Fairfax, VA: CEEM Information Services, 1994.

IMPROVEMENT OF THE TEST PROCESS
using TPI®

Martin Pol

IQUIP Software Control Testen

P.O. Box 263, 1110 AG Diemen
The Netherlands
Tel: +31 20 6606600
Fax: +31 20 6953298
e-mail: tpi@iquip.nl, polmarti@iquip.nl
English TPI® website: www.iquip.nl/tpi

Abstract

This paper presents the TPI®-model, which is based on current state-of-the-art test process improvement practices. The model gives practical guidelines for assessing the maturity level of testing in an organisation and for step by step improvement of the process. The purpose of such improvement could be reaching CMM level 3.

The model consists of 20 key areas, each with different levels of maturity. The levels of all key areas are set out in a maturity matrix. Each level is described by several checkpoints. Improvement suggestions, which help to reach a desired level, are part of the model.

The paper includes a general description of the application of model, which deals with how to implement and how to consolidate the improvements.

Speaker´s biography

Martin Pol has more than 25 years of experience in the information business, especially in test management. He is co-author of three bestsellers on structured testing. Martin is a respected speaker at conferences and training sessions throughout Europe and in the USA.

1 How good is your test process ?

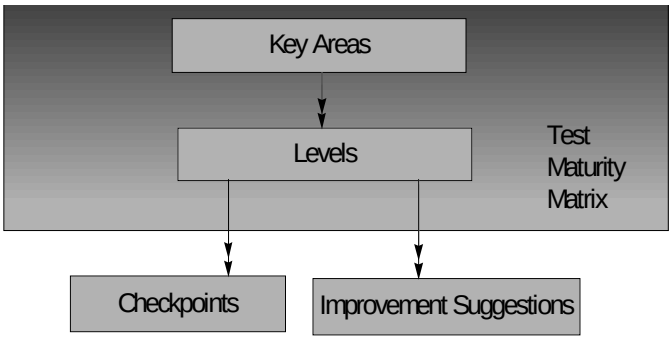
This seemingly easy question turns out to be very hard to answer in reality. Testing is often experienced as a troublesome and uncontrollable process. Testing takes too much time, costs a lot more than planned, and offers insufficient insight in the quality of the test process and, therefore, the quality of the information system under test and the risks for the business process itself. But can we do something about this ?

Many organisations realise that improving the test process can solve these problems. However, in practice it turns out to be hard to define what steps to take for improving and controlling the process, and in what order. A comparison can be made with improvement of the total software process, where models like the Capability Maturity Model© (CMM) offer support.

Based on the knowledge and experiences of a large number of professional testers the Test Process Improvement (TPI®) model has been developed. The TPI® model supports the improvement of test processes. The model offers insight in the "maturity" of the test processes within your organisation. Based on this understanding the model helps to define gradual and controllable improvement steps.

2 Description of the model

The model is visualised as follows:



2.1 Key Areas

In each test process certain areas need specific attention in order to achieve a well defined process. These **Key Areas** are therefore the basis for improving and structuring the test process. The TPI®model has 20 key areas.

The scope of test process improvement usually comprises black-box tests like system and acceptance tests. Most key areas are adjusted to this. However, to improve more "mature" test processes, attention must also be given to verification activities and white-box tests like unit and integration tests. Separate key areas are included in order to give due attention to these processes as well.

A full list of key areas is given below, followed by an explanation.

Test strategy	Test tools	Reporting
Life cycle model	Test environment	Defect management
Moment of involvement	Office environment	White-box testing
Estimating & planning	Commitment and motivation	Evaluation
Test design techniques	Testing functions and training	Test process management
Static test techniques	Scope of methodology	Testware management
Metrics	Communication	

Key Area	Description
Test strategy	Test strategy should be aimed at finding the most important defects as early and as cheap as possible. In the test strategy it is determined what (quality)risks are covered by testing. By involving more tests and more detective measures, a better balanced strategy is possible. Unintentional overlaps or gaps between different tests can be prevented by co-ordinating testers and test activities, and by determining thoroughness.
Life cycle model	Within the test process a number of phases are discerned: planning, preparation, design, execution and completion. In each phase several activities are performed. For each activity aspects like goal, input, process, deliverables, dependencies, techniques and tools, facilities and documentation are recorded. The importance of a life cycle model lies in better control of the test process, since the activities can be planned and controlled consistently.
Moment of involvement	Although the actual test execution usually starts after the realisation of the software, the test process should start a lot earlier. Earlier involvement of testing in the system development life cycle helps detecting defects as early and/or as easy as possible, and even helps preventing defects. Better co-ordination between tests is possible and the critical path time of testing can be greatly reduced.
Estimating & planning	Estimating and planning are required in order to define which activities are performed at what moment and how many resources will be needed. Estimating and planning is the basis for reserving capacity and for co-ordinating test activities and project activities.
Test design techniques	A test design technique is defined as 'a standardised approach for deriving test cases from documentation'. Usage of these techniques increases insight in the quality and coverage of tests and leads to higher re-usability of tests. Based on a test strategy, different test design techniques are used in order to produce test coverage of the intended parts of the software to the extent which was agreed upon.
Static test techniques	Not everything can and needs to be tested dynamically, i.e. by running the programmes. The phenomenon of checking products without running the actual programmes or evaluating specified quality measures, is called static testing. Checklists and similar devices are very useful here.
Metrics	Metrics are quantified observations (measurements). For the test process, measuring the progress and the quality of the software under test is very important and so are metrics in these areas. Metrics are used in order to be able to manage the test process, in order to support advice to be given, and also in order to compare different systems or processes. It helps answering questions such as 'Why is it that system A has far less failures in production than system B has, why is it that test process A can be performed faster and more thoroughly than process B can?' In the improvement of the test process, metrics are specifically important for judging the results of certain improvement actions. This is done by measuring before and after the improvement takes place.
Test tools	Automation of the test process can be done in a variety of ways. As a rule, automation serves one of the following goals: - less resource consumption; - less time consumption; - better test coverage; - more flexibility; - more or faster insight in the status of the test process; - better motivation of test staff.
Testing environment	Test execution takes place in a so called test environment. This environment consists of the following components: • hardware; • software; • communication facilities; • facilities for creation and use of data sets; • procedures. The environment should be arranged so that optimal testing is possible. The environment greatly influences the quality, duration and costs of the test process. Important aspects of the environment are responsibilities, control, timely and sufficient availability, flexibility and representativeness of the actual production situation.
Office environment	Test personnel need offices, desks, chairs, PC's, word processing facilities, printers, telephones, etc. Good and timely arrangement of the office environment positively

	influences motivation of testers, and communication and efficiency of (the execution of) test tasks.
Commitment & motivation	Commitment and motivation of the people involved in testing are conditions for a mature test process. People involved are not only members of the test team, but also, amongst others, project managers and senior management. The test process is supplied with sufficient time, money and resources (both quantitatively and qualitatively) to perform a good test. Co-operation and communication with the others in the project results in an efficient process and in earlier involvement.
Testing functions & training	The test team requires a certain composition. A mixture is needed of different disciplines, functions, knowledge and skills. For example, apart from specific test expertise, also knowledge of the system under test is necessary, knowledge of the organisation and general knowledge of automation. Certain social skills are also very important. In order to get this mixture, education and training is needed.
Scope of methodology	For each test process a certain methodology or approach is used, consisting of activities, procedures, standards, techniques, etc. If these methodologies differ for each test process in the organisation, or if the methodology used is too generic, a lot of things have to be reinvented over and over again. The aim for an organisation is to use a methodology that is sufficiently generic to be widely applicable, but that has enough detail at the same time to be able to prevent undesired reinvention for each new test process.
Communication	In a test process communication takes place in a number of ways, both between testers as a group, and between testers and other members of the project, such as the developer, the end-user, the project manager. Topics of communication are test strategy, progress and quality of the software under test.
Reporting	Testing does not only deal with the detection of defects, but also with giving advice on (the lack of) quality of software. Reporting should be aimed at giving well funded advice to the project and customer on (the quality of) software and even on the software development process.
Defect management	Although defect management is in fact the project's responsibility, testers are strongly involved here. A good administration should be able to control the life cycle of a defect and to create several (statistical) reports. These reports are used to give well funded quality advice.
Testware management	Test products should be maintainable and reusable, and should therefore be managed. Apart from test products, also the products of prior phases, such as design and realisation, have to be managed well (although not by testers!). The test process can be seriously disrupted by delivery of wrong programme versions, etc. The demand of good management of these products, increases testability (and quality) of software.
Test process management	In order to manage each process and each activity, the four steps of the so called Deming circle are essential: plan, do, check, act. A well managed test process is of the utmost importance to perform the best possible test in the often very turbulent test arena.
Evaluation	Evaluation in this context means testing deliverables such as the functional design. In comparison to testing, the advantage of evaluation is the opportunity of early detecting defects. This causes the costs of repair to be considerably lower. Also, evaluation is relatively simple to establish since no programmes have to be run, no environment needs to be composed, etc.
White-box testing	A white-box test is defined as a test of the internal properties of the object, using knowledge of internal functioning. These tests are performed by developers. Well known white-box tests are the unit test and the integration test. Just like evaluation these tests take place earlier in the system development life cycle than black-box tests. Also, white-box tests are relatively cheap because less communication is required and because analysis is easier (the person detecting the defect is often the same person as the one who is to do the repairing. Besides, smaller objects are tested).

2.2 Levels

The way key areas are organised within a test process determines the 'maturity' of the process. It is obvious that not each key area will be addressed equally thoroughly: each test process has its strengths and weaknesses.

In order to enable insight in the state of the key areas, the model supplies them with **Levels** (from A to B to C). On the average, there are three levels for each key area.

Each higher level (C being higher than B, B being higher than A) is better than its prior level in terms of time (faster), money (cheaper) and/or quality (better). By using levels we can unambiguously assess the current situation of the test process. It also increases the ability to advice targets for stepwise improvement.

Each level consists of certain requirements for the key area. The requirements (= checkpoints) of a certain level also comprise the requirements of lower levels: a test process at level B fulfils the requirements of both level A and B. If a test process does not satisfy the requirements for level A, it is considered to be at the lowest and, consequently, undefined level for that particular key area.

Below a description is given of the different levels of the key areas.

Levels Key Area	A	B	C	D
Test strategy	Test strategy for single test	Combined test strategy for black-box tests	Combined strategy for black-box tests plus white-box tests or evaluation	Combined strategy for all test and evaluation activities
Life cycle model	Planning, Design, Execution	Planning, Preparation, Design, Execution, Completion		
Moment of involvement	Completion of specification	Start of specification	Start of requirements definition	Project initiation
Estimating and planning	Fundamental estimating & planning	Statistically founded estimating & planning		
Design techniques	Informal techniques	Formal techniques		
Static test techniques	Intake test basis	Checklists		
Metrics	Project statistics (product)	Project statistics (process)	System statistics	Organisation statistics
Test tools	Planning & control tools	Test execution & analysis tools	Integrated test automation	
Test environment	Managed and controlled environment	Testing in most suitable environment	Environment on call	
Office environment	Adequate & timely office environment			
Commitment and motivation	Assignment of budget & time	Testing integrated in project organisation	Test engineering	
Test functions and training	Test manager and testers	Support (methodical, technical, functional), control	Internal Quality Assurance	
Scope of methodology	Project specific	Organisation, generic	Organisation, optimising (R&D)	
Communication	Internal communication	Project communication (defects, change control)	Communication in organisation	
Reporting	Defects	Progress (status of tests and products), activities (costs + time, milestones), defects with priorities	Risks & advice, including statistics	SPI advice
Defect management	Internal defect management	Extended defect management, flexible reporting facilities	Project defect management	
Testware management	Internal management & control of test deliverables	External management & control of test basis and test object	Reusable testware	Traceability: from requirements to test cases
Test process management	Plan, do	Plan, do, check, react	Check, react in organisation	
Evaluation	Evaluation techniques	Evaluation strategy		
White-box testing	Life-cycle: Planning, Design, Execution	White-box design techniques	White-box test strategy	

2.3 Checkpoints

In order to determine levels, the TPI®-model is supported by an objective measurement instrument. The requirements for each level are defined in the form of **Checkpoints**: questions that need to be answered positively in order to classify for that level. Based on the checkpoints a test process can be assessed, and for each key area the proper level can be established. As each next level of a key area is considered an improvement, this means that the checkpoints are cumulative: in order to classify for level B the test process needs to answer positively to the checkpoints both of level B and of level A.

2.4 Test Maturity Matrix

After determining the levels for each key area, attention should be directed as to which improvement steps to take. This is because not all key areas and levels are equally important. For example, a good test strategy (level A of key area Test Strategy) is more important than a description of the test methodology used (level A of key area Scope of Methodology). In addition to these priorities there are dependencies between the levels of different key areas. Before statistics can be gathered for defects found (level A of key area Metrics), the test process has to classify for level B of key area Defect management. Such dependencies can be found between many levels and key areas.

Therefore, all levels and key areas are related to each other in a **Test Maturity Matrix**. This has been done as a good way to express the internal priorities and dependencies between levels and key areas. The vertical axis of the matrix indicates key areas, the horizontal axis shows scales of maturity. In the matrix each level is related to a certain scale of test maturity. This results in 13 scales of test maturity. The open cells between different levels have no meaning in themselves, but indicate that achieving a higher maturity for a key area is related to the maturity of other key areas. There is no gradation between levels: as long as a test process is not entirely classified at level B, it remains at level A.

Scale **Key Area**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Test strategy		A					B				C		D	
Life cycle model		A			B									
Moment of involvement			A				B				C		D	
Estimating and planning				A							B			
Test design techniques		A		B										
Static test techniques					A		B							
Metrics						A			B			C		D
Test tools					A			B			C			
Test environment				A				B						C
Office environment				A										
Commitment and motivation		A				B						C		
Test functions and training				A			B			C				
Scope of methodology					A						B			C
Communication			A		B							C		
Reporting		A			B		C					D		
Defect management		A				B		C						
Testware management			A			B				C				D
Test process management		A		B								C		
Evaluation							A			B				
White-box testing					A		B		C					

The main purpose of the matrix is to show the strong and weak sides of the current test process and to support prioritising actions for improvement. A filled in matrix offers all participants a clear view of the current situation of the test process. Furthermore, the matrix helps in defining and selecting proposals for improvement.

The matrix works from left to right, so low mature key areas are improved first. As a consequence of the dependencies between levels and key areas, practice has taught us that real 'outlyers' (i.e., key areas with high scales of maturity, whereas surrounding key areas have medium or low scales) give little return on investment. For example, what is the use of a very advanced defect administration, if it is not used for analysis and reporting? Without violating the model, deviation is permitted, but sound reasons should exist for it.

In the example below, the test process does not classify for the lowest level of the key area test strategy(level < A), the organisation is working conform a life cycle model (level A) and the testers are involved at the moment when the specifications are completed (level A).

Scale **Key Area**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Test strategy		A					B				C		D	
Life cycle model		A			B									
Moment of involvement			A				B				C		D	
etc.														

Based on this instance of the matrix, improvements can be discussed. In this example, a choice is made for a combined test strategy for black-box tests (=> level B) and for a full life cycle model (=> level B). Earlier involvement is at this moment not considered to be of relevance. The required situation is represented in the following matrix.

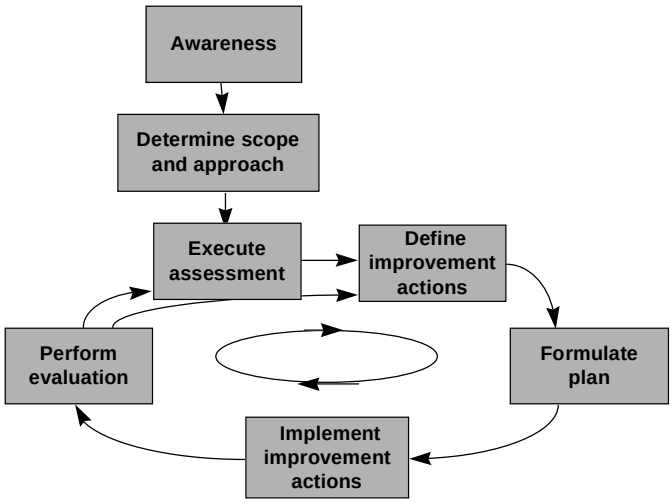
Scale **Key Area**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Test strategy		A					B				C		D	
Life cycle model		A			B									
Moment of involvement			A				B				C		D	
etc.														

2.5 Improvement Suggestions

Improvement actions can be defined in terms of desired higher levels. For reaching a higher level the checkpoints render much assistance. Beside these, the model has other means of support for test process improvement: the **Improvement Suggestions**, which are different kinds of hints and ideas that help to achieve a certain level of test maturity. Unlike the use of checkpoints, the use of improvement suggestions is not obligatory. Each level is supplied with several improvement suggestions.

3 Application of the TPI® model

The process of test improvement is similar to any other improvement process. The figure below shows the various activities of an improvement process. These activities are discussed, with special attention for the places where the TPI® model can be used.



Awareness
The first activity of a test improvement process is to create awareness for the necessity to improve the process. Generally speaking, a number of problems concerning testing is the reason for improving the test process. There is a need to solve these problems and an improvement of the test process is regarded as the solution. This awareness also implies that the parties mutually agree on the outlines and give their commitment to the change process. Commitment should not only be acquired at the beginning of the change process, but be retained throughout the project. This requires a continuous effort.

Determine scope and approach
We determine what the improvement targets are and what the scope is. Should testing be faster, cheaper or better? Which test processes are subjects for improvement, how much time is available for the improvement and how much effort is it allowed to cost?

Execute assessment
In the assessment activity, an evaluation is given of the current situation. The use of the TPI® model is an important part of the assessment, because it offers a frame of reference to list the strong and weak points of the test process. Based on interviews and documentation, the levels per key area of the TPI® model are examined by using checkpoints, and it is determined which checkpoints were met, which were not met, or only partially. The Test Maturity Matrix is used here to give the complete status overview of the test process. This will show the strengths and weaknesses of the test process in the form of levels assigned key areas and their relative position in the matrix.

Define improvement actions
The improvement actions are determined based on the improvement targets and the result of the assessment. These actions are determined in such a way that gradual and step by step improvement is possible.

The TPI® model helps to set up these improvement actions. The levels of the key areas and the Test Maturity Matrix give several possibilities to define gradual improvement steps. Depending on the targets, the scope, the available time and the assessment results, it can be decided to carry out improvements for one or more key areas. For each selected key area it can be decided to go to the next level or, in special cases, even to a higher level. Besides this, the TPI® model offers a large number of improvement suggestions which help to achieve higher levels.

Formulate plan
A detailed plan is drawn up to implement (a part of) the short term improvement actions. In this plan the aims are recorded and it is indicated which improvements have to be implemented at what time to realise these aims. The plan deals with activities concerning the content of the test process improvement as well as general activities needed to steer the change process in the right direction.

Implement improvement actions
The plan is executed. Because during this activity the consequences of the change process have the largest impact, much attention should be spent on communication. Opposition, which no doubt is present, must be brought to the surface and be discussed openly.
It has to be measured to what extent actions have been executed and have been successful. A means for this is the so-called "self assessment", in which the TPI® model is applied in order to quickly determine the progress. Here, the persons involved inspect their own test processes using the TPI® model.
Another vital part of this phase is consolidation. It should be prevented that the implemented improvement actions have a once-only character.

Perform evaluation
To what extent did the implemented actions yield the intended result? In this phase the aim is to see to what extent the actions were implemented successfully as well as to evaluate to what extent the initial targets were met. A decision about the continuation of the change process is made based on these observations.

4 Conclusions and remarks

Current developments proceed at a very high speed. The productivity of developers is rising continuously and the customers demand ever higher quality. Even if your current test process is fairly satisfactory, your process will need to improve in the future. The TPI®-model can help you with this.
The TPI®-model is an objective means to gain quick insight in the current situation of the test process. The model greatly offers help for improvement in the form of key areas, levels and improvement suggestions. It supports the definition of small and controlled improvement steps, based on priorities.
The reader might get the impression that use of the TPI®-model automatically leads to good analysis of the current and required situation. This is not true. The model should be seen as a tool for structuring the improvement of the test process and as a very good means of communication. Apart from the tool, improvement of test processes demands a high degree of knowledge and expertise of people involved, at least in the areas of testing, organisation and change management.

Book:
Koomen, T. and M. Pol (1998), Test Process Improvement®, Leidraad voor stapsgewijs beter testen, Kluwer BedrijfsInformatie, ISBN 90 267 27720 (the book is written in Dutch, the English translation will be published end of 1998)

Internet:
At 'www.iquip.nl/tpi' several TPI®-products can be viewed and downloaded. Also questions can be asked and remarks can be made.

Appendix TPI® checkpoints

The following table contains the checkpoints for each level. The left column shows whether the row is a key area, a level or a checkpoint:

[number] = Key Area
[number].[A/B/C/D] = Level
[number].[A/B/C/D].[number] = Checkpoint

Nr.	Key Area / *Level* / Checkpoint
1	**Strategy and scope**
1.A	*Testing strategy for single test*
1.A.1	A motivated evaluation of risks takes place, for which knowledge about the system and knowledge about the usage and the management of the system is essential.
1.A.2	There is a differentiation in the depth of the tests, depending on the risks and, if present, depending on the acceptance criteria. Not all subsystems are tested equally and not all quality attributes are tested (equally).
1.A.3	One or more formal or informal test design techniques, suitable for the desired depth of the test, are used.
1.A.4	For re-tests a (simple) strategy determination takes place as well, in which a motivated choice is made between 'solely test solved problems' and 're-test completely'.
1.B	*Combined testing strategy for black-box tests*
1.B.1	The various black-box test types, often the system test, the acceptance test and the production acceptance test, are adjusted to each other regarding test strategy (risks, quality attributes, scope, and planning).
1.B.2	The result of the adjustment is an overall strategy, which is laid down in writing. During the entire test process this strategy is monitored.
1.B.3	Based on the adjustment each black-box test type determines its own test strategy as described in level A.
1.B.4	Deviations to the overall strategy are reported, after which a founded adjustment of the overall strategy is made based on the risks involved.
1.C	*Combined strategy for black-box tests plus white-box tests or evaluation*
1.C.1	The various black box test <levels> and the white-box test <levels> or the evaluation <levels>, are adjusted to each other regarding test strategy (risks, quality attributes, scope, and planning).
1.C.2	Based on the adjustment, for each black-box test <level> a dedicated test strategy is determined as described in level A.
1.C.3	Based on the adjustment, for each white-box test <level> (if applicable) a dedicated test strategy is determined as described in key area 'White-box testing', level C.
1.C.4	Based on the adjustment, for each evaluation <level> (if applicable) a dedicated test strategy is determined as described in key area Evaluation, level B.
1.C.5	The result of the adjustment is an overall strategy, which is laid down in writing. During the entire (evaluation and) test process this strategy is guarded.
1.C.6	Deviations to the overall strategy are reported, after which a founded adjustment of the overall strategy is made based on the risks involved.
1.D	*Combined strategy for all testing and evaluation activities*
1.D.1	The various black-box test <levels>, the white-box test <levels>, as well as the evaluation <levels> are adjusted to each other regarding test strategy (risks, quality attributes, scope, and planning).
1.D.2	Based on the adjustment, for each black-box test <level> a dedicated test strategy is determined as described in level A.
1.D.3	Based on the adjustment, for each white-box test <level> (if applicable) a dedicated test strategy is determined as described in key area 'White-box testing', level C.
1.D.4	Based on the adjustment, for each evaluation <level> (if applicable) a dedicated test strategy is determined as described in key area Evaluation, level B.
1.D.5	The result of the adjustment is an overall strategy, which is laid down in writing. During the entire (evaluation and) test process this strategy is guarded.
1.D.6	Deviations to the overall strategy are reported, after which a founded adjustment of the overall strategy is made based on the risks involved.
2	**Life-cycle Model**
2.A	*Planning, Design, Execution*
2.A.1	For the test (at least) the following phases are recognised: planning, specification, and execution. These phases are performed successively, possibly per subsystem.
2.A.2	Activities to be executed per phase are:
2.A.3	describe assignment, establish the test basis, determine test strategy, set up organization, define test products, define infrastructure and tools, set up management, and determine planning (phase Planning);
2.A.4	define logical and physical test cases, realization of the test infrastructure (phase Specification);
2.A.5	intake test object and test infrastructure, populate base files, and test execution (phase Execution).
2.B	*Planning, Preparation, Design, Execution, Completion*
2.B.1	For the black box tests the following phases are recognised as well: preparation and completion. The phases are performed successively, possibly per subsystem.
2.B.2	Activities to be executed per phase are:
2.B.3	detailed intake (check if the test basis is suitable for the selected test specification techniques) (phase Preparation)
2.B.4	conserve the testware (complete en bring the testware up to date, in a way that the testware is re-usable for other test processes), evaluate the test process and test object, and deliver a final report (phase Completion).
3	**Moment of involvement**
3.A	*Completion of specification*
3.A.1	The activity 'testing' starts simultaneously with or before the completion of the test basis of a definite part of the system that can be independently tested.
3.A.2	(The system can be divided in several parts that are independently completed, realised, and tested. The tests for the first subsystem have to start simultaneously with or before the completion of the test basis of that specific subsystem.
3.B	*Start of specification*

3.B.1	The activity 'testing' starts simultaneously with or before the phase in which the test basis is prepared (often the functional design phase).
3.C	*Start of requirements definition*
3.C.1	The activity 'testing' starts simultaneously with or before the phase in which the requirements are defined.
3.D	*Project initiation*
3.D.1	When the project is initiated, the activity 'testing' starts as well.
4	**Estimating & planning**
4.A	*Fundamental estimating & planning*
4.A.1	The test budget and test planning can be founded (so not only 'the previous project was also performed like this').
4.A.2	During the test process the budget and planning are guarded and adjusted if necessary.
4.B	*Statistically founded estimating & planning*
4.B.1	Statistical data regarding the progress and quality are structurally collected (see Metrics, level B) for several, comparable projects.
4.B.2	This data is used for founding the test budget and test planning.
5	**Design techniques**
5.A	*Informal techniques*
5.A.1	The test cases are defined according to a documented technique.
5.A.2	The technique at least consists of: a) start situation, b) change process = test actions to be performed, c) expected end result.
5.B	*Formal techniques*
5.B.1	Besides informal techniques also formal techniques are used during the black-box tests, through which the test basis is transformed into test cases in a clear way.
5.B.2	It is possible to make a founded statement regarding the level of coverage of the test set (in relation to the test basis).
5.B.3	The testware is transferable (within the test team) as a result of the uniform method.
6	**Static testing techniques**
6.A	*Intake test basis*
6.A.1	Before defining the test cases an intake is performed on the testability of the test basis.
6.A.2	During this intake checklists are used.
6.B	*Checklists*
6.B.1	Other static tests than the intake take place using checklists (that are approved by the project and/or customer)
7	**Metrics**
7.A	*Project statistics (product)*
7.A.1	During the (test) project input statistics are collected:
7.A.2	- resources used (hours),
7.A.3	- performed activities (hours and lead-time),
7.A.4	- scale and complexity of the system under test (function points / number of functions / realization effort).
7.A.5	During the project output statistics are collected:
7.A.6	- test products (specifications / test cases / log reports),
7.A.7	- test progress (performed tests, completed/not completed),
7.A.8	- number of remarks (remarks per test type / remarks per subsystems / remarks per cause / priority / status / and so on).
7.A.9	The statistics are used in test reports.
7.B	*Project statistics (process)*
7.B.1	During the (test) project result measurements are performed for at least two of the statistics below:
7.B.1.1	defect location effectiveness:
7.B.1.1.a	- percentage of defects found in relation to the total number of defects; the latter is hard to measure but think of the number of defects found in future tests or during the first months the system is operational.
7.B.1.1.b	- analyze in which test the defects should have been found (this says something about previous tests!).
7.B.1.2	defect location efficiency:
7.B.1.2.a	- defects found per hour spent.
7.B.1.3	level of test coverage:
7.B.1.3.a	- depth of test as a percentage of the test goals that are covered by a test case in relation to the number of possible test goals. These goals can be determined for functional specifications as well as for software (e.g. statement coverage).
7.B.1.4	findings on testware:
7.B.1.4.a	- percentage of test defects in relation to the total number of defects.
7.B.1.5	perception of quality
7.B.1.5.a	- through reviews and interviews with users, testers and other people involved.
7.B.2	The statistics are used in the test reports.
7.C	*System statistics*
7.C.1	The statistics mentioned above are collected for new development projects and for maintenance on operational systems.
7.C.2	The statistics are used for to evaluate the effectiveness and efficiency of the test process.
7.D	*Organization statistics*
7.D.1	Throughout the organization comparable statistics are collected for the data mentioned before.
7.D.2	The statistics are used to evaluate the effectiveness and efficiency of separate test processes, in order to optimize the generic test method and future test processes.
8	**Test tools**
8.A	*Planning & control tools*
8.A.1	Automated tools (other than standard word processors) are used for the administration of findings and or at least two other activities for planning and management.
8.B	*Test execution and analysis tools*
8.B+	Record & Playback
8.B+	Load & Stress

8.B+	Test coverage
8.B+	Test data generator
8.B+	Simulators
8.B+	Drivers en Stubs
8.B+	Compiler
8.B+	Comparator
8.B+	Static Analyzer
8.B+	Query languages
8.B+	Debugger
8.B+	Monitor
	U&A-tools
8.B.1	At least two types of automated test tools are used for test execution, like record and playback tools, test coverage tools, and so on.
8.B.2	The test team has a broad insight in the cost and benefit relation of these tools.
8.C	*Integrated test automation*
8.C+	Case tool analyzer
8.C+	Test design
8.C.1	Automated tools (other than standard word processors) are used for the phases planning and management (for all five activities) and for the phases specification and execution (at least five types of tools).
8.C.2	The test team has insight in the cost and benefit relation of these tools.
9	**Testing environment**
9.A	*Managed and controlled test environment*
9.A.1	Adjustments to and deliveries in the test environment only take place with permission of the test manager.
9.A.2	The environment is set up in time. Setbacks are taken into account as much as possible
9.A.3	The test environment is managed (set up, availability, maintenance, version control, problem control, authorization, and so on)
9.A.4	Conserving / putting back certain test situations can be arranged quick and simple.
9.A.5	The environment is sufficiently representative for the tests to be executed, so the further in the test project, the more production-like the test environment must be.
9.B	*Testing in most suitable environment*
9.B.1	Each test (type) is performed in the most suitable environment, by executing in a different environment or by being able to adjust the own environment in a quick and simple way.
9.B.2	The environment is set up in time for the test and during the test there is no disturbance through other activities.
9.B.3	The risks involved are analyzed and effective measures have been taken.
9.C	*Environment on call*
9.C.1	The environment that is most suitable for a test is very flexible and can quickly be adjusted to changing demands.
10	**Office environment**
10.A	*Adequate & timely office environment*
10.A.1	The office infrastructure necessary for testing (workrooms, meeting rooms, telephones, PC's, network connections, office software, printers, and so on) are arranged in time.
10.A.2	Matters regarding the office infrastructure have a minimal impact on the course of the test process (so as little moving as possible, the testers are not physically separated from each other and from the rest of the project team, and so on).
11	**Commitment & motivation**
11.A	*Assignment of budget and time*
11.A.1	Testing is considered a necessity and important by all people involved.
11.A.2	An amount of time and budget are allocated to testing.
11.A.3	Management manages the tests based on time and money. Characteristically, in case of exceeding the planned test time or budget, a solution is usually looked for within the test project (overtime / deploy extra people in case of exceeding time / cutting ti
11.A.4	Within the team there is sufficient knowledge and experience regarding testing.
11.A.5	There are function descriptions of the various test functions within a test team.
11.A.6	The activities for testing are full-time for most of the participants (so not too many conflicts with other activities).
11.A.7	There is a good understanding between testers and other disciplines within the project or the organization.
11.B	*Testing integrated in project organization*
11.B.1	All people involved are convinced that testing has a clear perceptible positive influence on the quality of the product.
11.B.2	Management wants to have insight in the depth and quality of the tests.
11.B.3	Management manages testing based on time, money, and quality. Characteristically solutions for test problems (like exceeding time or budget) are also looked for outside the test project. In this case the person responsible for realization can also be addr
11.B.4	The test process is executed in phases (at least level A of 'Utilization of phasing').
11.B.5	In the project planning the cycle testing, fixing and re-testing is taken into account.
11.B.6	Testing influences the order of delivery of the person responsible for realization.
11.B.7	Advises from testing are discussed during project meetings.
11.C	*Test engineering*
11.C.1	Testing gets involved at least at the moment that the requirements are defined.
11.C.2	At the design and realization stage the test team is involved to obtain an optimal testability of the system ('design for test').
11.C.3	The test team has sufficient and skills to give both check points above a useful contents.
11.C.4	The organization and/or project seriously consider the recommendations of the test team.
11.C.5	Management supports testers (with people and means) to continuously work on improving the test process.
11.C.6	Participating in testing is considered a promotion; testing has a high standing.
11.C.7	The development process is sufficiently mature: management of at least time and quality takes place.
11.C.8	Test functions are described at organization level, including career opportunities and a suitable salary.
12	**Testing functions & training**

12.A	*Test manager and testers*
12.A.1	The test personnel are composed of at least a test manager and several testers.
12.A.2	The tasks and responsibilities are described.
12.A.3	The test personnel has followed specific test training (e.g. test management, test techniques, and so on) or sufficient experience in the area of testing.
12.A.4	People with knowledge of matter are available for the test team for the acceptance test.
12.B	*Support (methodical, technical, functional), control, and intermediary*
12.B.1	The task methodical support is fulfilled separate and consists of the activities defining and maintaining the test rules, the test procedures and test techniques, and checking if they are used correctly.
12.B.2	The task technical support is fulfilled separate.
12.B.3	The task functional support is fulfilled separate.
12.B.4	The task manage test process is fulfilled separate. This task has the responsibility for the registration, the storage, and the availability of all management objects of the test process.
12.B.5	The task manage testware is fulfilled separate and has the responsibility for the registration, the storage and the availability of all management objects of the test process. Sometimes by managing them itself, in other cases by setting up that management
12.B.6	The task manage test infrastructure is fulfilled separate and has the responsibility for the registration, the storage and the availability of all management objects of the test process.
12.B.7	The people who perform the tasks above have sufficient knowledge and experience.
12.B.8	The time necessary for the tasks above is planned. It is checked if these tasks are performed.
12.C	*Internal Quality Assurance*
12.C.1	An internal QA plan for testing is prepared parallel to the test plan.
12.C.2	The person with the QA task has no other tasks within the test team.
12.C.3	The results of QA activities are used as input for future test process improvements.
12.C.4	The person, who fulfills the QA task, has sufficient QA knowledge and experience.
13	**Scope of Methodology**
13.A	*Project specific*
13.A.1	The method is defined for each project.
13.A.2	The described aspects consist at least of: - a description of the complete phasing of the tests, - management of the test process (progress and quality), - test product management, - specification techniques to be used.
13.A.3	The method is followed.
13.B	*Organization, generic*
13.B.1	The method is laid down in a generic model for the organization.
13.B.2	Each project is executed according to this model.
13.B.3	Deviations are sufficiently argued and documented.
13.C	*Organization, optimizing (R&D)*
13.C.1	A structured process of feedback on the generic model exists.
13.C.2	Structural maintenance and innovation (R&D) takes place on the generic model, among other based on the feedback.
14	**Communication**
14.A	*Internal communication*
14.A.1	A periodical meeting within the test team takes place. This consultation has a fixed agenda and is mainly focused on the progress (lead time and hours spent) and the quality of the test object.
14.A.2	Periodically every team member takes part in the meeting.
14.A.3	Deviations to the test plan are communicated and laid down in writing.
14.B	*Project communication (defects, change control)*
14.B.1	Minutes are made of the internal test meetings.
14.B.2	The quality of the test process is, next to the progress and quality of the test object, a subject on the fixed agenda.
14.B.3	In the project meeting the test manager periodically reports regarding the progress and the quality of the test object, including the risks. The test manager reports also regarding the quality of the test process.
14.B.4	Agreements in this meeting are laid down in writing.
14.B.5	The test manager is timely notified with regards to changes in the planned and agreed delivery dates (of both the test basis and the test object).
14.B.6	During a periodical findings meeting (also called analysis forum) test results are discussed between representatives of the test team and other parties involved.
14.B.7	Testing is involved in change control judge the impact of change requests with respect to effort.
14.C	*Communication in organization*
14.C.1	A periodical meeting is held in which proposals for improvement of the used test method and the test processes are discussed.
14.C.2	Participants are representatives of the test teams and representatives of the test department.
15	**Reporting**
15.A	*Defects*
15.A.1	Findings are reported periodically, divided into solved and open findings.
15.B	*Progress (status of tests and products), activities (cost + time, milestones), defects with priorities*
15.B.1	The findings are reported, classified in severity categories according to clear and objective standards.
15.B.2	The progress of every test activity is reported in writing. Aspects about which is reported are: lead time, hours spent, what is specified, what is tested, what went correct, what did not go correct, subject still to be tested.
15.C	*Risks & advice, including statistics*
15.C.1	A quality judgment is given about the test object. This judgment is based on acceptance criteria, if present, and is related to the test strategy.
15.C.2	Possible trends with regards to progress and quality are spotted.
15.C.3	Reports contain risks (for company management) and recommendations.
15.C.4	The quality judgment and the observed trends are founded with statistics (of the finding administration and the progress control).
15.D	*SPI advice*
15.D.1	Besides on the area of testing advise is also given regarding other project parts.

16	**Defect management**
16.A	*Internal defect management*
16.A.1	The various stages of the life cycle of the findings are administrated (up to and including re-testing).
16.A.2	The following items of the finding are registered:
16.A.2*	- unique number
16.A.2*	- person who registered the finding
16.A.2*	- date
16.A.2*	- severity category
16.A.2*	- problem description
16.A.2*	- status
16.B	*Extended defect management, flexible reporting facilities*
16.B.1	The following items of the finding, which are necessary for future trend analysis, are registered:
16.B.1*	- test case
16.B.1*	- test
16.B.1*	- subsystem
16.B.1*	- priority (blocking Y/N)
16.B.1*	- program plus version
16.B.1*	- test basis plus version
16.B.1*	- cause (supposed + conclusive)
16.B.1*	- all status transitions of the finding including dates
16.B.1*	- a description of the problem solution
16.B.1*	- version of the test object and/or the object in which the finding is solved
16.B.1*	- problem solver
16.B.2	The administration makes extensive reporting possible, so listings can be selected or sorted in different ways.
16.B.2	One person sees to it that findings are registered correctly and consistently.
16.C	*Project defect management*
16.C.1	The finding administration is used integrally within the project. Findings come from various disciplines; the people who solve the findings register their solution in the administration themselves, and so on.
16.C.2	By means of authorization every user can only perform the task he or she is permitted.
17	**Testware management**
17.A	*Internal management and control of testing deliverables*
17.A.1	The testware (test cases, base files, and so on), test basis, test object, test documentation and test rules are managed internally according to a described procedure, with steps for delivering, registering, archiving, and inquiring.
17.A.2	Management covers the relations between the various parts (test basis, test object, testware, and so on).
17.A.3	Transfer to the test team takes place according to a fixed procedure. The elements of the transfer must be known: which parts /versions of the test object, which (version) test basis, solved defects, open defects, incl. those registered by the developer.
17.B	*External management and control of test basis and test object*
17.B.1	The test basis and test object (often design and software) is managed by the project according to a described procedure, with steps for delivering, registering, archiving, and inquiring.
17.B.2	Management covers the relations between the various parts (test basis, test object, testware, and so on).
17.B.3	The test team is timely notified regarding changes in test basis or test object.
17.C	*Re-usable testware*
17.C.1	The test products (agreed upon in advance) are finalised (= complete and up to date) after completion of the test and transferred to the maintenance department, after which the transfer is formally approved.
17.C.2	The transferred test products are really re-used.
17.D	*Traceability: from requirements to test cases*
17.D.1	Each system requirement and specification is clearly related to one or more test cases and vice versa.
17.D.2	The relations are monitored at the level of separate versions (e.g. system requirement A (version) related to FD B (version) related to programs C and D (version) related to test cases (version).
18	**Test process management**
18.A	*Plan, do*
18.A.1	Preceding the actual test activities a test plan is prepared, in which all activities that need to be executed are mentioned. Per activity the time frame, the required resources, and the deliverables are mentioned.
18.B	*Plan, do, check, react*
18.B.1	The execution of all planned activities is monitored.
18.B.2	Each activity is also monitored in terms of time and money.
18.B.3	Deviations are laid down in writing.
18.B.4	In case of deviations adjustments are carried out, either by adjusting the plans, or by executing certain activities afterwards.
18.C	*Check and react in the organization*
18.C.1	The application of the test methodology (methods/standards/techniques/procedures) is monitored at organizational level.
18.C.2	Deviations are laid down in writing and are reported to the test process.
18.C.3	In case of deviations the risks are analyzed and adjustments are carried out, e.g. by adjusting the methodology or by letting the activities/products comply with the methodology afterwards. The adjustment is founded.
19	**White-box testing**
19.A	*Life cycle: planning, design, and execution*
19.A.1	(at least) the following phases are identified for the test: planning, design, and execution. These phases are executed successively, possibly per subsystem.
19.A.2	Activities to be executed per phase are:
19.A.3	- formulating the assignment, establish the test basis, set up the organization, set up management, and determine the planning (phase planning and control);
19.A.4	- define test cases (phase design);
19.A.5	- populate base files and test execution (phase execution).
19.B	*White-box design techniques*
19.B.1	Besides informal techniques also formal techniques are used during white-box tests, through which it is

	possible to derive test cases from the test basis in a clear way.
19.B.2	For the white-box tests a founded statement regarding the level of coverage of the test set (compared to the test basis).
19.B.3	The testware is transferable (within the test team) as a result of the uniform procedure.
19.C	*White-box teasing strategy*
19.C.1	A motivated evaluation of risks takes place, for which knowledge of the system and of use and management of the system is essential.
19.C.2	There is a differentiation in the scope and the depth of the tests, depending on the possible risks and, if present, depending on the acceptance criteria: not all types of software are equally tested and not every quality attribute is equally tested.
19.C.3	One or more formal or informal test design techniques are used, suitable for the desired depth of a test.
19.C.4	For re-tests also a (simple) strategy determination takes place, in which a motivated choice is made between 'solely test solutions' and 're-test completely'.
19.C.5	The strategy is defined and afterwards also executed. It is monitored that the tests are executed according to the strategy and, if necessary, the execution will be adjusted.
20	**Evaluation**
20.A	*Evaluation techniques*
20.A.1	Techniques are used for evaluating (intermediate) products, in other words a formal and described procedure is used.
20.A.2	The evaluations and results are reported.
20.A.3	The handling of the results is monitored.
20.A.4	Testers are involved in evaluations.
20.B	*Evaluation strategy*
20.B.1	A conscious evaluation of risks takes place.
20.B.2	There is a differentiation in the scope and the depth of the tests, depending on the possible risks and, if present, depending on the acceptance criteria: not all types of software are equally evaluated and not every quality attribute is equally evaluated
20.B.3	A choice is made from multiple evaluation techniques, suitable for the desired depth of an evaluation.
20.B.4	For re-evaluations a (simple) strategy determination takes place, in which a conscious choice is made between 'solely test solutions' and 're-test completely'.
20.B.5	The strategy is defined and afterwards also executed. It is monitored that the tests are executed according to the strategy and, if necessary, the execution will be adjusted.

View Slides

Using the Software CMM® in Small Organizations

Mark C. Paulk

Abstract

The Capability Maturity Model℠ for Software developed by the Software Engineering Institute has had a major influence on software process and quality improvement around the world. Although the CMM has been widely adopted, there remain many misunderstandings about how to use it effectively for business-driven software process improvement, particularly for small organizations and small projects. Some of the common problems with interpreting the Software CMM for the small project/organization include:

- What does "small" mean? In terms of people? Time? Size of project? Criticality of product?
- What are the CMM "requirements"? Are there key process areas or goals that should not be applied to small projects/organizations? Are there "invariants" of good processes?
- What are the drivers and motivations that cause abuse of the CMM?

This paper discusses how to use the CMM correctly and effectively in any business environment, with examples for the small organization. The conclusion is that the issues associated with interpreting the Software CMM for the small project or organization may be different in degree, but they are not different in kind, from those for any organization interested in improving its software processes. Using the Software CMM effectively and correctly requires professional judgment and an understanding of how the CMM is structured to be used for different purposes.

MARK C. PAULK
Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213
Telephone: +1 (412) 268-5794
Fax: +1 (412) 268-5758
Internet: mcp@sei.cmu.edu

Mark is a Senior Member of the Technical Staff at the Software Engineering Institute. He has been with the SEI since 1987, initially working with the Software Capability Evaluation project. Mark has worked with the Capability Maturity Model project since its inception and was the project leader during the development of Version 1.1 of the Software CMM. He is also actively involved with software engineering standards, including

- ISO 15504 (aka SPICE -- Software Process Improvement and CapabilitydEtermination), an emerging suite of international standards for software process assessment
- ISO 12207, Software Life Cycle Processes
- ISO 15288, System Life Cycle Processes

Prior to joining the SEI, Mark was a Senior Systems Analyst for System Development Corporation (later Unisys Defense Systems) at the Ballistic Missile Defense Advanced Research Center in Huntsville, Alabama.

Mark received his master's degree in computer science from Vanderbilt University. He received his bachelor's degree in mathematics and computer science from the University of Alabama in Huntsville.

Professional society memberships and certifications

- Senior Member of the Institute of Electrical and Electronics Engineers (IEEE)
- Senior Member of the American Society for Quality (ASQ)
- ASQ Certified Software Quality Engineer

Using the Software CMM® in Small Organizations

Mark C. Paulk

1. Introduction

The Software Engineering Institute (SEI) is a federally funded research and development center established in 1984 by the U.S. Department of Defense with a broad charter to address the transition of software engineering technology – the actual adoption of improved software engineering practices. The SEI's existence is, in a sense, the result of the "software crisis" – software projects that are chronically late, over budget, with less functionality than desired, and of dubious quality. [Gibbs94] To be blunt, much of the crisis is self-inflicted, as when a Chief Information Officer says, "I'd rather have it wrong than have it late. We can always fix it later." The emphasis in many organizations on achieving cost and schedule goals, frequently at the cost of quality, once again teaches a lesson supposedly learned by American industry over twenty years ago and now enshrined in Total Quality Management (TQM).

To quote DeMarco [DeMarco95], this situation is the not-surprising result of a combination of factors:

- "People complain to us because they know we work harder when they complain."
- "The great majority [report] that their software estimates are dismal… but they weren't on the whole dissatisfied with the estimating process."
- "The right schedule is one that is utterly impossible, just not obviously impossible."

DeMarco goes on to observe that our industry is over-goaded, and the only real (perceived) option is to pay for speed by reducing quality.

The lesson of TQM is that focusing on quality leads to decreases in cycle time, increases in productivity, greater customer satisfaction, and business success. The challenge, of course, is defining what "focusing on quality" really means and then systematically addressing the quality issues. Perhaps the SEI's most successful product is the Capability Maturity Model for Software (CMM), a roadmap for software process improvement that has had a major influence on the software community around the world [Paulk95]. The Software CMM defines a five-level framework for how an organization matures its software process. These levels describe an evolutionary path from ad hoc, chaotic processes to mature, disciplined software processes. The five levels, and the 18 key process areas that describe them in detail, are summarized in Figure 1. The five maturity levels prescribe priorities for successful process improvement, whose validity has been documented in many case studies and surveys [Herbsleb97, Lawlis95, Clark97].

This work is sponsored by the U.S. Department of Defense.

Level	Focus	Key Process Areas
5 Optimizing	*Continual process improvement*	Defect Prevention Technology Change Management Process Change Management
4 Managed	*Product and process quality*	Quantitative Process Management Software Quality Management
3 Defined	*Engineering processes and organizational support*	Organization Process Focus Organization Process Definition Training Program Integrated Software Management Software Product Engineering Intergroup Coordination Peer Reviews
2 Repeatable	*Project management processes*	Requirements Management Software Project Planning Software Project Tracking & Oversight Software Subcontract Management Software Quality Assurance Software Configuration Management
1 Initial	*Competent people and heroics*	

Figure 1. An overview of the Software CMM.

Although the focus of the current release of the Software CMM, Version 1.1, is on large organizations and large projects contracting with the government, the CMM is written in a hierarchical form that runs from "universally true" abstractions for software engineering and project management to detailed guidance and examples. The key process areas in the CMM are satisfied by achieving goals, which are described by key practices, subpractices, and examples. The rating components of the CMM are maturity levels, key process areas, and goals. The other components are informative and provide guidance on how to interpret the model. There are 52 goals and 316 key practices for the 18 key process areas. Although the "requirements" for the CMM can be summarized in the 52 sentences that are the goals, the supporting material comprises nearly 500 pages of information. The practices and examples describe what good engineering and management practices are, but they are not prescriptive on how to implement the processes.

The CMM can be a useful tool to guide process improvement because it has historically been a common-sense application of Total Quality Management (TQM) concepts to software that was developed with broad review by the software community. Its five levels are simplistic, but when intelligently used they provide a lever for moving people such as the DOD program manager who bluntly stated, ""The bottom line is schedule. My promotions and raises are based on meeting schedule first and foremost."

While the Software CMM has been very influential around the world in inspiring and guiding software process improvement, it has also been misused and abused by some and not used effectively by others. The guidance provided by CMM v1.1 tends to be oriented towards large projects and large organizations. Small organizations find this problematic, although the fundamental concepts are, we believe, useful to any size organization in any application domain and for any business context.

Are meeting schedules, budgets, and requirements important to small projects? To small organizations? It is arguable that in some environments, such as the commercial shrinkwrap segment, cost is comparatively trivial when compared to the market share available to the first "good enough" product to ship. If the employees of an organization are satisfied with the status quo, there is little that the CMM can provide that will lead to true change; change occurs only when there is sufficient dissatisfaction with the status quo that managers and staff are willing to do things differently. This is as true for small organizations as large.

The CMM provides good advice on desirable management and engineering practices, with an emphasis on management, communication, and coordination of the human-centric, design-intensive processes that characterize software development and maintenance. It should be considered a guidebook rather than a dictate, however, and the CMM user must apply professional judgment based on knowledge and experience in software engineering and management, plus the application domains and business environment of the

organization. Because the CMM is focused on software, there are important aspects of TQM that are not directly addressed in the model, such as people issues and the broader perspective of systems engineering, which may also be crucial to the business. The CMM is a tool that should be used in the context of a systematic approach to software process improvement, such as the SEI's IDEAL model, illustrated in Figure 2 [McFeeley96].

An opening question for software process improvement discussions should always be: Why is the organization interested in using the Software CMM? If the desire is to improve process, with a direct tie to business objectives and a willingness to invest in improvement, then the CMM is a useful and powerful tool. If the CMM is simply the flavor of the month, then you have a prescription for disaster. If the driver is customer concerns, ideally the concerns will lead to collaborative improvement between customer and supplier. Sometimes the supplier's concern centers on software capability evaluations (SCEs), such as are performed by government acquisition agencies in source selection and contract monitoring. DOD policies on the criteria for performing SCEs would exclude most small organizations and small projects [Barbour96], but there are circumstances under which they may occur.

Many of the abuses of the Software CMM spring out of a fear of what "others" may do. If an organization applies common sense to the guidance in the CMM as guidance rather than requirements, then many of the interpretation problems of the model vanish. There are cases, however, where ignorance of good engineering and management practices is the problem. This is particularly problematic for good technical people who have been promoted into management positions, but who have little management experience or training. This contributes to the problems identified by a DOD task force [DOD87]:

- "Few fields have so large a gap between best current practice and average current practice."
- "The big problem is not technical... today's major problems with military software development are not technical problems, but management problems."

2. Small Organizations and Small Projects

The focus of this paper is on using the Software CMM correctly and effectively for small organizations because I am frequently asked, "Can the Software CMM be used for small projects (or small organizations)?" Yet the definition of "small" is challengingly ambiguous, as illustrated in Table 1. At one time there was an effort to develop a tailored CMM for small projects and organizations, but the conclusion of a 1995 CMM tailoring workshop was that we could not even agree on what "small" really meant! The result was a report on how to tailor the CMM rather than a tailored CMM for small organizations [Ginsberg95]. In a 1998 SEPG conference panel on the CMM and small projects [Hadden98a], small was defined as "3-4 months in duration with 5 or fewer staff." Brodman and Johnson define a small organization as fewer than 50 software developers and a small project as fewer than 20 developers [Johnson98].

Table 1. Defining a "Small" Project

Variant of "Small"	Number of People	Amount of Time
Small	3-5	6 months
Very small	2-3	4 months
Tiny	1-2	2 months
Individual	1	1 week
Ridiculous!	1	1 hour

Note that small to tiny projects are in the range being addressed by Humphrey in his Team Software Process℠ (TSP) work, and the individual effort is in the range of the Personal Software Process℠ (PSP) [Humphrey95]. TSP and PSP illustrate how CMM concepts are being applied to small projects. The "ridiculous" variant represents an interpretational problem. On the two occasions this variant has been discussed, the problem was the definition of "project." In both cases it was a maintenance environment, and the organization's "projects" would have been described as tasks in the CMM; the more accurate interpretation for a CMM "project" was a baseline upgrade or maintenance release… but the terminology clash was confusing.

One of the first challenges for small organizations in using the CMM is that their primary business objective is to survive! Even after deciding the status quo is unsatisfactory and process improvement will help, finding the resources and assigning responsibility for process improvement, and then following through by defining and deploying processes is a difficult business decision. The small organization tends to believe

- we are all competent – people were hired to do the job, and we can't afford training in terms of either time or money
- we all communicate with one another – "osmosis" works because we're so "close"
- we are all heroes – we do whatever needs to be done, the rules don't apply to us (they just get in the way of getting the job done), we live with short cycle times and high stress

Yet small organizations, just like large ones, will have problems with undocumented requirements, the mistakes of inexperienced managers, resource allocation, training, peer reviews, and documenting the product. Despite these challenges, small organizations can be extraordinarily innovative and productive. Although there are massive problems that may require large numbers of people to solve, in general small teams are more productive than large teams – they jell quicker and there are far fewer communication problems. The question remains, however, is process discipline needed for small teams? To answer this CMM mantra, we need to consider what discipline involves – and that leads to the heart of this paper's CMM interpretation discussion.

One last precursor, however. When assessing "small" organizations, it is advisable to use a streamlined assessment process; the formality of a two-week CMM-based appraisal for internal process improvement (CBA IPI) is probably excessive [Strigel95, Paquin98, Williams98]. The emphasis should be on efficiently identifying important problems, even if some are missed due to lack of rigor. I recommend focusing on the institutionalization practices that establish the organization's culture: planning, training, etc.; and explicitly tying process improvement to business needs.

3. Interpreting the CMM

Where does the Software CMM apply? The CMM was written to provide good software engineering and management practices for <u>any</u> project in <u>any</u> environment. The model is described in a hierarchy

Maturity levels	(5)
→ **Key process areas**	(18)
→ **Goals**	(52)
→ *Key practices*	(316)
→ *Subpractices and examples*	(many)

In my experience over the last decade of software process work, environments where interpretation and tailoring of the CMM are needed include:

- very large programs
- virtual projects or organizations
- geographically distributed projects
- rapid prototyping projects
- research and development organizations
- software services organizations
- small projects and organizations

The interpretation guidance for small projects and small organizations is also applicable to large projects and organizations. Intelligence and common sense are required to use the CMM correctly and effectively [Paulk96]. It is simultaneously true that all (software) projects are different and all (software) projects are the same. We are required to balance conflicting realities: similarity versus uniqueness, order versus chaos. Those who succeed build lasting organizations [Collins94] that are truly capable of organizational learning [Senge90]; the rest must derive their success elsewhere.

The "normative" components of the CMM are maturity levels, key process areas, and goals. All practices in the CMM are informative. Since the detailed practices primarily support large, contracting

software organizations, they are not necessarily appropriate, as written, for direct use by small projects and small organizations – but they do provide insight into how to achieve the goals and implement repeatable, defined, measured, and continually improving software processes. Thus we prevent such "processes" as the estimating procedure that was simply "Go ask George."

My most frequent interpretation recommendation is to develop a mapping between CMM terminology and the language used by the organization. In particular, terms dealing with organizational structures, roles and relationships, and formality of processes need to be mapped into their organizational equivalents to prevent misunderstandings such as the "ridiculous one-hour project." Examples of organizational structures include "independent groups" such as quality assurance, testing, and configuration management. Appropriate organizational terminology for roles such as project manager and project software manager should be specified. People may fill multiple roles; for example, one person may be the project manager, project software manager, SCM manager, etc. Explicitly stating this makes interpretation of the CMM much simpler and more consistent.

Once the terminology issues are understood, we can think about what the "invariants" for a disciplined process are and which practices depend on the context. In general we assume that key process areas and goals are always relevant to any environment, with the exception of *Software Subcontract Management*, which may be "not applicable" if there is no subcontracting. In contrast, I can conceive of no circumstances under which *Peer Reviews* can be reasonably tailored out for a Level 3 organization. This is a matter of competent professional judgment, although an alternative practice such as formal methods might replace peer reviews. Professional judgment and trained, experienced assessors are crucial, even for small organizations! [Abbott97]

I have never seen an environment where the following were not needed (though implementations differ):

- documented customer (system) requirements
- communication with customer (and end users)
- agreed-to commitments
- planning
- documented processes
- work breakdown structure

Some practices, however, deal with "large-project implementations." A small project is unlikely to need an SCM group or a Change Control Board… but configuration management and change control are always necessary. An independent SQA group may not be desirable, but objective verification that requirements are satisfied always is. An independent testing group may not be established, but testing is always necessary. We thus see that even for context-sensitive practices, the intent is critical even if the implementation is radically different between small organizations and large. Many of the context-sensitive, large-project implementation issues relate to organizational structure. If one reads the CMM definition of "group," it states that "a group could vary from a single individual assigned part time, to several part-time individuals assigned from different departments, to several individuals dedicated full time," which is intended to cater to a variety of contexts.

In addition to these, specific questions that arise repeatedly, especially for small organizations, relate to:

- management sponsorship
- measurement
- SEPGs
- "as is" processes
- documented processes
- tailoring
- training
- risk management
- planning
- peer reviews

Trite though it may seem, obtaining senior management sponsorship is a crucial component of building organizational capability. As individuals, we can exercise professionalism and discipline within our sphere of control, but if an organization as a whole is to change its performance, then its senior management must actively support the change. Bottom-up improvement, without sponsorship and coordination, leads to islands of excellence rather than predictably improved organizational capability. It should be noted, however, that for small organizations, while the president (or founder) is the primary role model, a respected "champion" frequently has the influence to move the entire organization – including the president.

Management by fact is a paradigm shift for most organizations, which must be based on a measurement foundation. To make data analysis useful, you need to understand what the data means and how to analyze it meaningfully. Begin by collecting a simple set of useful data. You also have to be sensitive to the potential for causing dysfunctional behavior by what you measure [Austin96]. The act of measuring identifies what is important, but some things are difficult to measure. Management needs to ensure that attention is visibly paid to all critical aspects of the project, including those difficult to measure, not just those it is easy to measure and track.

In most organizations, a software engineering process group (SEPG) or some equivalent should be formed to coordinate process definition, improvement, and deployment activities. One of the reasons for dedicating resources to an SEPG is to ensure follow-through on appraisal findings. Many improvement programs have foundered simply because no action resulted from the appraisal. Small organizations may not have full-time SEPG staff, but the responsibility for improvement should be explicitly assigned and monitored.

Begin with the "as is" process, not the "should be" process, to leverage effective practices and co-opt resisters. Mandating top-down that everyone will follow the new "should be" process, particularly if not developed by empowered workers, is a common recipe for failure. The "as is" process evolved because the people doing the work needed to get the job done – even if that meant going around the system. The "should be" process may, or may not, be feasible in the given culture and environment. With an organizational focus on process management and improvement, the "as is" and "should be" processes will converge, resulting in organizational learning.

Document your processes. The reasons for documenting a process (or product) are 1) to communicate – to others now and perhaps to yourself later; 2) to understand – if you can't write it down, you don't really understand; and 3) to encourage consistency – take advantage of repeatability. Documented processes support organizational learning and prevent reinventing the wheel for common problems – they put repeatable processes in place. Documentation is therefore important, but documents need not be lengthy or complex to be useful. Keep the process simple because we live in a rapidly changing world. Processes do not need to be lengthy or complex. The CMM is about doing things, not having things. A 1-2 page process description may suffice, and subprocesses and procedures can be invoked as needed and useful. Use good software design principles, such as locality, information hiding, and abstraction, in defining processes. Another useful rule of thumb is to track work at 2-3 tasks per week at most. Order is not created by complex controls, but by the presence of a few guiding formulae or principles [Wheatley92, page 11].

Processes need to be tailored to the needs of the project [Ginsberg95, Ade96]. Although standard processes provide a foundation, each project will also have unique needs. Unreasonable constraints on tailoring can lead to significant resistance to following the process. As Hoffman expresses it, "Don't require processes that don't make sense." [Hoffman98]

The degree of formality needed for processes is a frequent challenge for both large and small organizations [Comer98]. Should there be separate procedure for each of the 25 key practices at Level 2 that mention "according to a documented procedure?" [Hadden98a, Pitterman98] The answer, as discussed in section 4.5.5 "Documentation and the CMM" of the CMM book [Paulk95], is a resounding NO! Packaging of documentation is an organizational decision.

Documented processes are of little value if they are not effectively deployed. To achieve buy-in for the documented, process implementers must be part of process definition and improvement. Training, via a

wide variety of mechanisms, is critical to consistent and effective software engineering and management. The reason for training is to develop skills. There are many "training mechanisms" other than formal classroom training that can be effective in building skills. One that should be seriously considered is a formal mentoring program. In this case, formality means going beyond assigning a mentor and hoping that experience will rub off. Formality implies training people on how to mentor and monitoring the effectiveness of the mentoring.

Training remains an issue after the initial deployment of a process or technology [Abbott97, Williams98]. As personnel change, the incremental need for training may not be adequately addressed. Mentoring and apprentice programs may suffice to address this issue, but they cannot be assumed to be satisfactory without careful monitoring.

Management training is particularly important because ineffective management can cripple a good team. People who are promoted to management because of their technical skills have to acquire a new set of skills, including interpersonal skills [Mogilensky94, Curtis95, Weinberg94].

Some argue that software project management is really risk management. In one sense, the CMM is about managing risk. We attempt to establish stable requirements so that we can plan and manage effectively, but the business environment changes rapidly, perhaps chaotically. We try to establish an island of order in the sea of software chaos, but both order and chaos have a place. As Wheatley suggests, "To stay viable, open systems maintain a state of non-equilibrium, keeping the system in balance so that it can change and grow." [Wheatley92, page 78] Although we can establish processes that help us manage the risks of a chaotic world, we also need to change and grow.

This implies that you should use an incremental or evolutionary life cycle. If you want to focus on risk management, the spiral model may be the preferred life cycle model. If you want to focus on involving the customer, perhaps rapid prototyping or joint application design would be preferable. Few long-term projects have the luxury of the stable environment necessary for the waterfall life cycle to be the preferred choice – yet it is probably the most common life cycle. Note, however, that for small projects, the waterfall life cycle may be an excellent choice.

The #1 factor in successful process definition and improvement is "planfulness" [Curtis96]. Planning is needed for every major software process, but within the bounds of reasonable judgment, the organization determines what is "major" and how the plan should be packaged. A plan may reside in several different artifacts or be embedded in a larger plan.

Although you can argue over the best kind of peer review, the simple fact is that the benefits of peer reviews far outweigh their costs. The data suggests some form of inspection should be used [Ackerman89], but any form of collegial or disciplined review, such as structured walkthroughs, adds significant value. Recognizing the value of peer reviews does not mean, unfortunately, that we do them systematically. We need to "walk the walk," not just "talk the talk." This is very frustrating for technical people who do not understand the emphasis on management in the CMM, yet poor management leads to abandoning good engineering practices such as peer reviews.

There are other issues that have been identified for small organizations and projects. Paquin [Paquin98] identifies five:

- assessments
- project focus
- documentation
- required functions
- maturity questionnaire

We have not discussed the project focus of Level 2 as being a challenge for small organizations. Software process improvement involves overhead that may be excessive for a small project. Some recommend attacking small project process improvement from an organizational perspective [Comer98, Paquin98], which is certainly a reasonable approach, even it does seem to mix Levels 2 and 3. This is a consideration for any size organization or project [Paulk96]. Although an organization can achieve Level 2 without an organization process focus, the most effective organizational learning strategy will be one that

stresses organizational assets that lessen the overhead of projects. At the same time, it must be recognized that there may be resistance to change at the project level, perhaps based on valid concerns, and addressing resistance needs to be considered part of the organization's learning process.

Required functions are an issue because there may be more CMM functions than there are people. This issue has been discussed as terminology or role mapping. The maturity questionnaire is a concern because it uses CMM terminology, thus it may be confusing to those filling it out. Expressing the questionnaire in the terminology of the organization is thus a desirable precursor to even an informal assessment or survey.

Abbott [Abbott97] identifies six keys to software process improvement in small organizations:

- senior management support
- adequate staffing
- applying project management principles to process improvement
- integration with ISO 9001
- assistance from process improvement consultants
- focus on providing value to projects and to the business

If applying good project management to software projects is the best way to ensure success, then the same should be true for process improvement, which should be treated like any other project. ISO 9001 is more frequently an issue for large organizations than small, so it is interesting that Abbott points this out for his small company.

Brodman and Johnson [Johnson98] identify seven small organization/small project challenges:

- handling requirements
- generating documentation
- managing projects
- allocating resources
- measuring progress
- conducing reviews
- providing training

Brodman and Johnson have developed a tailored version of the CMM for small businesses, organizations, and projects [Johnson96, Johnson97, Brodman94]. Although the majority of the key practices in the CMM were tailored in the LOGOS Tailored CMM, the changes can be characterized as:

- clarification of existing practices
- exaggeration of the obvious
- introduction of alternative practices (particularly as examples)
- alignment of practices with small business/small organization/small project structure and resources

Therefore the changes involved in tailoring the CMM for small organizations should not be considered radical.

4. Abusing the Software CMM

Using the CMM correctly means balancing conflicting objectives. CMM-based appraisals require the use of professional judgment. Although the CMM provides a significant amount of guidance in making these judgments, removing subjectivity implies a deterministic, repetitive process that is not characteristic of engineering design work. The CMM is sometimes referred to as a set of process requirements, but it does not contain any "shall" statements. That is why it is an abuse of the CMM to check off (sub)practices for conformance.

Some are unwilling or unable to interpret, tailor, or apply judgment. It is easy to mandate the key practices, but foolhardy. This foolishness is frequently driven by paranoia about customer intentions and competence. On more than one occasion I have heard someone say they were doing something that was foolish, but they ware afraid that the customer was so ignorant or incompetent that they would be unable to understand the rationale for doing things differently than literally described in the CMM. This is particularly problematic for SCEs. It is true that judgments may differ – and sometimes legitimately so. What is adequate in one environment may not suffice for a new project. That is why we recommend that process maturity be included in risk assessment rather than using maturity levels to filter offerors

[Barbour96]. Small organizations should have less of a concern with this problem since it is unlikely that SCEs for small organizations are cost-effective. It is more of a problem for large organizations with many small projects.

Unfortunately I have no solution for this problem. "Standards" such as the CMM can help organizations improve their software process, but focusing on achieving a maturity level without addressing the underlying process can cause dysfunctional behavior. Maturity levels should be measures of improvement, not goals of improvement. That is why we emphasize the need to tie improvement to business objectives.

5. Conclusion

The bottom line is that software process improvement should be done to help the business – not for its own sake. This is true for both large organizations and small. The best advice comes from Sanjiv Ahuja, President of Bellcore: "Let common sense prevail!"

Building software is a design-intensive, creative activity. While the discipline of process is a crucial enabler of success, the objective is to solve a problem, and this requires creativity. Software processes should be repeatable, even if they are not repetitive. The balance between discipline and creativity can be challenging [Glass95]. Losing sight of the creative, design-intense nature of software work leads to stifling rigidity. Losing sight of the need for discipline leads to chaos.

The CMM represents a "common sense engineering" approach to software process improvement. Its maturity levels, key process areas, goals, and key practices have been extensively discussed and reviewed within the software community. While the CMM is neither perfect nor comprehensive, it does represent a broad consensus of the software community and is a useful tool for guiding improvement efforts, and it can be uses to help small software organizations improve their processes [Abbott97, Hadden98b, Hoffman98, Pitterman98, Sanders98].

Small organizations should seriously consider PSP and TSP [Ferguson97, Hayes97]. Having taken the PSP course, I can highly recommend it for building self-discipline. Note that the effect of reading the book is not the same as taking the course and doing the work! Where the CMM addresses the organizational side of process improvement, PSP addresses building the capability of individual practitioners. The PSP course convinces the individual, based on his or her own data, of the value of a disciplined, engineering approach to building software.

References

Abbott97 John J. Abbott, "Software Process Improvement in a Small Commercial Software Company," , **Proceedings of the 1997 Software Engineering Process Group Conference**, San Jose, CA, 17-20 March 1997.

Ackerman89 A.F. Ackerman, L.S. Buchwald, and F.H. Lewski, "Software Inspections: An Effective Verification Process," IEEE Software, Vol. 6, No. 3, May 1989, pp. 31-36.

Ade96 Randy W. Ade and Joyce P. Bailey, "CMM Lite: SEPG Tailoring Guidance for Applying the Capability Maturity Model for Software to Small Projects," **Proceedings of the 1996 Software Engineering Process Group Conference: Wednesday Papers**, Atlantic City, NJ, 20-23 May 1996.

Austin96 Robert D. Austin, **Measuring and Managing Performance in Organizations**, Dorset House Publishing, ISBN: 0-932633-36-6, New York, NY, 1996.

Barbour96 Rick Barbour, "Software Capability Evaluation Version 3.0 Implementation Guide for Supplier Selection," Software Engineering Institute, Carnegie Mellon University, CMU/SEI-95-TR-012, April 1996.

Brodman94 J.G. Brodman and D.L. Johnson, "What Small Businesses and Small Organizations Say About the CMM," **Proceedings of the 16th International Conference on Software Engineering**, IEEE Computer Society Press, Sorrento, Italy, 16-21 May 1994, pp. 331-340.

Clark97 Bradford K. Clark, "The Effects of Software Process Maturity on Software Development Effort," PhD Dissertation, Computer Science Department, University of Southern California, August 1997.

Collins94 James C. Collins and Jerry I. Porras, **Built to Last**, HarperCollins Publishers, New York, NY, 1994.

Curtis95 Bill Curtis, William E. Hefley, and Sally Miller, "People Capability Maturity Model," Software Engineering Institute, CMU/SEI-95-MM-02, September 1995.

Curtis96 Bill Curtis, "The Factor Structure of the CMM and Other Latent Issues," **Proceedings of the 1996 Software Engineering Process Group Conference: Tuesday Presentations**, Atlantic City, NJ, 20-23 May 1996.

DeMarco95 Tom DeMarco, **Why Does Software Cost So Much?**, ISBN 0-932633-34-X, Dorset House, New York, NY, 1995.

DOD87 Department of Defense, "Report of the Defense Science Board Task Force on Military Software," Office of the Under Secretary of Defense for Acquisition, Washington, D.C., September 1987.

Ferguson97 Pat Ferguson and Jeanie Kitson, "CMM-Based Process Improvement Supplemented by the Personal Software Process in a Small Company Environment," , **Proceedings of the 1997 Software Engineering Process Group Conference**, San Jose, CA, 17-20 March 1997.

Gibbs94 W. Wayt Gibbs, "Software's Chronic Chrisis," Scientific American, September 1994, pp. 86-95.

Ginsberg95 Mark Ginsberg and Lauren Quinn, "Process Tailoring and the Software Capability Maturity Model," Software Engineering Institute, CMU/SEI-94-TR-024, November 1995.

Glass95 Robert L. Glass, **Software Creativity**, Prentice Hall, Englewood Cliffs, NJ, 1995.

Hadden98a Rita Hadden, "How Scalable are CMM Key Practices?" Crosstalk: The Journal of Defense Software Engineering, Vol. 11, No. 4, April 1998, pp. 18-20, 23.

Hadden98b Rita Hadden, "Key Practices to the CMM: Inappropriate for Small Projects?" panel, Rita Hadden moderator, **Proceedings of the 1998 Software Engineering Process Group Conference**, Chicago, IL, 9-12 March 1998.

Hayes97 Will Hayes and James W. Over, "The Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers," Software Engineering Institute, Carnegie Mellon University, CMU/SEI-97-TR-001, December 1997.

Herbsleb97 James Herbsleb, David Zubrow, Dennis Goldenson, Will Hayes, and Mark Paulk, "Software Quality and the Capability Maturity Model," Communications of the ACM, Vol. 40, No. 6, June 1997, pp. 30-40.

Hoffman98 Leo Hoffman, "Small Projects and the CMM," in "Key Practices to the CMM: Inappropriate for Small Projects?" panel, Rita Hadden moderator, **Proceedings of the 1998 Software Engineering Process Group Conference**, Chicago, IL, 9-12 March 1998.

Humphrey95 Watts S. Humphrey, **A Discipline for Software Engineering**, ISBN 0-201-54610-8, Addison-Wesley Publishing Company, Reading, MA, 1995.

[Johnson96] Donna L. Johnson and Judith G. Brodman, **The LOGOS Tailored Version of the CMM for Small Businesses, Small Organizations, and Small Projects**, Version 1.0, August 1996.

Johnson97 Donna L. Johnson and Judith G. Brodman, "Tailoring the CMM for Small Businesses, Small Organizations, and Small Projects," Software Process Newsletter, IEEE Computer Society Technical Council on Software Engineering, No. 8, Winter 1997, p. 1-6.

Johnson98 Donna L. Johnson and Judith G. Brodman, "Applying the CMM to Small Organizations and Small Projects," **Proceedings of the 1998 Software Engineering Process Group Conference**, Chicago, IL, 9-12 March 1998.

Lawlis95 Patricia K. Lawlis, Robert M. Flowe, and James B. Thordahl, "A Correlational Study of the CMM and Software Development Performance," Crosstalk: The Journal of Defense Software Engineering, Vol. 8, No. 9, September 1995, pp. 21-25. Reprinted in Software Process Newsletter, IEEE Computer Society Technical Council on Software Engineering, No. 7, Fall 1996, pp. 1-5.

McFeeley96 Bob McFeeley, "IDEAL: A User's Guide for Software Process Improvement," Software Engineering Institute, CMU/SEI-96-HB-001, February 1996.

Mogilensky94 Judah Mogilensky and Betty L. Deimel, "Where Do People Fit in the CMM?," American Programmer, Vol. 7, No. 9, September 1994, pp. 36-43.

Paquin98 Sherry Paquin, "Struggling with the CMM: Real Life and Small Projects," in "Key Practices to the CMM: Inappropriate for Small Projects?" panel, Rita Hadden moderator, **Proceedings of the 1998 Software Engineering Process Group Conference**, Chicago, IL, 9-12 March 1998.

Paulk95 Carnegie Mellon University, Software Engineering Institute (Principal Contributors and Editors: Mark C. Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis), **The Capability Maturity Model: Guidelines for Improving the Software Process**, ISBN 0-201-54664-7, Addison-Wesley Publishing Company, Reading, MA, 1995.

Paulk96 Mark C. Paulk, "Effective CMM-Based Process Improvement," **Proceedings of the 6th International Conference on Software Quality**, Ottawa, Canada, 28-31 October 1996, pp. 226-237.

Pitterman98 Bill Pitterman, "Key Practices to the CMM: Inappropriate for Small Projects?" panel, Rita Hadden moderator, **Proceedings of the 1998 Software Engineering Process Group Conference**, Chicago, IL, 9-12 March 1998.

Sanders98 Marty Sanders, "Small Company Action Training and Enabling," in **The CMM and Small Projects**, Society for Software Quality Roundtable, Washington, DC, 26 January 1998.

Senge90 Peter M. Senge, **The Fifth Discipline: The Art & Practice of the Learning Organization**, Doubleday/Currency, New York, NY, 1990.

Strigel95 Wolfgang B. Strigel, "Assessment in Small Software Companies," **Proceedings of the 1995 Pacific Northwest Software Quality Conference**, 1995, pp. 45-56.

Weinberg94 Gerald M. Weinberg, **Quality Software Management, Volume 3: Congruent Action**, ISBN 0-932633-28-5, Dorset House, New York, NY, 1994.

Wheatley92 Margaret J. Wheatley, **Leadership and the New Science**, Berrett-Koehler Publishers, San Francisco, CA, 1992.

Williams98 Louise B. Williams, “SPI Best Practices for ‘Small’ Projects,” in **The CMM and Small Projects**, Society for Software Quality Roundtable, Washington, DC, 26 January 1998.

Process Synergy: Using ISO 9000 and the CMM in a Small Development Environment

Abstract

ISO 9001 certification does not guarantee SEI Level 2 or 3 conformance and Level 2 or 3 conformance does not guarantee ISO 9001 certification. However, the synergy between ISO 9001 and the Capability Maturity Model[SM] can be used to drive software development organizations toward ISO certification, when that is the primary goal, or toward higher maturity levels if that is the goal.

Small development organizations find compliance with the requirements of ISO 9001 and the Capability Maturity Model especially daunting. The overhead of process documentation, process conformance and record keeping can strain the resources of a small team. This case study explores how a small software development organization was successfully integrated into the overall corporate ISO 9001 registration, and moved higher up the CMM scale in the process.

The lessons learned from this journey from "classic Level 1" to ISO 9001 conformant and close to Level 2 apply to any ISO or process improvement effort whether it is inspired by an ISO requirement or a self motivated commitment to improve.

Author

Sharon E. Miller has over 30 years of diversified business and consulting experience with an emphasis in the areas of software process analysis, assessment and auditing. She is a co-architect of a widely-accepted software assessment program that compares an organization's performance against industry standards and produces a gap analysis and blueprint for improvement. An ASQ Certified Quality Auditor, a RAB-certified ISO 9000 and TickIT lead auditor, she has guided organizations across diverse industries to achieve ISO certification. An accomplished international speaker, Sharon has authored several publications outlining the role of software quality as an integral part of effective business operations. She holds a Master's degree in Advanced Management from Pace University and is currently Treasurer of the ASQ Software Division. Sharon is retired from Lucent Technologies' Bell Laboratories and is a founding partner in Northstar Consulting Group.

Wyckoff Building • 33 North Main Street • Marlboro, NJ 07738 • Tel.: (732) 933-9834
www.northstarcg.com

Process Synergy: Using ISO 9000 and the CMM in a Small Development Environment

Abstract

ISO 9001 certification does not guarantee SEI Level 2 or 3 conformance and Level 2 or 3 conformance does not guarantee ISO 9001 certification. However, the synergy between ISO 9001 and the Capability Maturity Model[SM] can be used to drive software development organizations toward ISO certification, when that is the primary goal, or toward higher maturity levels (Paulk 94) if that is the goal.

Small development organizations find compliance with the requirements of ISO 9001 and the Capability Maturity Model especially daunting. The overhead of process documentation, process conformance and record keeping can strain the resources of a small team. This case study explores how a small software development organization was successfully integrated into the overall corporate ISO 9001 registration, and moved higher up the CMM scale in the process.

The lessons learned from this journey from "classic Level 1" to ISO 9001 conformant and close to Level 2 apply to any ISO or process improvement effort whether it is inspired by an ISO requirement or a self motivated commitment to improve.

Keywords: software engineering, software product development, ISO 9001, process management.

Background

The company described in this case study is a large international computer manufacturer that has been ISO registered for a number of years. Their corporate level software development processes had passed ISO scrutiny, but there was a small (~ 20) software development organization, relatively uninvolved in the mainstream product lines, that had not yet been brought under the ISO registration. The external registrars had made it clear at the last surveillance audit that they expected to see progress toward bringing this organization under the ISO registration at the next audit, which at the start of the project was only 6 weeks away, and that they intended to begin auditing them on a regular basis 6 months later.

Because the corporate ISO umbrella encompassed all 20 ISO elements, the software development organization only had to deal with new process definitions for their design control activities. They did, however, have to ensure that they understood, and were conforming to, corporate ISO processes that affected them, such as document control, training records, job descriptions, etc.

While this small organization had some de facto processes in place, there were no documented process descriptions per se and project documentation was not well controlled. The testing function was in the process of being reorganized, so its relationship to the developers was flexible. Because the software releases were linked to hardware releases, there was an overall, well-defined project management process for the hardware/software system, but the software team was not tightly coupled to that process.

The plan to move into compliance with ISO 9001 did not originally have any process improvement intent. It was simply to put ISO conformant processes in place to pass the audit. Process improvements were a natural result of management involvement, team involvement, and lots of communication. The lessons learned in the course of this organizations' journey apply equally well to any ISO or process improvement effort whether it is an offshoot of an ISO requirement or a self motivated commitment to improve.

Lesson 1: It doesn't matter why you begin process improvements, as long as you tackle them with the right mindset.

The software development team was faced with an audit in 6 weeks. The auditors had made it clear they expected significant progress toward ISO 9001 compliance and management did not want to risk losing corporate, world-wide registration because of a small hardware/software organization operating outside the mainstream

product development processes. It was a bad time to be starting this effort, but they had no choice. They also could not jeopardize their next release date.

Faced with this challenge, the software manager approached the task with the mindset of defining processes that would best meet the needs of the business, improving the efficiency of their development efforts, and identifying how ISO could be used as a forcing function to help improve the working relationship with their hardware partners. With this mindset, the tone for all the activity that followed was set.

There was lots of ongoing communication with the entire development team. Process definition and development efforts involved the entire team. The end result was simple, effective software development processes that the entire team was willing to embrace, as well as a better interface with the hardware team.

Lesson 2: Don't assume you can adopt someone else's processes.

To be ready for the first audit in 6 weeks, the initial approach was "We need software development processes that will pass ISO scrutiny. Corporate has these processes in place, so we'll just adopt their processes and be finished."

Several problems quickly emerged. First, the rest of the corporation was doing large scale UNIX®-based systems development. This small organization was developing PC-based applications, so many of the corporate processes and tools were not applicable to their environment.

Second, the level of detail in the corporate processes was such that it would require significant additional resources to manage the process overhead, a situation this small team could not afford. There was a lot of documentation called for by the corporate processes to manage the interfaces among groups in very large software development efforts.

And finally, the corporate processes did not describe the way the small organization worked as a team nor how they interfaced with their hardware development partner.

The need to for the small team to develop their own set of software development processes was clear.

Lesson 3: Figure out how you really work before you write your processes.

The attempt to adopt the corporate processes, even an attempt to simplify those documents and use them, would have resulted in trying to use documented processes that did not match the actual way people worked and interfaced with each other. The organization was a classic CMM Level 1, a highly skilled team developing robust code without documented processes, but they were following *de facto* processes and there was a formal documented project management process for the hardware/software releases.

Using interviewing and process modeling techniques, a picture of how the organization actually worked emerged, showing that there was a fair amount of formal process being followed by this team. These findings became the basis for the team's documented process descriptions, job descriptions, and skill requirements.

Lesson 4: Don't bite off more than you can chew.

A common mistake organizations make is to over-document processes, both with too many processes and too much detail in each process description. When this is the case audits generally find people are not following the documented processes and that there are no quality records for many of the activities. The manager was committed to not having this happen. She wanted to keep things simple, as this small organization had only 6 weeks to show progress for the auditors, and she wanted to keep things realistic so audits would not be stressful. In addition they needed their processes to reflect the business climate of frequent releases to a highly volatile market.

For the purpose of the initial audit, the team identified and documented the 4 processes that captured the basics of their software development environment and defined a plan for full compliance by the next audit (roughly 6 months away). These initial processes documented were:

- Software Project Management,
- Design,

- Implementation, and
- Software Test.

With these 4 processes in place and a plan for full compliance to show the auditors, the team satisfied the auditors that they had made significant progress, and were successful in being brought under the corporate registration.

In the second phase of their process development, the team added 3 additional processes based on functions the team already performed and activities that would improve their software development environment. These processes were:

- Software Beta Test,
- Modification Request, and
- Configuration Management.

The checklist in Attachment 1 shows which activities of the Capability Maturity Model (Paulk 93) were incorporated into their processes. This small organization would not be assessed at Level 2, but their processes were a good compromise between satisfying much of the spirit of the CMM and keeping the processes manageable.

Lesson 5: Document what you do, not what you think you should be doing, for the most part.

Another mistake often made in process improvement or ISO efforts is to define the process as a "textbook" description of the ideal software process. This is a recipe for disaster. The process will not match how the people work. The people will not feel any connection to the processes, and will not follow them. The small team was careful to document what they were actually doing, for the most part. The team's manager was determined not to put anything in a process document that would not be followed in practice, but the team also realized that this was an excellent opportunity make some necessary process improvements.

For example, when the team was asked to describe how they did configuration management, they realized that the process they were describing was not serving their needs very well. So, in this case, rather than documenting the existing process, the team redesigned the process, with the buy-in and commitment of all key stakeholders to the new process. The new process was a significant improvement over they way they had been managing configuration control.

Lesson 6: Controlling documents does not have to be hard.

While the team generally had adequate project documentation, it was not consistently controlled or maintained across projects. Not all project documents were accessible to all project members. Some were on shared drives in shared directories; some were in personal directories. Since everyone had access to a shared drive, the team set up a consistent directory structure by project and a naming convention for all project documents. The manager's involvement was instrumental in getting this accomplished, making sure that the directory structure was established, that everyone was aware of it, and that all documents were moved to the appropriate place. While this delighted the auditors, the team also benefited from knowing exactly where the right version of all project documents could be found.

Process documentation was put in appropriate directories under the corporate ISO document control structure on the corporate intranet.

Lesson 7: Train and communicate, effectively and often.

Process implementation involves change, and change requires training and communication. Lots of it. Everyone in the organization must be aware of the requirements of their processes, especially if they are new or changed. Everyone needs to be aware of how the changes affect them personally; how their skills match the new requirements of their processes. This can only be accomplished through training and communication. This team

had several training sessions on the new processes, and electronic mail was used very effectively to keep the whole team informed of progress throughout the process.

The training also included sessions on the corporate ISO program to make the team aware of the ISO requirements and the specifics of the corporate quality system. A key aspect of both training sessions was making sure the team knew where all the relevant documents, quality records and procedures were stored on the shared drive and the corporate intranet.

The training also included some of the "do's and don't's" on being audited. The training coupled with 2 internal audits prior to the surveillance audits made people less apprehensive about the prospect of being audited.

Lesson 8: Utilize checklists.

In addition to training, a checklist was developed for each process so the team members could make sure they were doing everything required by the process. These checklists are shown in Attachment 2. Checklists serve two very useful purposes, especially when preparing for an audit. First, checklists give team members a quick visual check of anything that might have been missed, either a document or a process step. Second, checklists impress the auditors, making their job easier, which usually makes the audit go more smoothly.

Lesson 9: It's OK to get outside help.

Because of the size of this team and their schedule commitments, there were no resources on the team that could be spared to manage the project. In addition, no one on the team was familiar with the ISO standard and how its requirements are applied to a software development organization. The only way they could get ISO conformant processes defined and documented in time was to bring in an outside expert, who understood both ISO 9001 and software processes. It was successful because both the manager and the consultant approached the problem with the same mindset; keep it simple, keep it realistic, and make sure that, in the end, the processes belong to the team, and not to the consultant.

Using a consultant kept the time and effort required by team members to a minimum. The consultant facilitated process definition sessions with key stakeholders for each process, documented the findings of those sessions in process documents, and held reviews before finalizing each process document. The consultant provided ISO and process training to the team. Most team members spent less than 3 days in the preparation effort over a 6 month period, 3 half days of training, 1/2 day in a facilitated meeting to define their process, and less than 1/2 day reviewing their process document. Three key team leaders were more heavily involved in the process documentation and review and in the establishment of the document control structure for the team. These team members spent between 6 and 12 days each over the 6 month period. The consultant was helping 4 small organizations prepare simultaneously, and was not dedicated to the software team full time.

The consultant acted as the liaison to the corporate quality group, ensuring that all the corporate requirements were satisfied, particularly document control, training records and job descriptions. The consultant also acted as a liaison between the software and hardware group, ensuring that the process descriptions were consistent across both groups, and that the roles, responsibilities, and handoffs were clearly defined.

The team was able to implement the new processes, successfully survive 2 internal and 2 external audits, and meet their original release schedule.

Lesson 10: It takes both "top-down" and "bottom-up" support to make it work.

Management commitment is always cited as critical to the success of any ISO registration program, and it was crucial in this case. The team was very process averse to start, but they soon realized that their manager was committed to their being brought under the corporate ISO umbrella and that documenting and following their processes was not going to be optional. However, they were still skeptical that anything good would result from the efforts. The manager was diligent in delivering the "top-down" message that this was something that had to happen, but that it was going to happen in a way that improved their development environment.

The manager was able to engage the full support of a few key team leaders who had the respect of the rest of the team, and these team leaders were instrumental in convincing their peers that the resulting process was going to be an improvement.

When the team saw the first drafts of the processes and realized that they were short and simple and described the way the team wanted to work (especially the clarification of the interface to their hardware partner), the entire team became supportive and engaged.

The team did very well on their both surveillance audits, having only a couple of minor nonconformities in each.

Summary

While this case study describes a successful ISO effort, it also describes process improvement resulting for the ISO effort. The success in successfully being brought under the corporate ISO umbrella and seeing some process improvement as well was due in large part to the commitment of the manager to satisfy the ISO requirements with software processes that met the needs of the business, but to be willing to let the team implement some process changes, such as configuration control, that led to significant improvement. Without her continued emphasis on simplicity and reality and without the engagement of key team leaders, it could easily have been yet another process exercise that produced a binder that gathers dust on everyone's shelves.

References:

Paulk, Mark C., A Comparison of ISO 9001 and the Capability Maturity Model for Software, (CMU/SEI-94-TR-12). Pittsburgh, PA: Software Engineering Institute, July 1994.

Paulk, Mark C., et. al. Key Practices of the Capability Maturity Model, Version 1.1, (CMU/SEI-93-TR-25, ADA 263432). Pittsburgh, PA: Software Engineering Institute, February 1993.

Attachment 1:

CMM Practices Mapped to ISO Requirements and Software Processes

CMM Practices Adopted	ISO Requirement Satisfied	Software Process Addressing the Requirement
Requirements Management		
☐ Requirements are documented and reviewed	– Design input, design review	– Design
☐ Project plans and activities are kept consistent with requirements if they change	– Design and development planning – Design changes	– Project Management
Software Project Planning		
☐ A software project plan is documented	– Design and development planning	– Project Management
☐ All stakeholders are aware of and agree to their commitments	– Organizational and technical interfaces	– Project Management
Software Project Tracking and Oversight		
☐ Actual results are tracked against the estimates	– Design control	– Project Management
☐ Corrective action is taken when actuals deviate significantly from estimates	– Design control	– Project Management
☐ Changes are agreed to by all stakeholders	– Design changes	– Project Management
Software Configuration Management		
☐ Selected work products are identified, controlled and available	– Product identification and traceability, document control	– Configuration Management
☐ Changes to those work products are controlled	– Product identification and traceability, document control	– Configuration Management
☐ Stakeholders are aware of the status and content of software baselines	– Product identification and traceability, document control	– Configuration Management

CMM Practices Mapped to ISO Requirements and Software Processes (Cont'd.)

Training Program		
☐ Training activities are planned	– Training	– N/A[1]
☐ Training is provided for management and technical personnel	– Training	– N/A[1]
☐ Individuals receive needed training	– Training records	– N/A[1]
Peer Reviews		
☐ Peer review activities are planned	– Design reviews	– Implementation
☐ Defects in work products are identified and removed	– Design verification	– Implementation
Software Product Engineering		
☐ Software deliverables are produced according to the defined software process	– Design control	– Project Management – Design – Implementation – Test – Beta Test – Modification Request – Configuration Management
☐ Consistency is maintained across software work products (i.e., the documentation tracing requirements through design, code, and test cases is maintained)	– Product identification and traceability, design	– Configuration Management – Design – Test
☐ The project follows a written policy which requires the use of appropriate methods and tools for building and maintaining software products	– Design control	– Project Management – Implementation – Test
☐ Software code is developed according to the project's defined software process	– Design control	– Implementation
☐ Software testing is performed according to the project's defined software process	– Design validation	– Test – Beta Test

[1] Training processes are covered in a corporate level process.

Attachment 2:

Software Process Execution Checklists

Project Management Checklist	Yes	No
Project plan complete		
Development environment on shared drive established		
Schedules complete		
Requirements complete		
Requirements reviewed		
Document placed on shared drive		
Quality record of review placed on shared drive		
Design complete		
Design reviewed		
Document placed on shared drive		
Quality record of review placed on shared drive		
Code inspected		
Quality record of review placed on shared drive		
Status meetings held		
Schedules revised at the completion of each phase		
User documentation available before the end of unit test		
Design Checklist	**Yes**	**No**
Feature requirements available		
Design Specification stored on shared drive		
Design review held		
Design review minutes stored on shared drive		
Design changes reviewed		
Implementation Checklist	**Yes**	**No**
Code inspections planned and documented in project plan and schedule		
Developer debug completed		
Meeting notices sent 2 days in advance of meeting		
Inspection packet sent 2 days in advance of meeting		
Records of inspections on shared drive		
Official build for unit test		
Source submitted for unit test		
Modification Request (MR) Checklist	**Yes**	**No**
MR system of choice specified in project plan		
MRs created		
MRs closed		

Software Process Execution Checklists (Cont'd.)

Software Test Checklist	Yes	No
Code under source control		
Code compiled and linked cleanly		
Code inspection process followed for new code developed		
Source control generated binaries available		
Unit test plan documented and under change control		
Unit test plan reviewed		
1st official build delivered		
Unit test 100% complete		
Unit test report completed		
Feature integration test (FIT) plan document under change control		
Feature integration test plan reviewed		
Customer ready model hardware available		
Documentation available (transmittal, install guide, kit instructions, etc.)		
Master install media available (generated via the source control system)		
List of all open hardware, firmware, and software problems		
Unit test metrics satisfied		
Feature integration test 100% complete		
Regression testing done, if needed		
Configuration Management Checklist	**Yes**	**No**
CM tool specified in project plan		
Project members familiar with source control tool		
Project created in source control tool by feature team leader (FTL)		
1st official build given to FIT leader by FTL at unit test plan review		
Official build machine designated and controlled by FIT leader		
Unique build numbers for each version of software		
Diskettes with official build numbers delivered to FIT leader		
Source code backups done nightly		
Storage of source code		
Final official build created by FTL and delivered to feature audit		
Beta Test Checklist	**Yes**	**No**
Beta test plan documented in the project plan		
Beta customers account profiles prepared		
Beta customers selected		
Pre-release agreements signed by Beta customers		
Legal provided with copies of media labels, screens, customer information		
Beta media prepared by FTL		
Beta media appropriately labeled		
Beta media shipped by Beta team leader		
Log of customer contacts for Beta		
MRs of Beta found problems		
Customer records provided to Beta test leader		
Beta test report completed		
Beta software replaced with released software		

Quality Assurance Activities in the Software Development Center, Hitachi Ltd.

This document describes the QA department has contributed to enhance productivity of software development by evaluating quality precisely in various methods.

Hitoshi Kihara, Senior Engineer, Quality Assurance Department, Software Development Center, Hitachi, Ltd.,

5030 Totsuka-cho, Totsuka-ku, Yokohama,244-8555,Japan

Phone:+81-45-881-7161 Fax:+81-45-865-9069

Quality Assurance Activities in the Software Development Center, Hitachi Ltd.

Hitoshi Kihara, Software Development Center, Hitachi, LTD. Japan

Yutaka Fukunaga, Software Development Center, Hitachi, LTD. Japan

Abstract

The Quality Assurance (QA) department in our Software Development Center (SDC) is responsible for quality assurance activities and is independent of product development departments. The QA department reviews software designs, and inspects and maintains software products. In SDC, the QA department focuses on researching quality assurance schemes for detecting the causes of errors, and on preventing similar errors. Through these studies, the QA department has combined various quality improvement measures and thus enhanced productivity in the development of high-quality software. From 1997, SDC has been able to develop mainframe software products in about half the time it took in 1991. SDC also applies these quality improvement measures to client/server software products, which consequently achieved as high level of performance and reliability as the level of mainframe software products.

This document describes possible problems in software development. This document also describes solutions and resulting effects.

1. Quality management overview and problems

1.1 Quality management overview

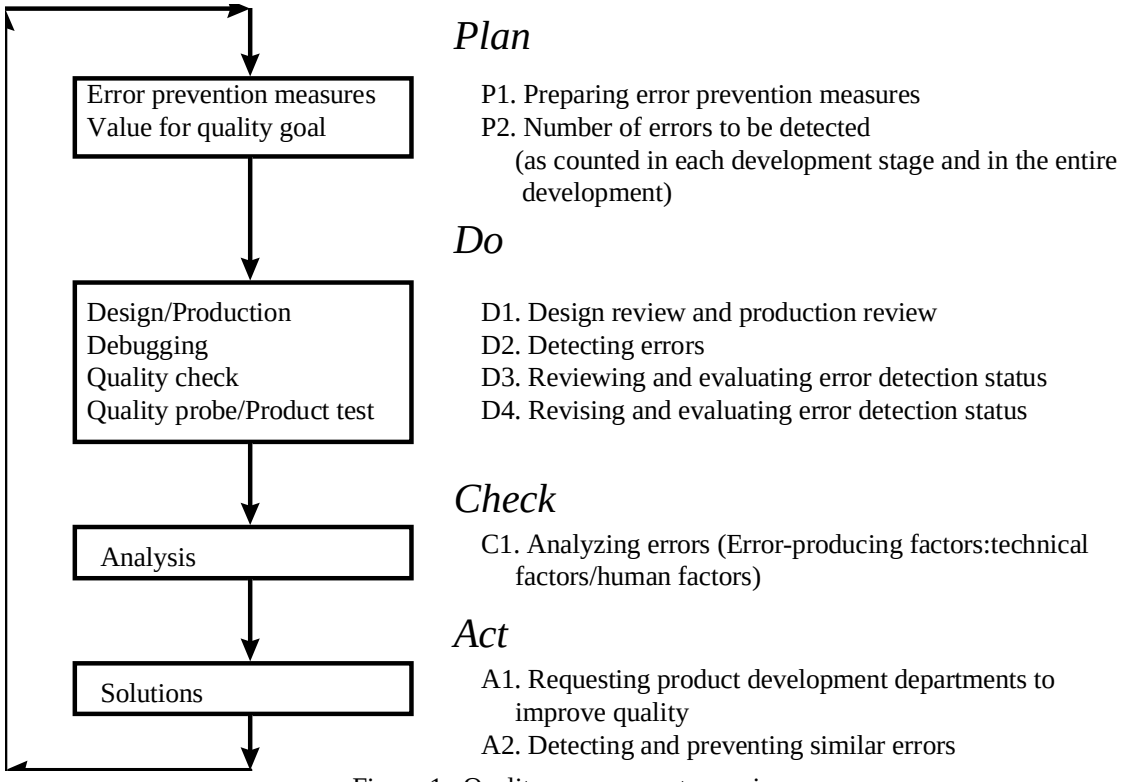


Figure 1: Quality management overview

Quality management must be applied to all development stages including software design, debugging, inspection and on-site operation. In every development stage, we must:

- prepare error prevention measures and set up quality goals. (PLAN)
- review products and detect errors. (DO)
- analyze errors. (CHECK)

- apply quality improvement measures. (ACT)

Figure 1 shows tasks in the PDCA (Plan-Do-Check-Act) cycle. The duty of QA is to check whether high quality products are created in the cycle of the PDCA tasks, and also to direct countermeasures when some problems occur.

1.2 Problems

If all the tasks in the quality control stages in Figure 1 were performed perfectly, we would never have to worry about quality issues. However, the level of performance depends heavily on the persons performing the tasks, and the level greatly affects the quality of the software product. The level of performance is related to the following problems:

Problem 1: 90% of bugs remain in the downstream stages of the programming process

We need a lot of time and money to improve a low-quality program which needs functions added, performance improved, and bugs fixed.

Figure 2 illustrates the correlation between bugs produced and bugs detected.

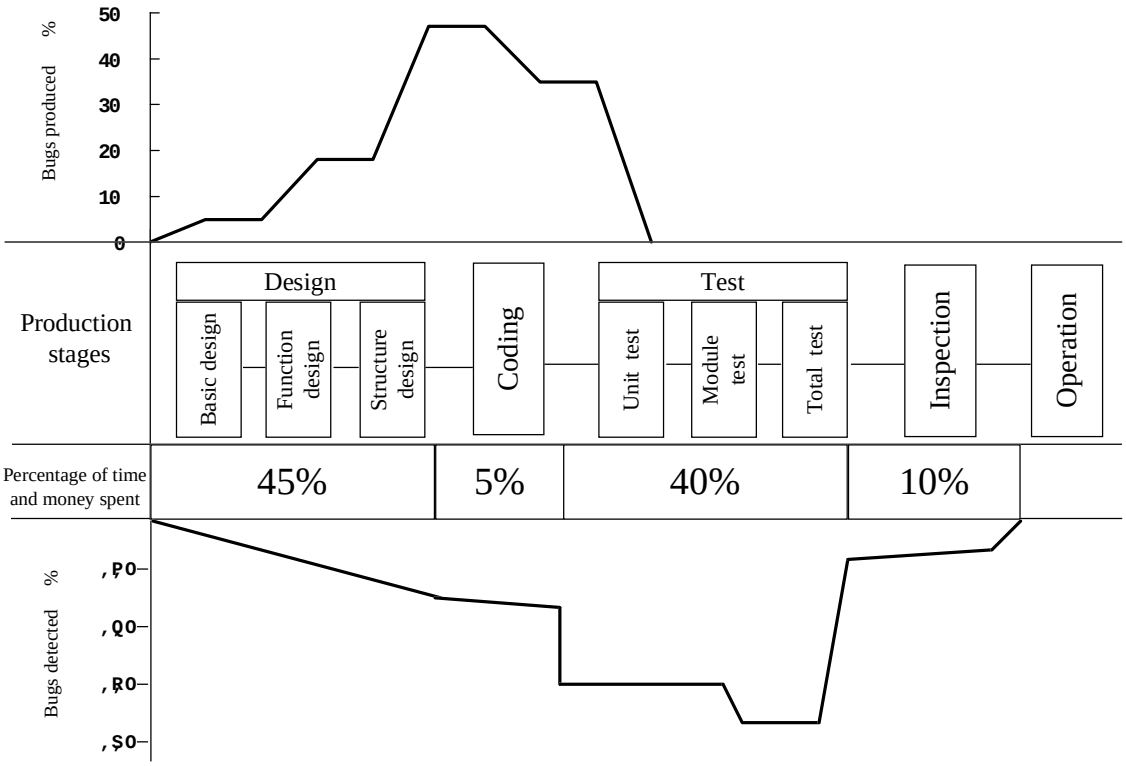


Figure 2: Correlation between bugs produced and bugs detected

In the testing stages and later stages, we must spend a lot of time (even up to 50% of all the work-hours) to detect more than 90% of the errors produced in the previous designing and coding stages. Moreover, the later the production stage, the greater the computer-related costs are, as well as the greater the human resources required. Therefore, we have to reduce bugs produced in the upstream stages and try to detect bugs as early as possible.

Problem 2: Errors damage the customers satisfaction

Customers are very unsatisfied if errors are detected immediately after installation. Customers are also very unsatisfied if errors are detected in basic functions. Most of all, customers expect high quality from on-line programs and operating systems, which are developed with the assumption that reliability is a key goal. Errors that occur immediately after installation of these programs and errors in basic functions of these programs significantly damage the vendor's reputation for reliability and damage good

relationships between the vendor and the customer. These errors also severely damage sales promotion activities by the vendor.

Problem 3: Multiple defects are difficult to detect

Bugs in error recovery routines are difficult to detect because these routines only function when a hardware failure occurs. Such bugs can cause multiple defects and thus can significantly reduce customer satisfaction. Customers, however, regard error recovery routines as an *insurance* and spend a lot of money on such routines.

In a duplex file system as illustrated in Figure 3, a hardware failure in one path might cause a system failure if the recovery routine blockades failure path and retries the i/o requestin the operating system contains bugs. The customer would strongly complain about the malfunction of a dual path system for which the customer made a large investment for insurance.

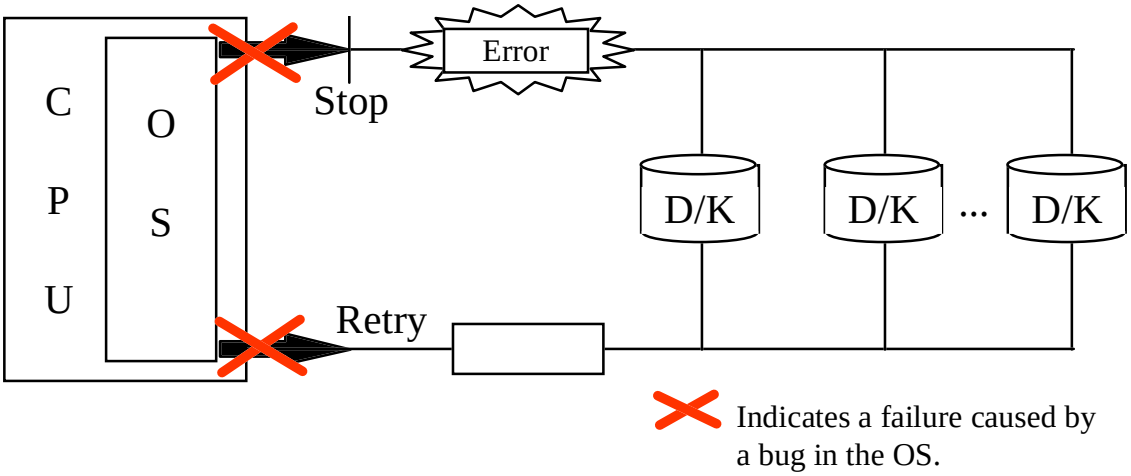


Figure 3: Example of multiple failures in a dual path system.

2. Quality management methods

2.1 Solution to problem 1

Enhance quality in the upstream stages of the programming process (To improve review system)

The workers from QA (Quality Assurance) and SD (Software Designers) meet programmers to review the functional design and discuss ways to achieve functions. They try to identify the spots that may be the cause of insufficient functions or capacities. QA inspects the specifications after the functional design is completed, and tries to remove the causes of bugs before programming begins.

The main plans are:

(1)Enhance the quality of documents by making checklists for each product and each function

When defining the specifications, the designer makes checklists by extracting know-how from past development projects for each product and for each function. These checklists prevent insufficient discussion that omits points that need to be discussed.

For example, for a program to add new features of devices, a checklist consisting of the following points was created and it was used to make the specifications.

Hardware interface

- Differences from existing devices
- Existence of asynchronous interrupts
- Propriety of canceling input or output, and whether it is successful
- Resource sharing with other systems

etc.

(2)Reinforce the review process by establishing chief-reviewers who are responsible for each product and each function

To develop one function, components are often created in multiple departments or workgroups. In these cases, the different departments or workgroups should cross-check the specifications from an inter-component viewpoint in order to fit the function image and satisfy the customer's requests. Therefore, persons who have previous development experience for the product or designers who discussed the project in advance are selected as chief-reviewers for each component. Since the chief-reviewers attend all reviews for each specification, the quality of revues improve.

(3)Improve the quality of all software parts by establishing a period for reexamination based on the analysis of past failures.

In software quality, removing errors completely is almost impossible. So, after several upgrades, undetected errors accumulate in the new version. This failure potential left in programs may appear at a customer's site and cause some problem. To reduce such situations, we analyze past errors and determine, from both technical and management viewpoints, how to remove the cause of the errors and how to prevent later versions from failing because of inherited bugs.

Moreover, when developing an upgraded version, the working steps also include checking of all the software parts not just the upgraded functions. And the technical problems in the past versions are included in a checklist to detect similar failures.

2.2 Solution to Problem 2

Use a Quality Probe (QP) to check quality in the midstream stages of the programming process

A QP is a test of basic functions and is usually performed after testing the combined modules and programs.

Generally, in mass production, a QP means a test of a sample of all the products, with the ratio of defects in all the products assumed to be the same as in the samples. On the other hand, a software product is not a mass-production product, rather software is a single product with multiple functions. Therefore the checkable items for all the functions are treated as the total number of products in mass-production, and the ratio of bugs found in a check of a sample of the functions is assumed to be the same in all the functions.

Performing a QP after combining the modules and programs is a statistical method to get quantitative quality measurements at an early stage.

The purposes of a QP are:

- To perform an early combination test of the basic functions, and remove and modify insufficient functions and bugs related to basic functions.
- To evaluate the quality of the basic functions before a total test, and estimate the number of remaining errors in order to judge whether the quality satisfies the design goals.
- To reinforce frequent checking of items and jobs for weak functions which have many defects in order to prevent failures of detection.

QA sets conditions for performing a QP of a program. When the program clears the conditions, QA tests a sample of checkable items to evaluate the quality of the program.

(1) Setting guidelines for improving quality

The first QP (QP1) is performed before designers and programmers have finished their own program checking (when 90 to 95 percent of all the checkable items are finished). QP1 aims to set guidelines for improving quality. The QP1 is used to help define the target values for quality (the number of defects and additional check items, etc.).

(2) Confirming the quality of the program to be released to QA for inspection

If the quality of the program that is released to QA for inspection is bad, the program will need to be released to and be inspected by QA again and again, which delays the delivery period. So using a QP to confirm the quality of the program that is to be released to QA for inspection is important. But performing a QP many times should be avoided because it is the same as QA performing the inspection many times. Therefore, a QP should be performed up to two times at most.

The second QP (QP2) is performed to check the results of improving quality after QP1, and to confirm the quality of the program to be released to QA for inspection. The designers will insist that there are no defects in the program when they have finished all their own check items, therefore, QA needs to perform a QP at this time to confirm the quality of the program to be released for inspection.

2.3 Solution to Problem 3

Use SQS to enhance quality in the downstream stages of the programming process

Just before a software product is shipped, we install it in the SQS (Software Quality assurance System), which simulates user environments, and perform a total test. SQS is an original system for the QA Department.

The purposes of the SQS test are:

- To remove bugs concerning locking controls and synchronous controls among several functions. This is achieved by performing high-stress tests that create various complex timings. Such complex timings that occur infrequently in actual environments are created by simulating the status of acceleration limits. This enables the test-target program to be checked.
- To remove bugs caused by unusual processing. This is achieved by performing obstacle tests. These obstacle simulations can be achieved by introducing actual failures into the hardware, or by using software tools to simulate the failures. These tools are maintained to be flexible, and can be used not only to test whether a program can recover directly from a problem but also to test programs that are influenced indirectly by the failure.
- To measure quality by quantitative analysis. This is achieved by monitoring performance and reliability. Evaluation data accumulated in the past can be used when evaluating the performance and quality of the test target program.

Thus, the software designers and programmers use feedback about bugs detected by SQS to help detect similar bugs and to help improve the quality of their products.

3. Example

The example below show the effectiveness of the combination of the three quality control methods. These software products were developed in the Software Development Center.

3.1 WDCP/ES (Double Disk Volume Control Program/ Extended System): a software product for mainframe systems

WDCP/ES enhances reliability by writing to dual volumes and enables system operation to continue even if a disk error occurs. WDCP/ES also ensures better maintainability than a mirrored disk configuration. The sales point of this product is reliability and so no bug can be allowed. This product requires double disks (as illustrated in Figure 4), which means customers need to spend double their usual equipment investment. So, for this product, customers request a high quality level commensurate with their investment; in other words they are paying for "insurance".

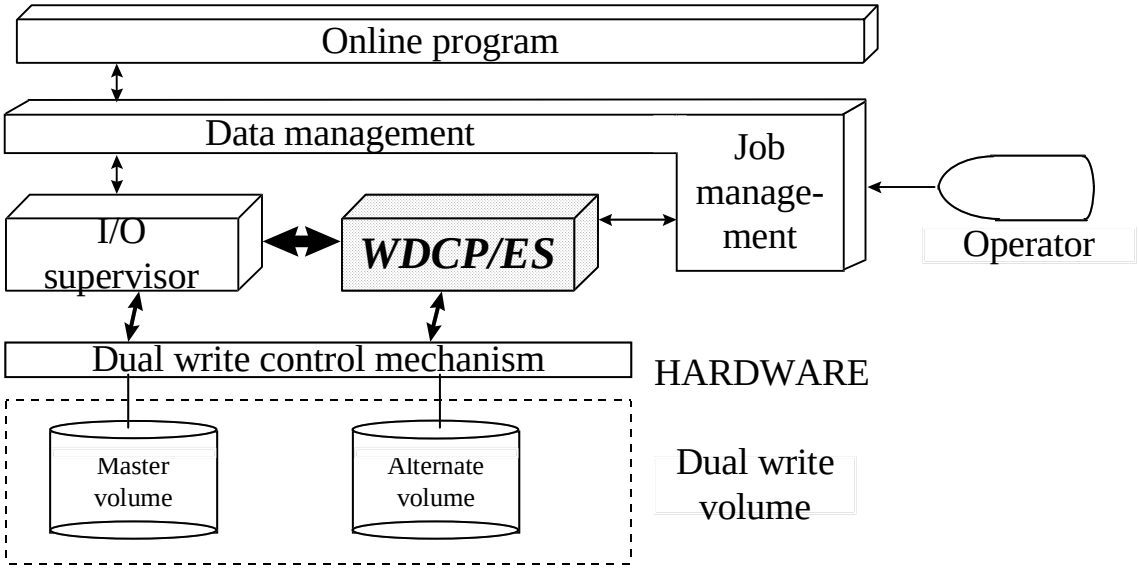


Figure 4: WDCP/ES Positioning

Effect

Table 1 compares WDCP/ES Version 7 with Version 11. The scale of the versions are about the same. Version 11 was developed from October to December 1997 in the Software Development Center.

Table 1: Comparison between Version 7 and Version 11

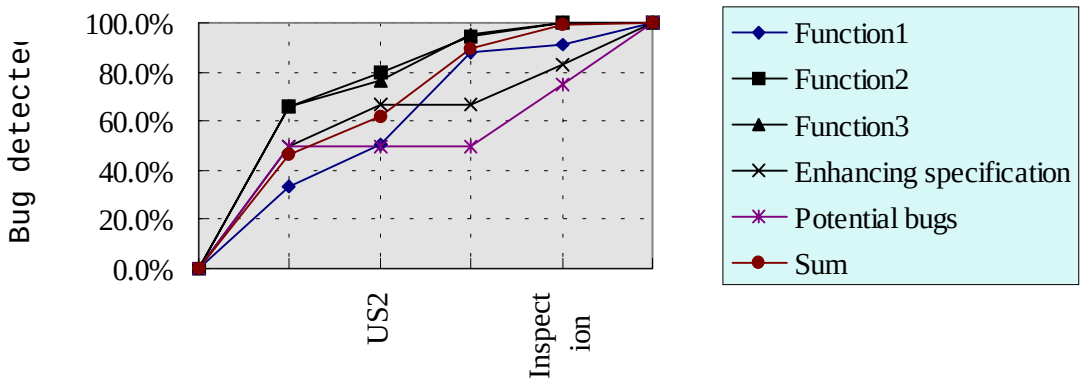
	Version 7	Version 11
Year	1991	1997
Cost (Designers)	35 (5 person * 7 months)	21 (7 person * 3 months)
Cost (QA)	7 (2 person * 3.5 months)	1 (1 person * 1 months)
Bugs removed upstream	47 %	94%
Bugs removed in QP	0 (Not performed)	0.35/Kstep
Bugs removed in SQS	4.52/Kstep	0.28/Kstep

Kstep:The measure of cording scale. 1,000 lines of coding = 1 Kstep.

The ratio of bugs removed in upstream stages was 94 % in Version 11, but 47% in Version 7. This was the result of enhancing quality in the upstream stages. In addition, the ratio of bugs removed in SQS in the downstream stages in Version 11 was 1/16 of Version 7. This reduced the development costs such as computer-related costs.

The period of development for Version 11 was half of the period for Version 7 because most bugs were removed during the upstream stages.

Figure 5 illustrates the error detection status in each stage of Version 11. In the upstream stages, the error detection of functions 1 to 3, which occupy 97% of all of the development scale, was successfully achieved; so as much as 99% of bugs were detected before inspection.



US:Upsteam stage

Figure 5: Error detection status

Relationship to the quality control methods

The following sections show how the quality control methods improved the product.

<Enhance quality in the upstream stages of the programming process>

We carried out the following to enhance quality in the upstream stages of the programming process.

(1) Enhancing the quality of documentation

Basically, we used the following checklists to enhance the quality of documentation.

(a) Checklist for creating the basic specification

When creating the specifications, the designer uses a checklist which lists points to be considered. Using the checklist during discussions can prevent omission of points in the specifications, and unify the quality of the documentation.

Table 2 shows an example of the checklist. All items are categorized roughly in the *Classification* column, and specific check items are listed in the right *Check item* column with a number. Items listed in the *Classification* column correspond to the table of contents of the basic specifications, and specific items listed in the *Check item* column are points to be checked in each chapter of the basic specifications.

Table 2: Example of a checklist for creating the basic specifications

Classification	No.	Check item
Overview	1	Clearly explain the purpose of support and background
	2	Does the purpose of development match the functions
	3	Do the required specifications match the required conditions
	...	...
Functions	9	Clearly define the functional specifications and the range to be made public
	10	Classify the functions into hierarchical layers
	11	Compatibility with other systems
	...	...

Classification	No.	Check item
Design	14	Is the design policy appropriate for implementing the functions?
	15	Are the relationships with other functions considered?
...	...	...

(b) Evaluation sheet for the basic specifications documentation review

Next, the created specifications documentation must be reviewed, and we evaluate the review.

An example of the evaluation sheet for the review is shown in theTable 3. The evaluation items and their importance are expressed numerically and used for pass/fail criteria for going to the next step.

Table 3: Example of an evaluation sheet used in a review

Evaluation item	Importance	Supervisor		Expected points by worker
		Evaluation	Points	
Review level				
• Are there enough attendees?	2	3	6	6
Functions				
• Do the functions satisfy the user's needs?	2	3	6	6
• Are the reasons for defining the functions clear?	2	3	6	6
• Are the external functions that users can see and internal functions described separately in the specifications?	0.5	3	1.5	1.5
• ...	...	...	...	...
Implementation procedures				
• Can the procedures adequately achieve the desired functions?	2	3	6	6
• Are the sufficiency of the functions discussed by applying a matrix?	1	2	2	2
• ...	...	...	...	...

Evaluation item	Importance	Supervisor		Expected points by worker
		Evaluation	Points	
Performance				
• Have the number of steps increased from the previous version?	1	3	3	3
• Have the number of inputs/outputs increased from the previous version?	1	3	3	3
• Has the size of memory increased from the previous version?	2	3	6	6
• ...	...	...	...	...
...	...	...	...	...
		Sum	92.5	92.5
		Criteria	OK	

Evaluation

- OK. No change needed. 3
- OK. Partial change needed. 2
- OK, but insufficient. Changes needed. 1
- NG. Must discuss again. 0
- Not applicable (3)

Formula for Points

*Points = Importance * Evaluation*

The shaded numbers of importance are *weighted items.*

Criteria

- If all the evaluations for the weighted items are equal to or greater than 2, and the sum of the points is equal to or greater than 90: Accepted
- If all the evaluations for the weighted items are equal to or greater than 2, and the sum of the points is between 80 to 89: Not accepted (The supervisor must decide whether to perform the review again.)
- If the evaluations for the weighted items contain a point equal to or smaller than 1, or the sum of the points is equal to or smaller than 79: Not accepted (Re-create the specification and perform the review again.)

(2) Appointing the overall reviewer

The device this product supports is a disk system, so the procedures to implement the basic function specifications were based on existing programs or on programs that were developed simultaneously. And similar functions in these programs were referenced to create this product efficiently. Therefore, a person who has experience in developing these existing programs or simultaneously developing programs was appointed as the overall reviewer. Thus, usable components could be diverted to this product during the review. As a result, commonality of the components was achieved between this product and the two controls in other programs in the operation system that were developed simultaneously, and these interfaces were defined clearly.

In addition to (1) and (2) above, we performed the procedures (3) and (4) for this product.

(3) Changing the development policy from a base-change method to an option-added method

Based on the analysis of past failures, we changed the development policy from a base-change method to an option-added method. In previous versions, the scale of the basic program was 140 K steps and we revised the basic program whenever WDCP/ES was upgraded. This method easily caused corruption of the basic program (we called it “degrading”) and decreased the reliability. In Version 11,

we aimed to localize the effect of revisions by creating any new function as a new component, and keeping revisions of the basic program to a minimum by changing the connection part of the basic program and the new functions.

(4) Taking countermeasures in the later process based on analysis during the development process

Based on analysis during the development process, we found the following tendencies:

(a) Simple coding mistakes and careless correction errors

(b) Interface failures between components because the existing module specifications were unclear

For problem (a) above, we performed a cross-check among software designers. For problem (b), we revised and created new specifications.

<Use a Quality Probe (QP) to check quality in the midstream stages of the programming process>

We detected three bugs by the quality probe. Based on the analysis of these bugs and similar bugs, we detected 16 potential bugs before the program test.

We also checked the operation of the basic functions that were supported for the first time by Version 11, and proved that there was no problem. Thus, errors immediately after installation or errors detected in basic functions, described previously in problem 2, were prevented.

In this version of the product, there were three main functions. Table 4 shows the error detection status on each function. The error detection status of QP indicated the quality of Function 1 was bad, and the error ratio of the final records also indicates the quality of Function 1 was bad. This proves the effectiveness of the QP.

Table 4: The error detection status on each function

Development item	Development scale	Upstream stages	QP	Inspection	Final record
Function 1	5.05	222	2	2	254
Function 2	2.75	70	0	0	74
Function 3	2.75	92	0	0	97
Enhancing specification	0.20	4	0	1	6
Potential bugs	0.00	2	1	1	4
Sum	10.75	390	3	4	435

<Use SQS to enhance quality in the downstream stages of the programming process>

In the past, WDCP/ES caused a serious failure due to a hardware failure at the site of a large-scale user. This was due to a failure in locking. When a hardware failure occurred, WDCP/ES shut down the target device, however, when the target device was a disk, which was connected to multiple systems, simultaneous access had already occurred. The action of the errors in a dual write volume is illustrated in Figure 6.

In Version 11, we checked the error recovery facility during locking by multiple systems. We performed these checks in the Software Quality assurance System (SQS). In the SQS environment, we also checked the operation and performed error recovery tests for 10 types of hardware controller disks supported by WDCP/ES.

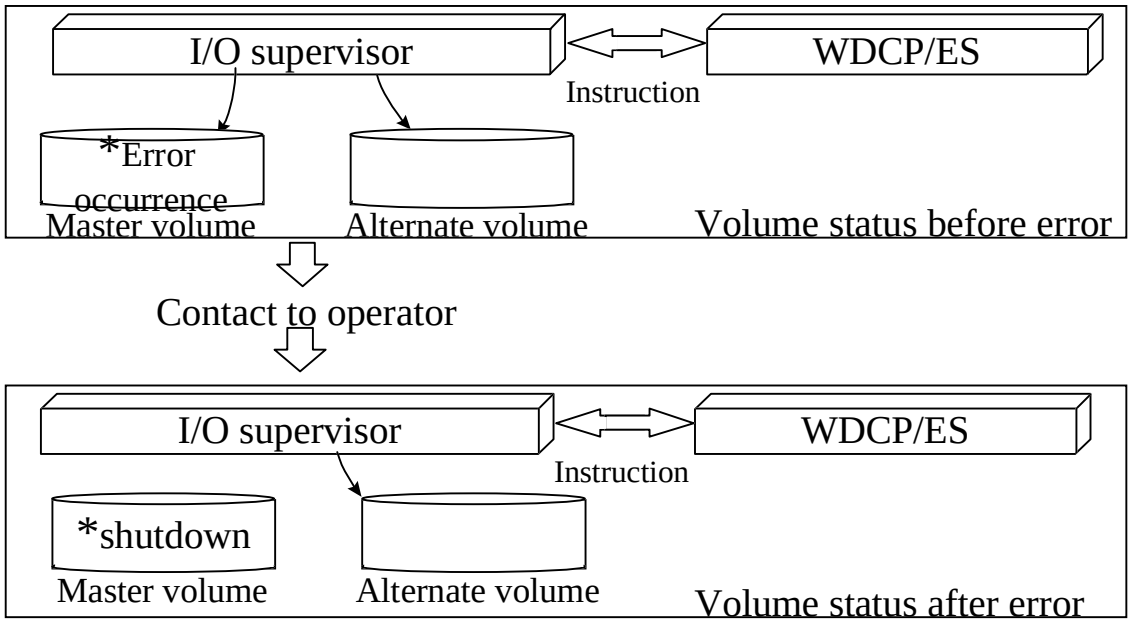


Figure 6: Error in dual write volume

3.2 HiRDB: a software product for Client/Server systems

HiRDB is a high-performance and high-quality Relational Database Management System (RDBMS) for workstation servers or PC servers. HiRDB supports both stand-alone servers and multi-parallel servers.

SQS in client/server systems can:

- continuously maintain a large-scale configuration that uses the latest machines
- connect various I/O devices (such as MT and DAT) and networked machines
- continuously monitor and cover the main users' machine configuration

If SQS has many components, we can test it in an environment similar to the user's environment and prevent the occurrence of trouble just after installation at a user site.

According to this policy, we have increased the machines in SQS as Table 5 shows.

Table 5: Enlarged SQS for client/server systems

	Configuration # 1 (1995/2 -)	Configuration # 4 (1998/2 -)
CPU	16 nodes * PA-7200 (120MHz)4Way	8 nodes * PA-8000 (180MHz)10Way 12 nodes * PA-8000 (180MHz)4Way 5 nodes * PA-8000 (180MHz)4Way
HD	400GB	2700GB
Connection between nodes	5 * CD10LAN 1 * FDDI 4 * High-speed communication device	(In addition to the configuration #1) Channel connection Optical channel connection

Effect

Table 6 shows the evaluation of HiRDB from the same viewpoints as WDCP/ES.

The quality level of HiRDB is similar with WDCP/ES Version11.

Table 6: HiRDB quality status

	Version 1
Year	1995
Cost (Designers)	840 (70 person * 12 months)
Cost (QA)	84 (7 person * 12 months)
Bugs removed upstream	92.9%
Bugs removed in QP	0.35/Kstep
Bugs removed in SQS	0.11/Kstep

To evaluate HiRDB quality, we performed the following tests:

- Evaluated all points of HiRDB in a 16-parallel-server system (assuming a large-scale client system)
- Evaluated performance in databases (from 20 GB small databases to 2 TB large databases)
- Evaluated reliability by running continuously over 200 hours

With these evaluation results, we could verify that we passed our primary quality targets.

4. Conclusion

Because most software bugs are caused in the upstream stages, some action to prevent them should be done during these stages. However, effective methods to improve the quality of software during upstream stages were insufficient.

To resolve the problem, we applied all the quality control methods during the upstream, midstream, and downstream stages. This enabled us to improve quality. We could also improve quality by building a center that can provide an environment similar to the user environment, analyzing the bugs there, and feeding back the analysis results to workers during the upstream stages.

Reference

Katsuyuki Yasuda, ”The thinking and fact of Software Quality Assurance, Nikka-giren”,pp.29-37,Nov. 1995(in Japanese)

Introduction and Evaluation of SPA Method for Telecommunication Software Development

Ichiro Kamata, NTT Communicationware Corporation, Japan
Sadayuki Higasi, NTT Communicationware Corporation, Japan
Fujio Moriyama, NTT Communicationware Corporation, Japan

Abstract

It is important to develop high quality telecommunication software and to maintain its quality level for public use. For the last twelve years, we have been trying to improve the quality and productivity of telecommunication software development. Our efforts have been based on improving software development processes using mainly development data. However, we realized that it would become difficult to meet the increasing annual demand by improving only software development processes. We understood that we also had to improve development management processes.

We paid attention to development management processes and established a customized SPA (Software Process Assessment) method, TMF (Telecommunication software Management-level Framework), which also uses development data. We created the TMF for application to actual activities of telecommunication software development projects, referring to SEI-CMM. However, the TMF requirements were based on our working standard for development management. Most projects that had introduced the TMF improved quality and productivity. This paper explains and evaluates the TMF application.

Ichiro Kamata, Engineer, an assessor based on the SPA method (TMF), and an internal quality auditor based on ISO9000s.
Quality Assurance and Management Department, NTT Communicationware Corporation,
NTT Makuhari Bldg. 1-6 Nakase Mihama-ku Chiba-shi, Chiba, 261-0023, Japan
Phone: +81 43-211-3539, Fax: +81 43-211-5162

1. Introduction

For the last twelve years, we have been trying to improve the quality and productivity of telecommunication software development. We had mainly used development data and our efforts had been in carrying out improvements to software development process until we introduced SPA (Software Process Assessment) method. The improvements tended to be in areas where there was no consciousness of the development process, e.g., such as testing in lower process. These efforts certainly produced good results.

In those days, communication network services were increasing rapidly. In addition, we found out that the activities in the telecommunication business world would be deregulated gradually. We predicted that competition between the telecommunication business world would become keener. We thought that improving both the quality of products and productivity of development was indispensable to win the competition. Moreover, we judged that it would be difficult to win the competition through only improvements to development processes based on development data.

At that time, SEI-CMM and ISO9000s were becoming popular as a trend in the industrial world. The designers of these models paid attention to software process. Therefore, we had to pay attention to development management process, which we had hardly done. Consequently, we decided that we would carry out improvements to quality and productivity by paying attention to this process.

2. The features of a customized SPA method

The activities of a development project involve the quality of products and productivity of development. Moreover, development projects had been getting down to improvements to the development processes through TQM (Total Quality Management) and other activities by themselves. Therefore, we devised a customized SPA method that could be applied individually within the project.

Besides, the development projects until then had been trying to improve development processes mainly from the viewpoint of development technology. Because of the efforts, the development projects improved quality and productivity. Moreover, most problems that occurred under development were solved by development technology. Therefore, we judged that the projects had higher development technology than management technology. In addition, we could not understand what management technology the projects had had. Consequently, we decided to establish the SPA method that paid attention to development management processes.

We created the customized SPA method for application to actual activities of telecommunication software development projects, referring to SEI-CMM. The customized SPA method pays special attention to development management processes of a project in all its processes. Requirements of the customized SPA method were based on our working standard for development management. We have called the customized SPA method TMF

(Telecommunication software Management-level Framework• . Figure 1 shows the applicable coverage of TMF.

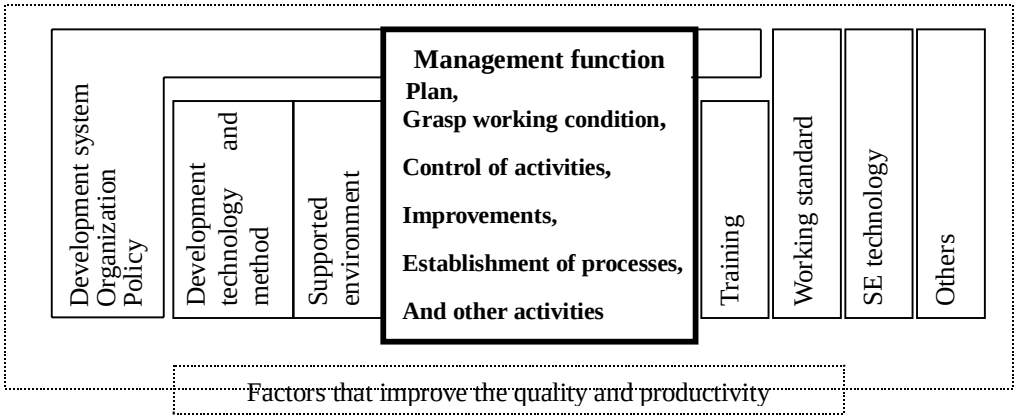


Figure 1: The applicable coverage of TMF

TMF has four major functions: estimation, planning, schedule management, and quality management. By applying TMF, we try to understand the level of the project's management processes and improve the development management processes. The purpose of introducing TMF is to improve the quality and productivity of software development by enhancing the ability of development management.

2.1 The methods of assessment

There are three major features in the methods of assessment.

2.1.1 Assessment of relationship between development management processes and development data

The purpose of introducing the SPA method is to improve the development results. In other words, it is to improve the quality of products and productivity of development. Therefore, it is necessary to assess not only development management processes but also development data. Moreover, we were afraid that projects' members might only pay attention to improvements to processes if we should have assessed only processes. The reason for this is that improvement to processes is a means. We grappled with improving processes to improve quality and productivity.

TMF assesses both development management processes and development data. The results of these assessments fall into six steps: from level 1 which means a state without management to level 6 which means the most suitable state. Assessors investigate the causal relationships between the process and the data. Then, they propose improvements to management processes to achieve the purpose of quality and productivity. Figure 2 shows the assessment items and the management levels.

Assessment Items \ Level		1 Self-management	2 Initial	3 Repeatable	4 Control of work	5 Control of management	6 Optimizing
Management processes	Estimation function	- Objects of assessment and improvement limited in the development management processes - Assessment on both sides of the management process and the development data - Management level classified into six steps (level 6 is an ideal type)					
	Planning function						
	Schedule management						
	Quality management						
Development data	Difference from plan and results						
	Productivity , cost						
	Quality, resource						
	Schedule, others						

Figure 2: The assessment items and the management levels

TMF was created referring to SEI-CMM. However, the TMF requirements were based on our working standard for development management. Moreover, the criteria of management levels were based on our own characteristics. Furthermore, the number of management processes, which TMF requires, is fewer than their requirements and its range is narrower too. Therefore, We did not compare these management levels with CMM or ISO9000 exactly. However, we can say the following as a general idea of the comparison.

- TMF levels ranged between two and five are equivalent to CMM levels ranged between two and three.
- TMF levels ranged between three and five are equivalent to requirements of ISO9000.

Figure 3 shows the criteria of management levels.

Level		The point of criteria
1	Self-management	Depending on each personal judgment
2	Initial	Applying rules for development management and managing roughly
3	Repeatable	Observing rules for development management and managing smoothly
4	Control of work	Carrying out continuous improvements to development processes
5	Control of management	Carrying out continuous improvements to development management processes
6	Optimizing	Spreading their own improvements throughout other projects

Figure 3: The criteria of management levels

2.1.2 Comparison of primary management process assessment, based on intuitiveness, and management process assessment, based on fact-finding

There are two methods of measurement in TMF. The first is the primary measurement of management processes based on intuitiveness. This is a method of measuring the processes simply and easily. In other words, this method is measurement based on the image of the assessors. This method can only be applied to members of a development project that appreciate the actual development activities. Therefore, only self-assessment should make it possible to use this method. The second is the measurement of management processes based on fact-finding. This is a method of measuring the processes accurately. This is a measurement based on the documents, data, and records of each management activity, which we looked on as the actual evidence of the activities.

Besides, assessors also measure development data in the same way as the management processes. Assessors compare the results of measurement based on these two methods of measurement, and reflect the comparative results on the final assessment and the extraction of proposals to improve the management processes. Figure 4 shows the relationship between methods of measurement and assessment.

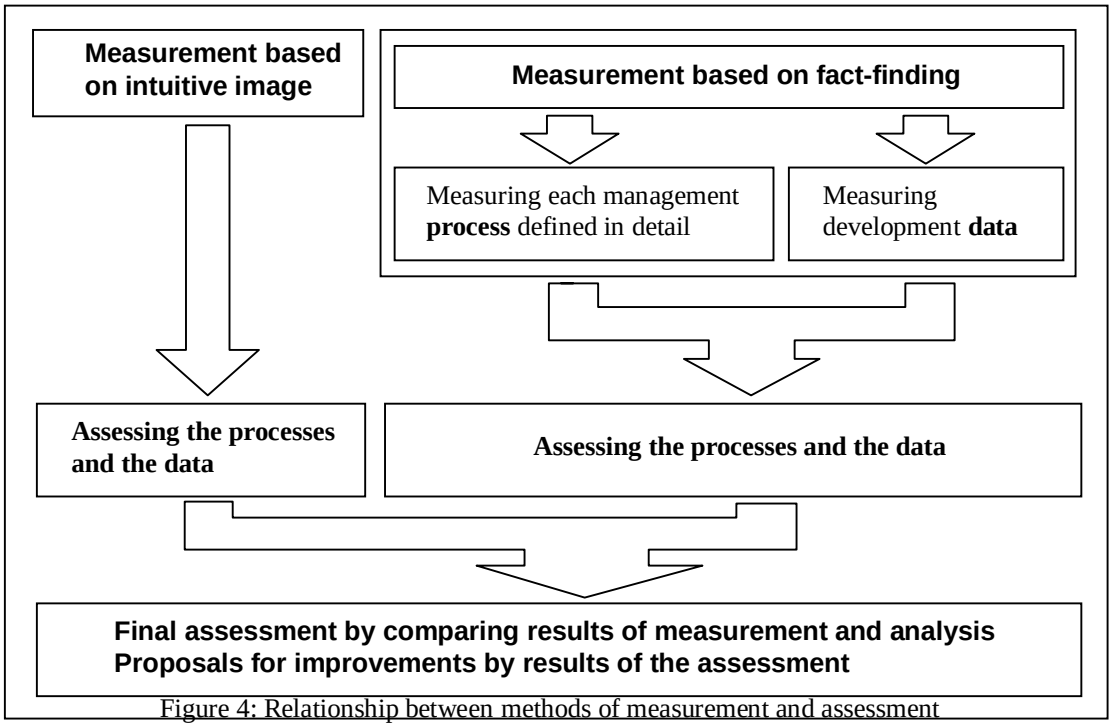


Figure 4: Relationship between methods of measurement and assessment

2.1.3 Assessment of state of management process using detailed management activities

When we assess development management processes based on fact-finding, we use this method which defines development management processes in detail. This method supports measurement of the state of development

management processes.

Development management processes were too abstract for us: e.g., the process to remove bugs found under development with no omission and the process to ensure that specifications' changes were incorporated in products. In addition, we thought that it was too difficult for us to measure development management process. It was because we thought this measurement would require assessors to acquire a good knowledge of development management processes. We desired that assessments would be carried out by self-assessment, i.e., projects' members assess themselves. Therefore, we tried to devise a method that supported measurement to reduce the burden on the assessors.

The method of support was to clarify development management processes by defining every general development management activity in detail such as each component unit of management. Assessors investigate whether a project is carrying out a management activity that consists of its component units. If they could confirm that all component units of the activity are being carried out, assessors systematically judge that the project has the development management process. Moreover, this support made it easy for assessors to make concrete proposals to improve management processes.

2.2 The methods of application

We applied TMF to development projects. The method of application was self-assessment. Assessment was carried out by self-assessment, i.e., project members, consisting of project managers, leaders, and programmers, assessed themselves. That is, all members of the project participated in assessment, as in a TQM activity. We think that this method of application unifies all members of the project. Moreover, we think that it is generally easy for Japanese to accept such a method where all members of the project participate.

However, there are two methods of application except self-assessment. The first is assessment carried out by other development projects. This method is carried out mainly to motivate projects' members to improve their own processes to a higher level. The second is assessment carried out by assessors who belong to Quality Assurance and Management Department. This method is carried out mainly to support new assessors, i.e., assessors teach new assessors how to assess the process and data. However, a fault of these methods is that it is difficult to measure management process based on the image of assessors because they are not familiar with actual activities of the development project. Therefore, project members must do their own measurement based on the image after all. Consequently, these methods are secondary methods and self-assessment is fundamental method.

3. The results of assessment of development management processes

The assessment of management processes is done based on four major functions: estimation, planning, schedule management, and quality management. We assessed seventeen projects in total. To research the features of the

management processes of projects, we analyzed the results of assessments with the following processes.

(1) The classification of proposals for improvement based on the management processes, the proposals for improvement fall into about four categories: the categories are much the same as TQM categories, i.e., Plan, Do, Check, and Act.

(2) The total of classified proposals for improvement, i.e., the total of projects that should carry out the same proposal for improvement.

(3) The percentages of projects that should carry out the same proposal compared to all projects.

The number of proposals for improvement varied according to a project range from three to five cases. However, assessors made a proposal to improve an activity that could not be carried out in lowest level prior to making a proposal in high level. It was because TMF could not recognize a quality system in next high level if a project have not established a quality system in low level. Therefore, we judged that this data would be able to make us grasp the state of the management process for each management function.

This data showed the percentages of projects that should carry out the same proposal compared to all projects. In other words, the data indicated that the proposal of high percentages was projects' common delicate and weak management process. These proposals for improvement were common proposals with many projects. Therefore, a proposal which was characteristic of a few projects was not contained in this data. We will introduce this data in the next section.

3.1 Features of management processes for each management function

3.1.1 Estimation function

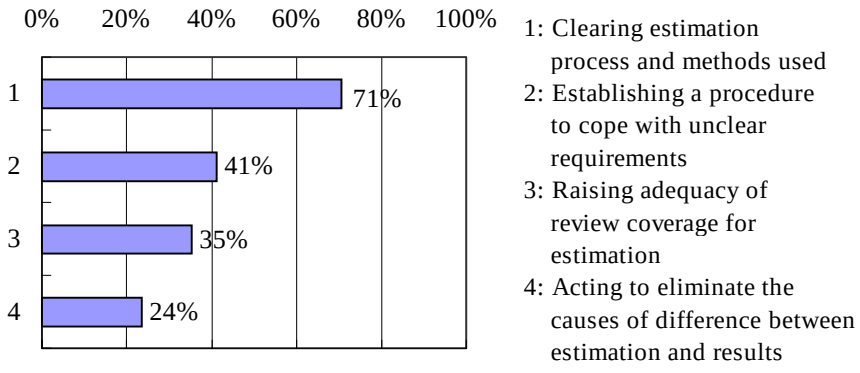


Figure 5: Proposals for improvement in the estimation function

Figure 5 shows proposals to improve the management process in the estimation function.

The features of management processes, which were why these improvements were proposed, were as follows.

- The management did not function fully with undocumented estimation processes and estimation by specific persons having a good skill.
- As a personal good skill was not documented as an organization skill of a project, it was difficult to improve not only estimation process but also methods used.
- The procedure to cope with unclear requirements was unclear.

3.1.2 Planning function

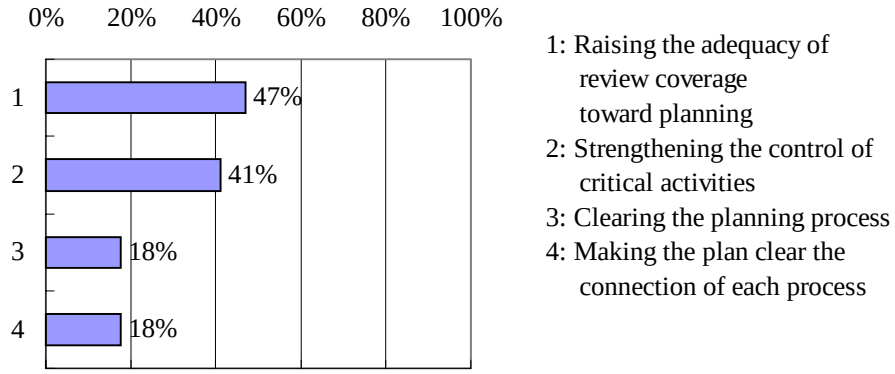


Figure 6: Proposals for improvement in the planning function

Figure 6 shows proposals to improve the management process in the planning function.

The features of management processes, which were why these improvements were proposed, were as follows.

- The procedure to review a plan was unclear.
- The procedure to grasp pending cases and critical activities was unclear.

3.1.3 Schedule management

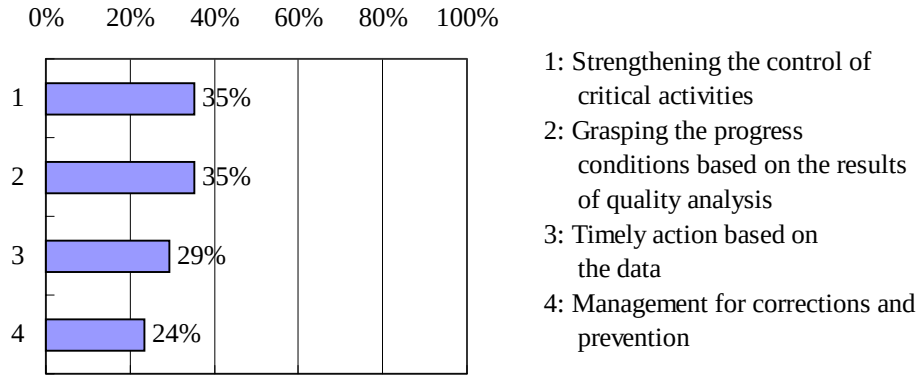


Figure 7: Proposals for improvement in schedule management

Figure 7 shows proposals to improve the management process in schedule management. There were few common

proposals with many projects in the area of improvement. We thought that schedule management was easier than other management functions about grasp of development conditions. This is because it was easy to define the data to be collected, and many projects had a rule that data were regularly collected. Therefore, many projects had fundamental management processes, from level 2 to level 3, in the schedule management. The features of the management process were as follows.

- Most projects had the necessary fundamental processes for schedule management.
- The procedure to control development activities corresponding to progress conditions was unclear.
- Effectiveness of procedures for corrective and preventive action based on schedule management data was low.

3.1.4 Quality management

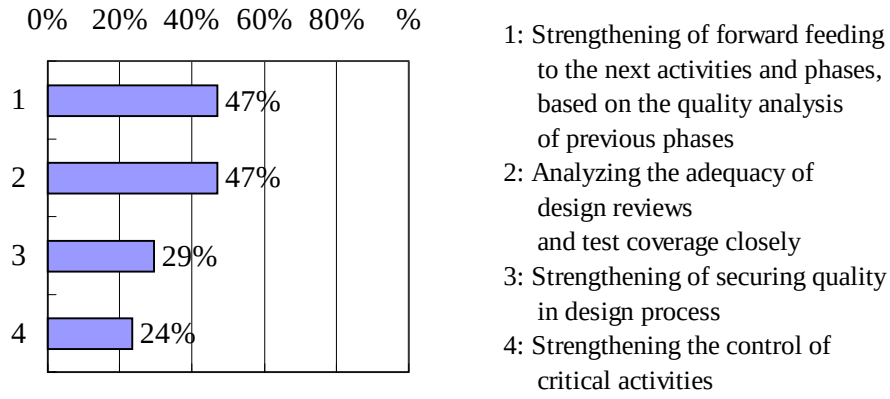


Figure 8: Proposals for improvement in quality management

Figure 8 shows proposals to improve the management process in quality management.

The features of management processes, which were why these improvements were proposed, were as follows.

- The procedure of forward feeding to development activities using the results of the quality analysis was unclear.
- The procedure to find bugs was clear, while the procedure to not make bugs was unclear.
- The analysis procedure for found bugs was clear, while the analysis procedure to understand the adequacy of design reviews and test coverage was unclear.

3.2 The tendencies of management processes in all projects

TMF showed the following tendencies in all projects that introduced TMF.

(1) Development processes and development management processes were not closely combined.

(2) Some development activities strongly depended on personal knowledge.

(3) The project plan was not fully checked at the beginning of the project, and neither was estimation.

(4) Management based on importance of development activities was insufficient.

(5) Most management activities started when problems occurred.

Many projects attached more importance to development technologies than management technologies. There were many individual skills in the development technologies. Most problems that occurred under development were solved by individual skills.

3.3 The effectiveness of assessment based on TMF

An analysis of assessment based on TMF clearly showed that the management process on a quantity basis that could not be attained by existing analysis approaches based on the development data.

TMF proposed effective improvement to the development management activities by comparing development data with management process. Moreover, TMF provided a concrete approach to improve quality and productivity. Furthermore, TMF suggested necessity of the management process to projects that depended on individual development skills too much. This means that the development and management technologies mutually complement each other's weaknesses. We expect this to contribute to a smooth progression to the development activities.

4. The effects of using the SPA method and the future subjects

4.1 The effects of using the SPA method

The necessity of introducing the SPA method was discussed. In particular, the relationship between the development process and the development data was repeatedly reviewed. Most members who intended to participate in the assessment understood the concept of process assessment.

At first, self-assessment was not accepted by development project members. It was because most members tended not to like assessing themselves. However, members of the project and the Quality Assurance and Management Department discussed the matter, and the SPA method was applied to the project. Project members were motivated by such discussion and came to understand the following. It was necessary for them to improve software quality. In order to achieve quality improvement, improvement to management process was necessary. Moreover, to improve processes, it was necessary to assess their own project objectively. It was because the project members themselves knew the actual activities of the project best.

The effects of using this SPA method in our organization were the following.

(1) Projects paid attention to not making bugs and carried out improvements to design process. Therefore, the number of problems that occurred in the field decreased to less than ninety percent quantity of previous projects. Figure 9 shows the number of problems that occurred in the field after shipment. X is a version of the product that was developed after improvements that were carried out with the SPA method by the projects.

(2) Productivity of most projects was over twenty percent more than quantity of previous projects, while a few

projects maintained previous productivity. Figure 10 shows the productivity of the projects. In the same way as figure 9 showed, X is a version of the product that was developed after the improvements.

(3) All members of the project participated in TQM style assessment, shared the results of assessment, and came to understand what development management was.

(4) Process improvement could be performed from the management level, not from development data.

(5) All members were motivated to improve their own processes to a higher level and to act by themselves. This is very important.

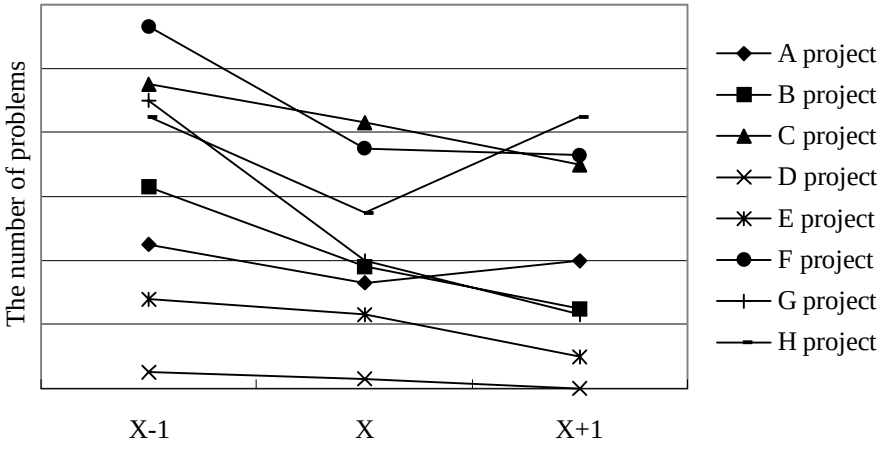


Figure 9: Quality after shipment

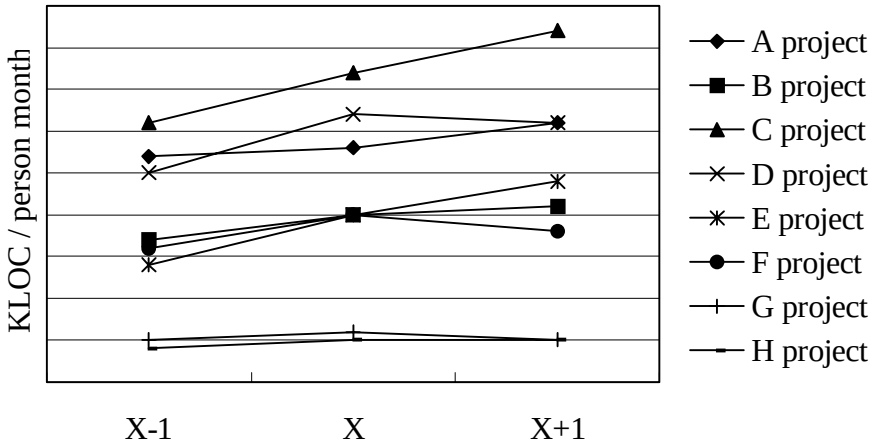


Figure 10: Productivity (KLOC: Kilo Lines Of Code)

4.2 Subjects for future study

We suggested improvement of the management process from the point of view that there was a relationship between the development management process and development data to the development projects. This was because we wanted to improve the development results, in other words, quality and productivity, by improving the

management process.

However, all development projects have been carrying out various actions up till now to improve quality and productivity; e.g., the application of new development methods, the introduction of good development tools, and the training for human skills. Therefore, we should try to prove that this improvement of management processes, which is one action, positively contributes to improving quality and productivity. We believe this proof would encourage project members to use the SPA method. As subjects for future study, we should show the effectiveness of process assessment on a quantity basis; good processes produce good products.

5. The unexpected effect brought about by the SPA method

We have been doing activities to seek ISO9000s certification up till now. Some projects have already obtained certification. The SPA method has been useful in certification activities. In particular, the SPA method has made it easy for us to understand both the processes and their purposes that ISO requires. Consequently, projects that have introduced the SPA method have had less trouble with activities to seek certification than ones that have not done this. Figure 11 shows the number of nonconformity reports from internal quality audits or preliminary review before obtaining the ISO9000s certification.

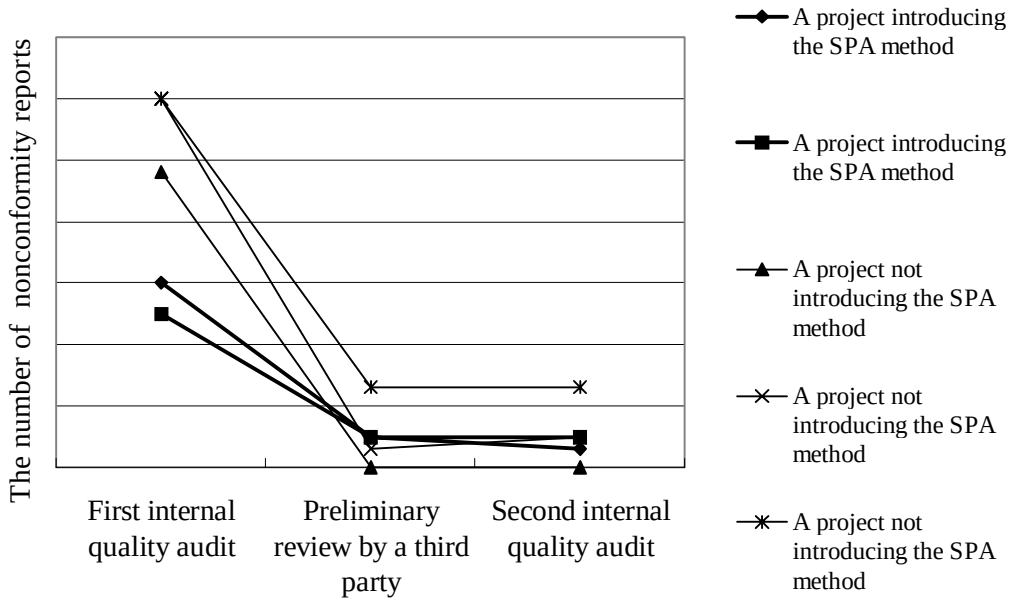


Figure 11: The unexpected effect

6. Conclusion

The projects that introduced the SPA method have been developing telecommunication software. However, all the projects were in our company. Therefore, the features of management processes shown by the SPA method might

be only in our company. Moreover, we may not be able to say that there were these features in the telecommunication software field.

Nevertheless, it was a fact that the projects had higher development technology than management technology. Moreover, the projects were developing software with paying more attention to development technology than management technology. In addition, there were many individual skills in the development technologies. In such a condition, most problems that occurred under development were solved by the development technologies. We call such a development style the Development Technology Lead Type.

Therefore, we predict that results of assessment will be similar to our features when the SPA method applies to a project of the Development Technology Lead Type even if the project belong to other company. Of course, requirements of the SPA method should be tailored when the SPA method applies to the project which belong to other company. It is because the requirements are based on our working standard for development management.

7. References

[1] Humphrey, W. S. *Managing the Software Process* (Japanese translation), JUSE Press Ltd. 1991(in Japan)

A Process Improvement Method Using Software Quality Analysis Tools

Takeshi Tanaka, Minoru Aizawa, Hideto Ogasawara,

Takumi Kusanagi, Atsushi Yamada, Hideo Nakamura

Systems & Software Research Laboratories, Toshiba Corporation

70, Yanagi-cho, Saiwai-ku, Kawasaki-shi, Kanagawa 210-8501, Japan

+81-44-548-5480

{takeshi, aizawa, hideto, kusana, yamada, nakamura}@ssel.toshiba.co.jp

ABSTRACT

Quantitative evaluation of software development process is important, and software analysis & measurement tools constitute one of the approaches to the accomplishment of this task. However it is often difficult to feed back information on analysis results to inter- and intra-phases because of the lack of know-how and time restrictions in the actual development departments. In order to overcome the problems, we propose the support activity of software quality analysis & measurement service. This is performed by HQC (High Quality software creation support virtual Center), an independent staff group within our company. Some effective case studies were obtained as follows:

- Extracted modules by tools contained about 40% of all the problems detected by review and testing.
- 56% of faults were detected before testing.

HQC activities employ quantitative methods and are applicable to improvement of software development processes. Therefore HQC activities implement new software processes thereby endowing process control practice with a greater degree of quantitativeness, for example, a higher level of CMM.

AUTHORS BIOGRAPHY

Takeshi Tanaka is on the Software Quality Assurance Team of Systems & Software Research Laboratories at Toshiba Corporation. He received the M.Sc. degree in information sciences from Tokyo Institute of Technology. His current interests are software quality assurance and software metrics.

Minoru Aizawa is on the Software Quality Assurance Team of Systems & Software Research Laboratories at Toshiba Corporation. He received the M.E. degree in information and computer sciences from Osaka University. His current interests are software configuration management and program static analysis.

Hideto Ogasawara is on the Software Quality Assurance Team of Systems & Software Research Laboratories at Toshiba Corporation. He received the M.E. degree in management science from Science University of Tokyo. His current interests are program static analysis, software testing, measurement, and reliability.

Takumi Kusanagi is on the Software Quality Assurance Team of Systems & Software Research Laboratories at Toshiba Corporation. He received the M.E. degree in mechanical engineering from Waseda University. His current research interests are software inspection, process improvement, and CMM.

Atsushi Yamada is on the Software Quality Assurance Team of Systems & Software Research Laboratories at Toshiba Corporation. He received the M.E. degree in electrical engineering from Osaka University. His cur-

rent interests are software quality model, metrics, life-cycle processes, process assessment and human factors. He participates a international standardization activity of ISO/IEC JTC1/SC7 software engineering committee. He is also member of the IEEE and the ACM.

Hideo Nakamura is a Laboratory Leader of Systems & Software Research Laboratories at Toshiba Corporation. He graduated from the graduate school of Engineering in Tokyo University, Dr-Eng.. His current interests are Software Engineering, CASE Tools for Firmware and HW/SW Co-design.

1. INTRODUCTION

In order to develop a large-scale complicated software in a short period, it is important to construct a framework for the software development and to improve the development processes [6]. We have already reported on software process assessment and improvement activities using CMM (Capability Maturity Model), which involves such departments as software engineering, administration, and quality assurance [1]. We also constructed a model for software quality using quality metrics, and reported the obtained results when we applied it to actual projects [2]. We also proposed a software metrics for GUI (Graphical User Interface) [3].

A software development process can be divided into several phases such as design phase, coding phase, and testing phase. For each phase there is a corresponding unit of software quality assurance. It is effective to utilize software analysis & measurement tools in order to improve the software process and quality. However several problems may occur and the followings are common problems.

1. It takes a considerable time before everything is settled down, for example, setting up a tool and how to use it. Moreover most development departments often don't have enough time.

Therefore time required for the SQA process must be reduced. Number of re-work loops must be decreased. An example is presented in [4].

2. Know-how (e.g. analytical skill, knowledge, method of using tools, or utilization of measured data by tools etc.) can't be accumulated and fed back effectively. Important know-how is often accumulated through long experience. In this part accumulation means to store know-how.

Therefore accumulation and feedback of know-how must be obtained.

Once it is understood how to improve development processes and how to use tools, it becomes easier to achieve higher productivity and better quality. The development departments need continuous & concurrent support from an independent staff group such as the administration department and the quality assurance department. The support activities are useful for obtaining such know-how. In this paper we propose process improvement activity, using software quality analysis & measurement tools. Service activities are executed by HQC (High Quality software creation support virtual Center), which is an extension of the SEPG (Software Engineering Process Group) [7] concept because we try to improve software processes by measuring and analyzing products. Measurement values respecting software development processes are collected and analyzed, and resulting in the improvement of software processes in [8].

In 1996, we started HQC activities [4] [5]. Our activities enabled quantitative evaluation of software process improvement mentioned in [9] and, as a result, endowed process control practice with a greater degree of quantitativeness, for example, a higher level of CMM.

Figure 1 shows the relation between product quantitative evaluation and software process improvement [9]. Measuring software quality by HQC leads product quantitative evaluation with check point in phases or sub-phases, so software development process can be evaluated quantitatively. Software product quality can be measured quantitatively in sub-phases of a phase and at the end of each phase. In our approach, the following items are provided for check points.

- measurement of product quality
- the extent of resolution
- problems detected by product quality analysis

In the following sections, we describe the activities of our group.

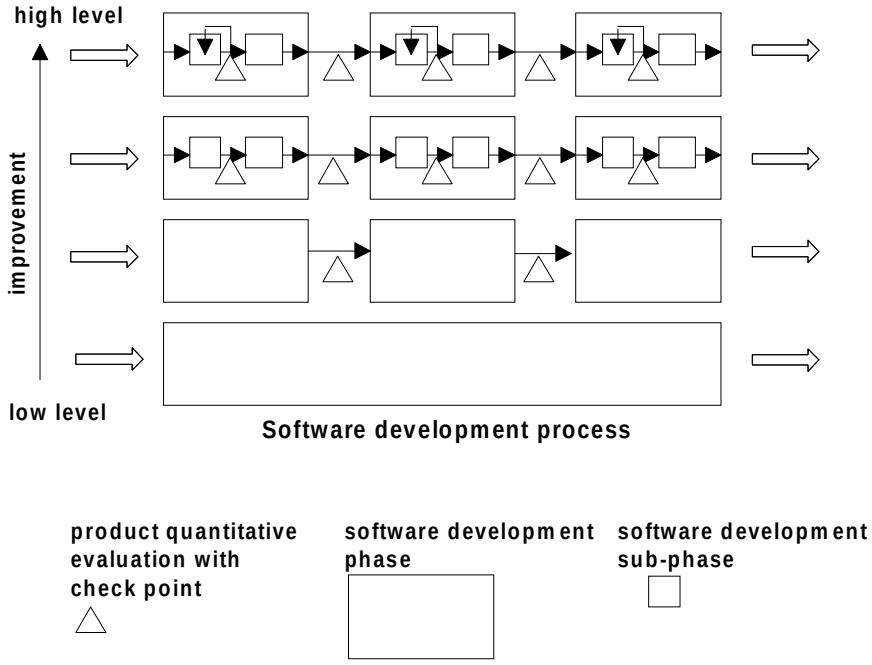


Figure 1 The relation between product quantitative evaluation and software process improvement

2. HQC ACTIVITY

In this chapter, we explain HQC activity.

2.1. Summary of HQC

New methods and techniques must be gradually introduced according to necessity, for example, alteration of work flow in software development process, and utilization of new tools or their analytical results. If there is an expectation that the introduction of a certain technique will yield good results, the technique should be applied to multiple software development departments. However, the following issues must be resolved in order to obtain good results.

- Members of the development department don't have enough time to evaluate the effectiveness of tools and utilize analytical data.
- New methods and techniques are sometimes difficult to acquire.
- Know-how of development departments respecting improvement of processes and utilization of analytical data is usually insufficient because of a shortage of experience.

- It is not easy to grasp the relation among know-how within only one development department.

The first two issues are related to point 1 in section 1, and the third and last two issues are related to the point 2 in section 1.

Therefore we started service activities as an independent staff group to improve software processes in software development departments. Generally know-how in development departments isn't shared effectively. In order to share know-how, the activities conducted by an independent staff group are useful.

The activities are conducted by HQC. Figure 2 shows the relation between HQC and the development departments.

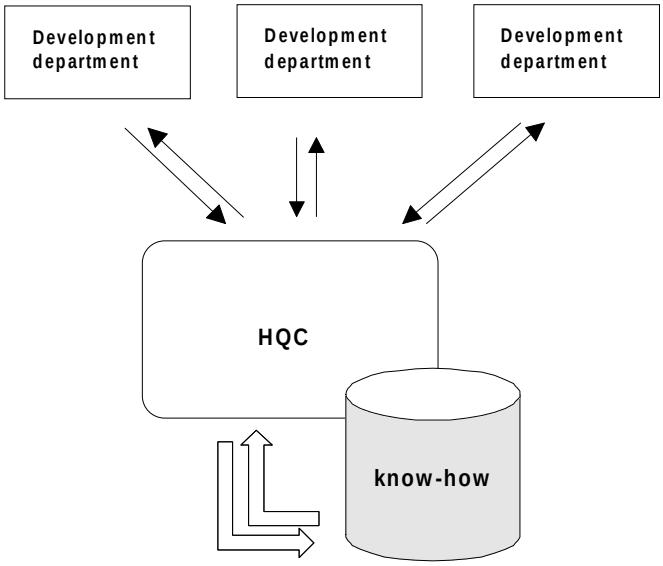


Figure 2 The relation between HQC and the development departments

HQC has been supporting the following services.

[Step 1] SPA (Software Process Assessment)
[Step 2] Program analysis service

Step 1, which is related to process improvement, makes it possible to grasp the present status of the development department. Step 2, which is related to quality of the product, make it possible to measure software quality. We can start our service from each step. We repeat Step 1 and Step 2, and improve software process and quality. The following viewpoints are useful for improving software processes and for increasing the efficiency of review or testing. If two viewpoints are adopted, our activity is accomplished (See Figure 3) . Ratio or numbers in two viewpoints are determined according to each software development.

[viewpoint 1] The ratio of review period in a phase, the number of reviews in a software process and problems detection rate

[viewpoint 2] The ratio or number of detected faults in an early phase, especially the value in integration testing phase, system testing phase or site operation testing phase.

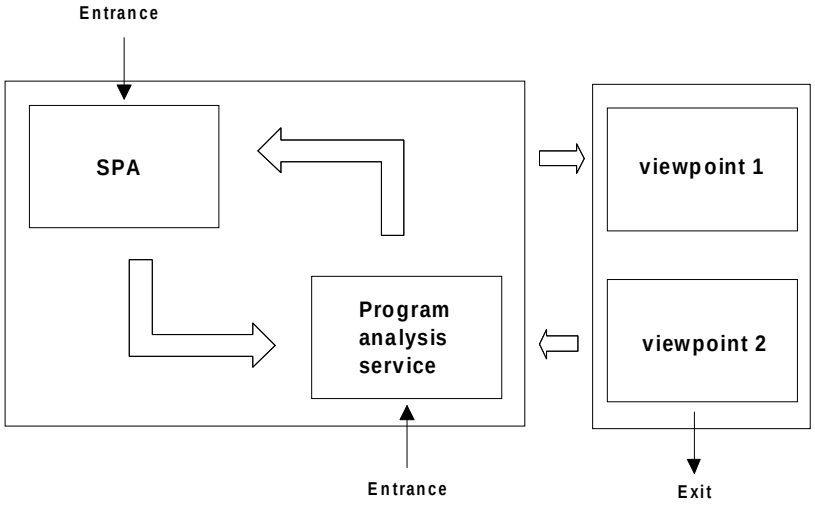


Figure 3 Workflow in HQC

Most faults are usually detected in a testing phase. However, if faults are detected in an early phase, the amount of re-work is decreased and the thoroughness of review contributes to product quality.

If viewpoint 1 is not adopted, review using results of program analysis service by means of the software analysis & measurement tools is required. If viewpoint 2 is not adopted, more testing is needed. HQC activities enabled quantitative evaluation of software process improvement. We explain program analysis service in detail in the next section.

2.2 Program Analysis Service in HQC

In service activities, we analyze program source code by software static analysis & measurement tools and explain analyzed data. This method is one of our group's main services, and it is related to product quality. We think that software product improvement leads to software process improvement.

The way in which HQC provides support differs according to the status of the development department. HQC's program analysis service activities are composed of the next 3 stages (See Figure 4).

The first stage is education.

When HQC receives a request from a software development department, we seek to identify what would constitute an appropriate service for solving the problems of the development department. At this stage we hold seminars to explain the importance of introducing new tools or alternations to the work flow. We support the development department on a trial basis. Our group is the main source of service activities. Through the activities, the consciousness of the department in question is gradually raised. Based on our group's know-how and service, developers learn how to accomplish their task more efficiently. Moreover, we can also accumulate data and know-how, including know-how respecting the efficient utilization of tools.

Figure 5 shows the relation between development and HQC. Our activities are in parallel with the actual project. Information can be fed back in a timely manner.

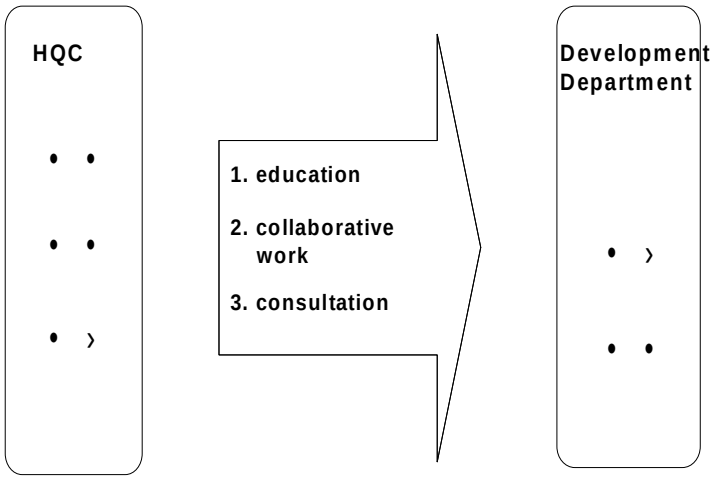


Figure 4 The stages of program analysis service activities

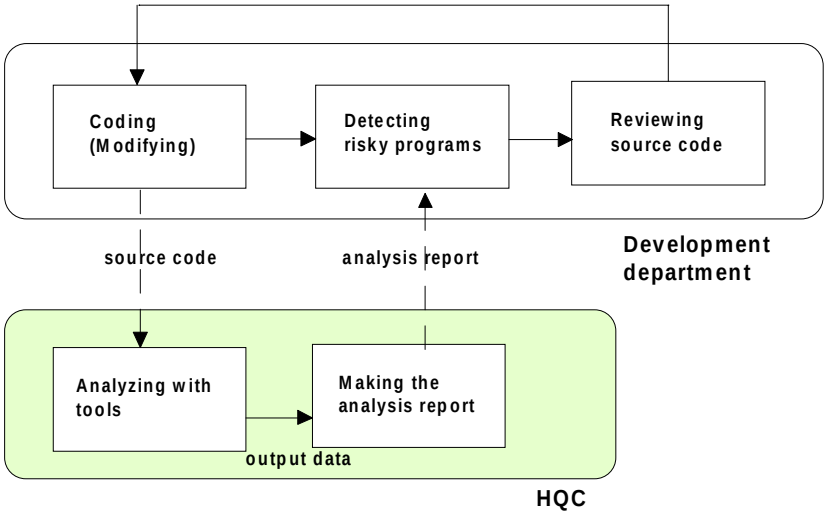


Figure 5 Information feedback between development department and HQC

We receive a software program source code, and analyze and measure it with a software static analysis tool. After that we select the functions or modules which have an especially complex structure. These are determined by acceptable quality value and the warnings about source code indicated by tools, because faults tend to be involved in these modules. We describe acceptable quality value in section 2.3.1 in detail. Acceptable quality value is determined based on accumulated know-how, and are used in review or testing. Selected functions

or modules are reported as an analysis report, and reviewed and modified if need.

The second stage is collaborative work.

We continue our activity in collaboration with the development department. The development department starts a self-improvement process in collaboration with our group, and so we can gradually transfer our accumulated know-how concerning new technologies to them, that is, teach how to use stored knowledge. It is important that know-how is accumulated in the development department. Information on faults or trouble begins to be fed back to our group. This information is useful for predicting faults or trouble in similar software developments. We can modify acceptable quality value and important warnings about source code according to each software development department.

The final stage is consultation.

At this stage the development department starts its own self-improvement activity. For example they measure and analyze their source code using customized acceptable quality values and important warnings. Our group monitors the software quality throughout the software life cycle.

We help them to establish their own improvement process. Our group's service is related to both process assessment and product evaluation. Our activity is widely reproduced and becomes widespread.

2.3 A Report on Program Analysis Service

We want to decrease the number of times re-work is executed. Therefore we need to make a report about complexly structured modules, because faults tend to be involved in these modules. For this purpose we use one of the typical metrics measurement tools, ESQUT *[1] -VB (Visual Basic) *[2]. GUI can be easily implemented by VB. We can measure VB source code by ESQUT-VB. Moreover we use one of the typical static analysis tools, QAC / QAC++ *[3]. A transaction logic portion is implemented by C / C++ language.

Characteristics of these tools are as follows:

- Measurement of metrics depending on GUI such as controls on a screen (ESQUT-VB)
- Measurement of metrics such as number of files, number of functions, and step size of a module (ESQUT-VB, QAC / QAC++)
- Output of warning messages respecting items that developers are likely to overlook (QAC / QAC++)

2.3.1 ESQUT-VB

We describe the functions and utilization of ESQUT-VB.

Implementation by VB is divided into three phases, that is, screen layout phase, screen event phase and transaction logic phase.

First, we implement a screen layout. Figure 6 shows a sample program implemented by VB. This phase is called “screen layout phase”.

Next, we implement a design of events on the screen which is mainly requested by user's operation. This phase is called "screen event phase".

Finally, we implement a detailed transaction logic for the screen event, common functions and so on. This phase is called “transaction logic phase”. It is similar to a programming language such as C or Cobol.

---

[1] Evaluation of Software Quality from User's viewpoinT. ESQUT is produced by Toshiba Corporation.

[2] Visual Basic is a registered trademark of Microsoft Corporation.

[3] A Quality Assurance Toolset for C / C++. QAC and QAC++ are registered trademarks of Programming Research Ltd.

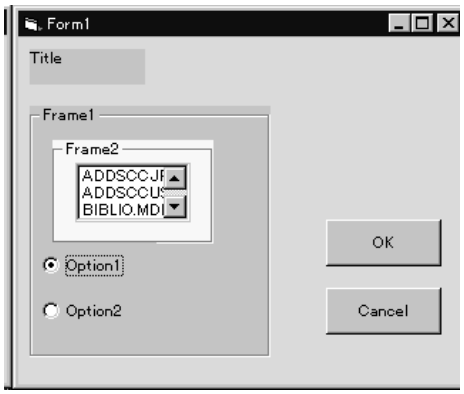


Figure 6 Sample program implemented by VB

The three phases are repeated, and a program is gradually implemented. The size of the program gradually increases, and so we must check the quality of source code in a timely manner. In each phase we can obtain an executable program. Thus we can measure source code in each phase by using ESQUT-VB.

Figure 7 shows a bar chart concerning the number of screen events on one screen.

A horizontal axis indicates the name of a screen. A measured system is composed of 2 sub-system (A,B). A sub-system "A" has 7 screens (left side in the figure) and a sub-system "B" has 10 screens (right side in the figure). A vertical axis indicates number of screen event. From this figure we can detect complexly structured screens (B1,B2,B3) which tend to involve faults.

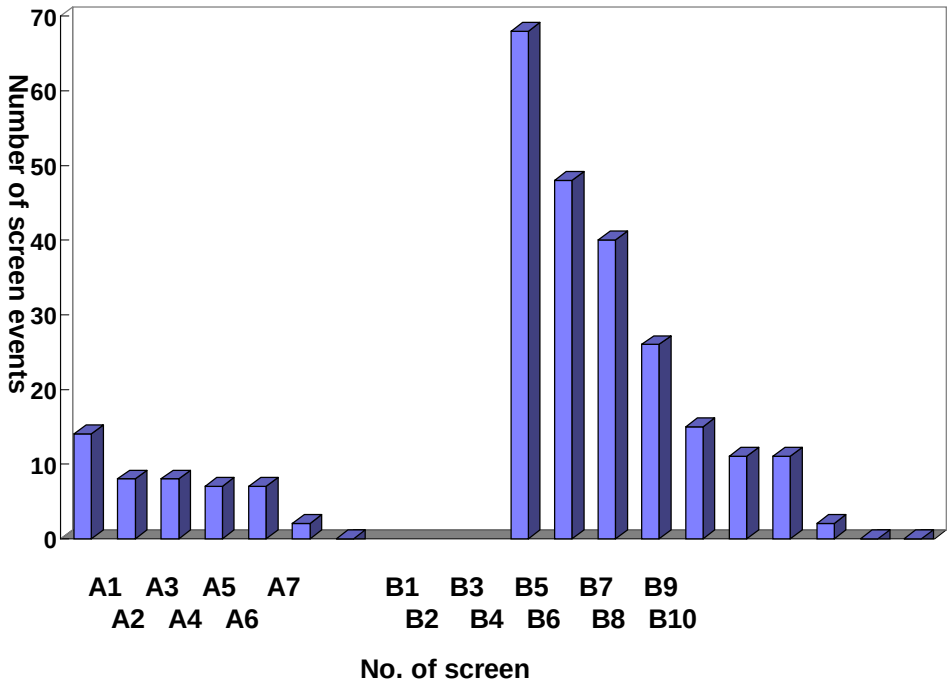


Figure 7 A sample output from ESQUT-VB (1)

This result can be obtained when we implement a screen layout and a possible event on the screen. It is not necessary to implement the transaction logic portion. Thus we can use this result in the midst of the development.

Figure 8 shows a scatter figure concerning module size (step size) and module complexity (number of conditions and loops). This figure can be obtained in transaction logic phase.

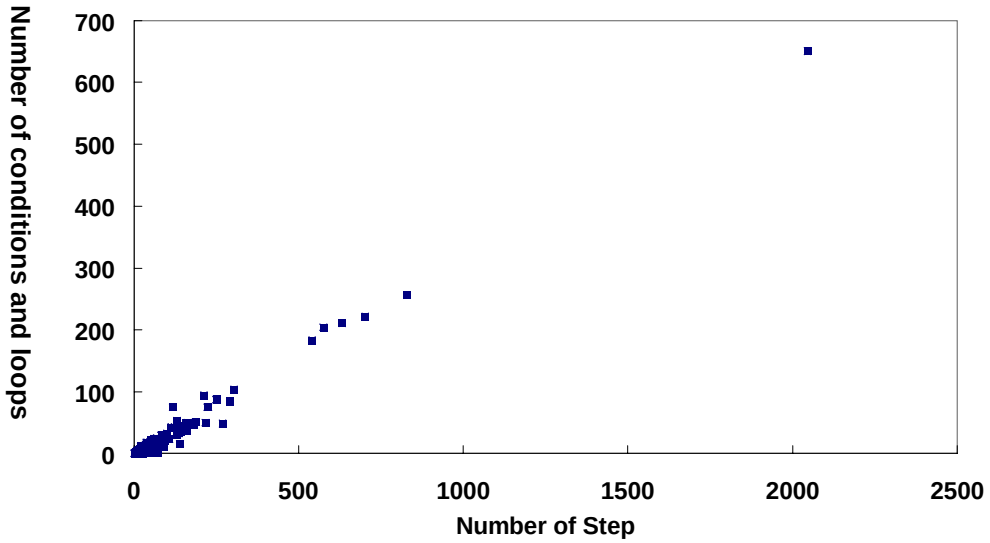


Figure 8 A sample output from ESQUT-VB (2)

A point which is plotted at upper-right side or upper side is a complexly structured module. These also involve faults.

We can use know-how in order to determine whether the program is complex or not. One of the criteria about acceptable quality value in ESQUT-VB is shown in Table 1. Acceptable quality value is acceptable measurement value of metrics.

Table 1 One of the criteria about acceptable quality value in ESQUT-VB

	Metrics	acceptable quality value
screen	number of event	30
module	size (step)	50
	nest (condition, loop)	5
	goto statement	3
	on error statement	0

2.3.2 QAC

Next, we describe QAC in detail. One of the most important functions in QAC is to indicate warnings about source code. These warning messages are related to the implementation of source code. Important QAC warnings are shown in Table 2.

Table 2 QAC warning messages

classification	warnings
type of variables	conversion of variable to another type
	floating
condition statement	unreachable statement
	lack of 'default' in switch statement
	lack of 'else' in if statement
maintainability	magic number (not defined as a constant value)
	goto statement
	use of braces even if the body has only one statement
others	strange or risky usage concerning pointers
	initialization of variables
	mistake between substitution "=" and comparison "=="

Figure 9 shows an example of a QAC warning.

In this source code, this condition always becomes true because a substitution statement "=" is used in if statement, so a comparison statement "==" must be used.

```
if (err == 0) {
  if (M_ID = 0xFF) {
        ^
-This control expression is an assignment
   test the result against zero
```

Figure 9 An example of a QAC warning (1)

Figure 10 shows another example of a QAC warning.

In this source code, an unsigned long variable is compared with negative constant. This comparison is degenerate.

```
#define MIN -100
 ...
func()
{
    unsigned long result;

    result = 100;
    if (result < MIN) {
          ^
    An unsigned value is never negative
    - this test is degenerate.
```

Figure 10 An example of a QAC warning (2)

2.3.3 Utilization of Results from Tools and Know-how

Tools are used even before code review or unit testing, because we want to construct a framework for feedback in the software process.

It is difficult for software development departments to determine acceptable quality value (ESQUT-VB). Therefore we present accumulated know-how in HQC on a trial basis. In the same way we use important warnings (QAC) derived from accumulated know-how.

Through several measurements and feedback of information, our group and the development department want to customize acceptable quality value according to the product. For example, we change the acceptable quality value according to the importance of sub-system.

On the other hand, QAC warnings about source code are accumulated as the number of measurement increases. On occasion we also obtain newly collected information from the development departments. This information is also classified according to the kind of product. In order to accumulate know-how, we need to collect measured data. This data includes information which is dependent on system or is applicable to other development departments. Classification is important in order to support other development departments

Data can be used by both project managers and developers. The managers determine which parts of the system are likely to involve faults or which parts of the system are falling behind schedule or what should be concentrated on in order to solve the indicated problems. The developers use results in testing or code review, and they detect a complex source code and modify it. They also determine the testing order from the results.

2.4 Effect of Program Analysis Service

We have continued our service activities. In actual software development, there are some effects (See Table 3):

Table 3 Effect of program analysis service in a case study

	size	language	tools	effect
(1)	10KLOC	VB	ESQUT-VB	Extracted VB modules (10% of all modules) contained about 40% of all the problems detected by review and testing.
(2)	100KLOC	C++ VB	QAC++	56% of faults were detected before testing.
(3)	1,200KLOC	C	QAC	A large system (1,200 KLOC) was analyzed within a day.

The first effect is as follows.

We received a sub-system from the development department. This sub-system is implemented by VB, and the step size is about 10KLOC. We measured it before the unit testing phase by using ESQUT-VB and acceptable quality value derived from know-how in HQC (See Table 1). This result was returned to the development department and they reviewed all the modules which are complexly structured from the result of ESQUT-VB. As a result extracted VB modules (10% of all modules) contained 40% of all the problems detected by review and testing. The developers commented that most modules which were detected by tools were modules which were difficult to maintain.

The second effect is as follows.

We received a sub-system from a development department. This sub-system is implemented by VB and C++, and the step size is about 100KLOC. We measured it before the unit testing phase by using QAC++. We select warnings based on the accumulated know-how (See Table 2). Managers commented that troubles in system testing or site operational testing were fewer than in the past.

The third effect is as follows.

It is difficult for managers or developers to overview a system, especially large size system. Using QAC, a large system was analyzed within a day.

3. CONCLUSION

Introduction of new tools or methods is difficult. In this paper, we explained our activities using a case study. It is important to support the development in parallel with the development tasks. We also have shown the effectiveness of service activities of this type, which provides software quality engineering tools and techniques and transfers know-how, that is, teaches how to use stored knowledge as follows.

1. We have been able to report the results of program analysis service and detect more faults before the testing phase.
2. HQC know-how on tool utilization and program analysis service improves the efficiency of review or testing. Armed with the know-how, some development departments have started self-improvement processes on their own.

HQC activities employ quantitative methods and are applicable to software development processes, and implement new software processes thereby endowing process control practice with a greater degree of quantitativeness, for example, a higher level of CMM.

ACKNOWLEDGEMENTS

The authors wish to thank the members of several development departments of Toshiba Corp. for preparing data for us.

REFERENCES

[1] A. Yamada, et al. “Software Process Assessment and Improvement”, Toshiba Review VOL.51. NO.2, 39--41, February 1996.

[2] H. Ogasawara, et al. “Experiences of Software Quality Management Using Metrics through the Life-Cycle”, 18th International Conference on Software Engineering, 81--88, May 1996.

[3] T. Tanaka, et al. “A proposal of software quality assurance method for VB (Visual Basic) program considering development process”, 17th Symposium on Software Reliability, May 1997.

[4] M. Aizawa, et al. “Support for Software Quality Improvement by Using Program Analysis and Measurement Tools”, 17th Symposium on Quality Control of Software Production, JUSE, 81--88, September 1997.

[5] T. Tanaka, et al. “Software Quality Analysis & Measurement Service Activity in the Company”, 20th International Conference on Software Engineering, 426--429, April 1998.

[6] V. Basili, et al. “Software Process Evolution at the SEL”, IEEE Software (Measurement Based Process Improvement), 58--66, July 1994.

[7] Humphrey, W. S. Managing for Innovation - Leading Technical People, Englewood Cliffs, NJ: Prentice-Hall, pp24, 1987.

[8] Thomas J. Haley. “Raytheon’s experience in software process improvement”, IEEE Software (Toward a Discipline), 33--41, November 1996.

[9] Mark C.Paulk, et al. Capability Maturity Model for Software, CMU/SEI-91-TR-24, SEI, August 1991.

PNSQC and 8ICSQ SUBMISSION COVER FORM

Title of Submission: Example of field quality improvement by raising the level of CMM (Capability Maturity Model)

Primary Contact Author Name: Mitsuru Ishio, Quality Assurance Department, NEC Communication Systems
Affiliation:
Email address: ishio@mt.ncos.nec.co.jp
Mailing address: MITA KOKUSAI BUILDING
1-4-28,MITA, MINATO-KU, TOKYO, 108, JAPAN
Work Phone: +81-3-5232-6337 Home : +81-44-945-2448 Fax : +81-3-5232-6390

[Abstract]

Since 1992, NEC Communication Systems (NCOS) has been utilizing a software process assessment system based on CMM to improve quality. In particular, in 1995, we were able to confirm that the in-house level had risen from CMM level 2 to level 3, and at the same time the field quality had risen by 50% every year for three years.

This paper verifies the effectiveness of CMM in terms of the improvement of our maturity level and its efficiency in improving field quality, and looks at some characteristic benefits based on the results of the assessment. The assessment questionnaire which was created by the sub-committee of software process maturity in NEC, was introduced to software process assessment.

Key Words/phrases : CMM, software process assessment, field quality improvement

[Author biography]

The author graduated from the mathematics department of Josai University in 1981 and joined NEC Communication Systems. in the same year. He worked in switching software department for 15 years, including two years as a manager in this field, and has been involved in quality assurance for the past three years. The author is currently the manager of the Quality Assurance Department at NEC Communication Systems, and his principal responsibility is implementing improvement activities. In this position he has also carried out 12 external assessments and 14 internal assessments.

Example of field quality improvement by raising the level of CMM (Capability Maturity Model)

Quality Assurance Department
NEC Communication Systems
Mitsuru Ishio

[1. Abstract]

Since 1992, NEC Communication Systems (NCOS) has been utilizing a software process assessment system based on CMM to improve quality. In particular, in 1995, we were able to confirm that the in-house level had risen from CMM level 2 to level 3, and at the same time the field quality had risen by 50% every year for three years.

This paper verifies the effectiveness of CMM in terms of the improvement of our maturity level and its efficiency in improving field quality, and looks at some characteristic benefits based on the results of the assessment. The assessment questionnaire which was created by the sub-committee of software process maturity in NEC, was introduced to software process assessment.

CMM: Capability Maturity Model

[2. History of process improvement]

NEC Communication Systems, a wholly-owned subsidiary of NEC, was established in 1980, and is involved in the design of switching systems. NEC Communication Systems designs a variety of switching and mobile systems, and develops software and hardware. It employs 1750 people. The systems we design include digital switching systems, ATM, LAN, and packet switching systems.

SWQC (Software Quality Control) activity created by NEC was introduced to our voluntary workgroup activities to improve software quality in the second year of our company's history, 1981. Since 1984, we have expanded TQC (Total Quality Control) activity organizationally as a top-down activity. In addition, we started using the Humphrey method of improving process maturity in 1992 to improve the constitution of the organization for the 21st century. The goal of this activity was the standardization of processes and horizontal expansion of strong points by diagnosing processes and narrowing down weak points company-wide. The in-house assessment method was introduced in order to strengthen and improve the organization from the beginning.

Our assessment is based on the guidelines of the Software Engineering Institute (SEI). We use the Software Assessment Guidebook and tools based on CMM methods created by NEC's Software Process Assessment (SPA) Subcommittee.

Fig. 1 shows the history of this activity.

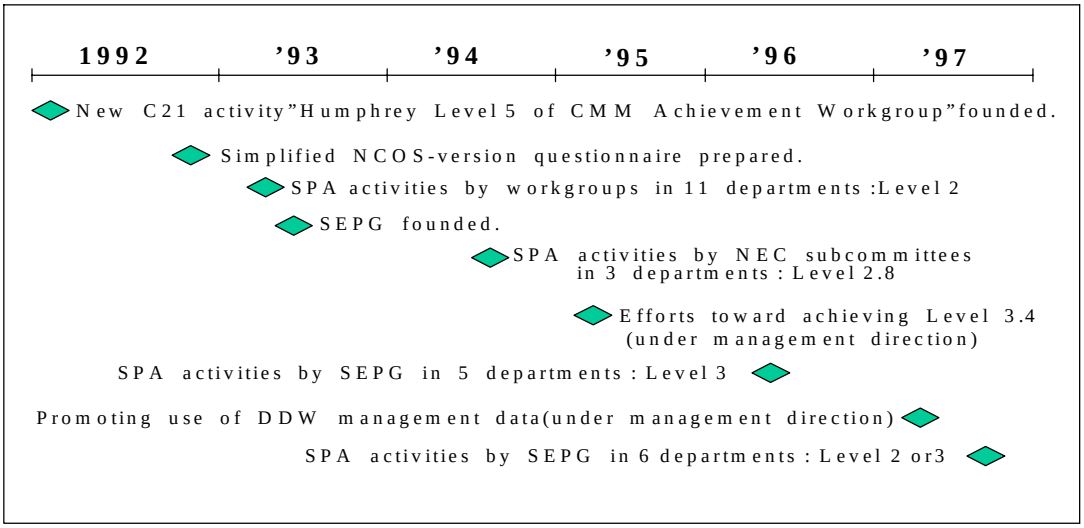


Fig. 1 History of process improvement

SEPG : Software Engineering Process Groups

[3. Effectiveness of process improvement]

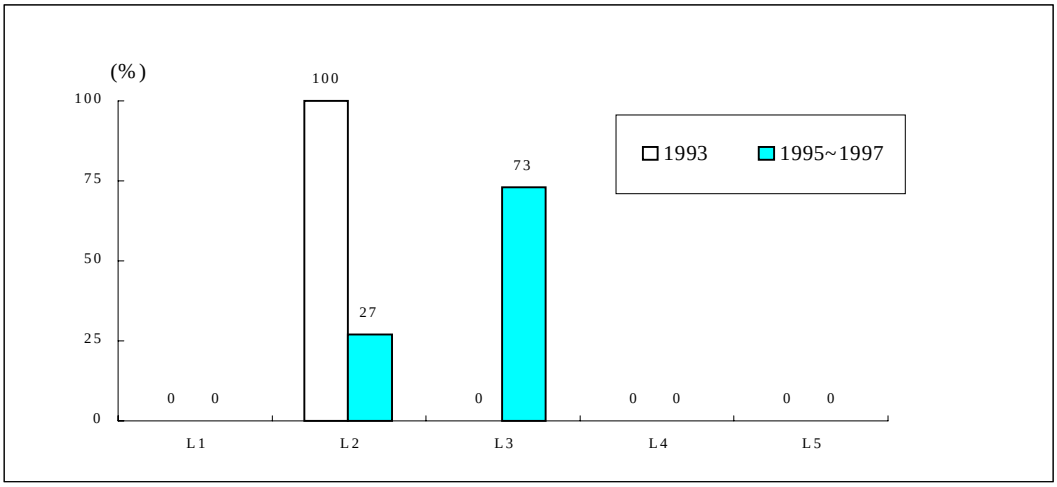
The results of in-house assessment of all software departments by assessors showed that the process maturity had improved from level 2 to level 3 from 1993 to 1997. The results for 1993 showed that all departments (11 departments: 11 projects) were level 2. Over the period from 1995 to 1997, we confirmed eight departments out of eleven to be level 3. The primary policy of process improvement in 1993 was "Establish and implement tracing and monitoring of projects."

We compared NCOS assessment results in 1993 with the assessment results from 1995 to 1997 using Fig. 2. We confirmed the improvement of process maturity from 100% level 2 in 1993 to 73% level 3 in 1995.

However, the most important aim of improvement activities at NCOS is not to raise the level of process maturity. The aim is to improve processes in order to develop products of a quality that is able to compete in the market. For this reason, we defined two ways of measuring the important factor of field quality, and continuously monitored the effectiveness of the process improvement using these two measurements.

During the process improvement activities in this period, we also confirmed the effectiveness of CMM for achieving improvement using two measurements to control field quality as well as the rise in the level of maturity in all divisions at NCOS.

Fig. 2 Changes in assessment results



1) Improvement of field quality

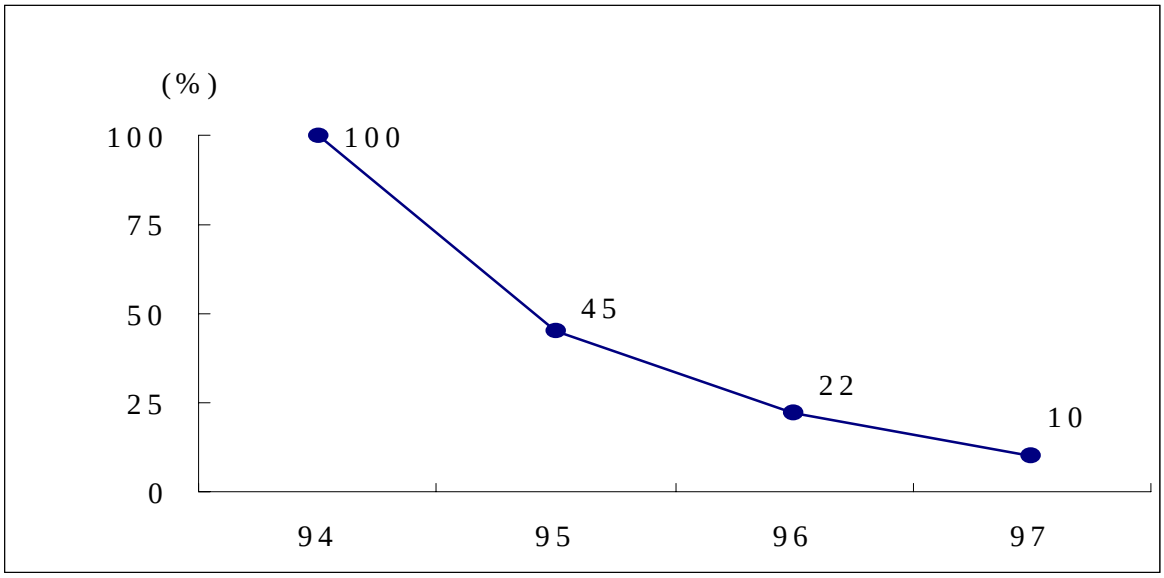
Field Quality is determined by monitoring the product for 6 months after start of service in order to be able to control the initial quality at NCOS. This measurement is applied to all software development divisions.

Fig. 3 shows the improvement of field quality of software with the rise in the level of CMM maturity.

After the level reached level 3, the quality at NCOS improved by 50% compared to the previous year for three year. Quality was measured by the number of defects in 1,000 lines of source code.

The factor which improved the field quality was that the SEPGs of all divisions tackled the weak point identified at level 2. The weak point was that the process transition determination that forms NCOS's quality system, was not defined. To solve this problem, we defined processes to create a system in which a development plan is decided upon and evaluated, and the decision to more on to the next step of the development process is based on the results of a review. In addition, we established a system to trace current status of development.

In this way, review were properly implemented in every process, and the quality of the reviews was improved. As a result of this, field quality also improved.



* The value for 1994 was normalized to 100.

Fig. 3 Quality 6 months after service-in

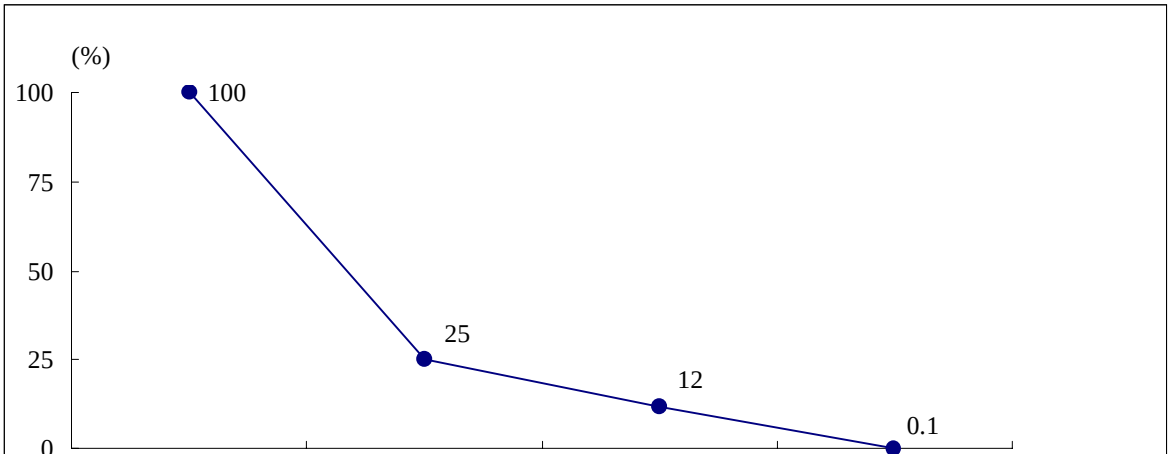
2) Failure ratio of file modification

Events after service-in are particularly important to telecommunication companies, who are our main customers. They focus on whether their new service will succeed or fail. The old files switching system files are replaced online with new files without shutting down the system. This function is called the "file up" function.

The outcome of important events after service-in need to be improved to enhance customer satisfaction. The failure rate of on-line replacement of files for switching systems was monitored as a macro quality measurement for software.

We analyzed the factors that degrade the quality, and found that the level of software quality, configuration management, and lack of pre-analysis affect the quality.

Fig. 4 shows the improvement of the failure rate of file modification along with the rise in CMM level. After the level reached level 3, quality improved by 50% compared to the previous year for three years. The quality was measured by the number of failures out of all events.



* The value for 1994 was normalized to 100.

Fig. 4 Failure ratio of file modification in advance

[4. Factors in rising level of CMM]

The factor in raising the level of process maturity to level 3 was the firm establishment and implementation of tracing and monitoring of projects. Specifically, manager reviews were implemented on the results of reviews, and a mechanism to determine the possibility of process transfer was firmly established.

The criteria for process transition is a difference of less than 20% between the planned process quality and actual results from reviews.

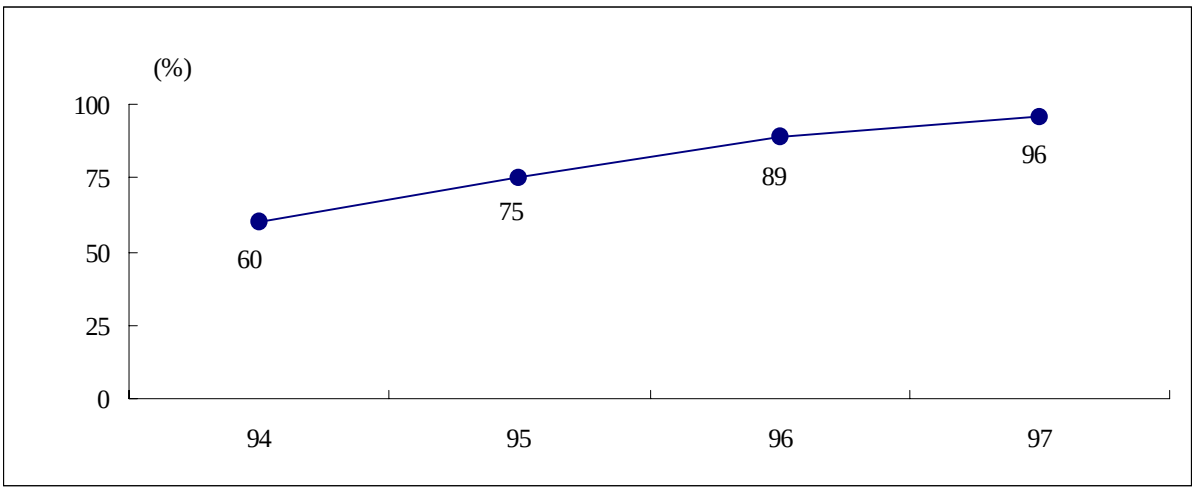
This mechanism was implemented in the quality accounting system in accordance with the concepts of Design of Designing Work (DDW).

The concepts of DDW are:

Before starting design work,
foresee what problems are associated with the development processes,
design processes which incorporated corrective steps in advance, and
check the individual processes in keeping with the progress of the work
to verify the effectiveness of the corrective steps.

As a result of the implementation of the DDW quality system in top-down activities as well as bottom-up activities such as Software Engineering Process Groups (SEPG), reviews within each process and audits by managers were thoroughly carried out.

We defined and monitored the rate of implementation of the DDW quality system in all projects as a management index. The implementation rate in 1994 was 60%, and currently the rate is constantly maintained



around 96%.(Fig. 5)

Fig. 5 Implementation ratio of DDW activity

[5. Characteristics of assessment implementation]

Initially, the problems of implementing process improvement included feelings of hurt and disappointment as people's weak points were pointed out and their level judged by someone outside the department, and the lack of experienced assessors. To solve these problems, we implemented assessments with the features shown below.

(1) We positioned assessments as voluntary improvement activities in the development departments. For this reason, we selected representative managers from each department as SEPG members. The SEPGs in each division implemented the assessments.

Staff had only an advisory role and supported the voluntary activities of the SEPG.

(2) However, it was necessary to standardize the assessment method. Therefore, the staff studied assessment methods used outside the company and disseminated this standardized know-how throughout company.

(3) To train SEPG assessors, we implemented self-assessments so that would fully understand CMM.

(4) Real assessments were primarily used for training assessors in assessment methods. Real assessments were implemented using an interactive role playing method.
(5) Then, the assessors were assigned one of three roles, prosecutor, attorney, and judge, and then observed real assessments. This allowed them to understand their own roles through the role playing.

The reasons for including the features shown above in the process assessments were the basic policy of SWQC group activities (QC Circles) and its organizational characteristics. The policy of SWQC group activities is expressed by the slogan "As part of company-wide activities for quality management, every employee implements self-development and interactive development utilizing the QC methods, and continuously manages and improves the work place."
Non-mandatory process improvement activities are natural in a company in which SWQC activities have been established.
Issues that we paid special attention to, in addition to the above, were as follows.

1) Assessments are designed to help bring out as much information as possible.
   This is because the people who best understand the problems at any particular stage are the people who work at that stage. Methods to help get accurate information from the people who are being assessed are described in the guidebook on assessment.
2) Self assessment is carried out before the external assessment.
   This is because self-assessment enables the person being assessed to diagnose their project using self-assessment activities. In addition, the person can understand their areas of weakness and this enables them to prepare the necessary evidence in advance. This also helps assessors to implement the external assessments more effectively.
   In other words, we can use our time more effectively to identify problems.
3) Training assessors in each division
   Assessors are needed in each division to spread self-assessment activities for processes as part of TQC activities.
   . The assessors are required to have the following four characteristics.
   * Familiar with the process maturity model.
   * A modest attitude in order to draw out information on problems from the people who are being assessed rather than simply deciding on a level.
   * Familiar with software processes.
   * Highly motivated to carry out process improvement.

We carry out assessments in the following way to improve these characteristics.

a) Distributing assessment tools
   An assessment questionnaire made using MS-Access can be downloaded from the homepage of the quality assurance department. (product of NEC-SPA sub-committee)
b) Establishing SEPG system
   The SEPG was founded under the direction of senior management.
   General managers were selected as leaders, and the managers of each line department were the members of the SEPGs.
   The roles of members are:
   * Implementation of internal assessments.
   * Promotion of improvement activities in the department.
   * Liaison with the quality assurance department.
c) Training assessors
   * Implementation of self-assessment.
   * Training in assessment methods (2 hours).
   * Experienced assessors explain actual examples of assessments.
   * Implementation of compatible assessment supported by the quality assurance department.
d) Implementation of self-assessment
   Each department selects a typical project, and the assistant manager carries out the self-assessment.
e) Implementation of assessment by members of SEPG
   i) Configuration of assessment
      Assessors: A leader, an interviewer, a recorder (all SEPG members)
      Clients: Assistant manager in charge of the project, two SEPG members in the department
   ii) Agenda of the interview
      * Opening greeting (cautions)

* All 85 items are covered within 6 hours.
* Confirmation of the results
* General evaluation

f) Creation of assessment report
Summarize strong points, weak points, points for improvement, and report to the SEPG leader.

g) Creation of improvement plan based on the assessment report
Plan the improvement in each department.
Trace the status of implementation of the plan.

h) Confirmation of effects
Confirm the results of process improvement using re-assessment then move on to the next step of the improvement cycle.

[6. Characteristics of assessment results]

Our assessments revealed some noteworthy results. These are shown below.

1) Comparing the results of self-assessment and the results of assessments by the SEPG revealed that assessments conducted by the SEPG after self-assessments are effective for determining the accuracy of assessment results and identifying problems.

a) On average the results of self-assessments by assistant managers differ by 27% from the SEPG cross assessments. The maximum difference was 44%. From these results we can see that we have to take an error of 50% into consideration in self-assessments using questionnaires. In other words, the field investigations and interviews are required to make process assessments effective. Another tendency we observed is that questions are very often misread and misunderstood.

b) When the scope of the assessment is the mechanism of overall organization, the results differ by 37%. It is difficult to determine the mechanism of the overall organization from the assessment of only one respondent (in this case, the leader). Therefore, process assessments require interviews with at least the managers, assistant managers, and subordinates.
* Interviewees can not be expected to know about all operations of their department.

c) The level decided upon changed in 67% of all cases.
We have to understand that there are many errors in level determination in self-assessments, and these happen more often than differences between results of self-assessments and SEPG cross-assessments.

d) In self-assessments, all departments underestimated themselves.
At NCOS, small group activity QC Circles have been introduced and voluntarily quality improvement activities have spread company-wide. We presume this is because self assessment is becoming stricter with enhanced maturity.

e) Quantitative process management was always overestimated.
* We believe an organization at level 3 cannot envision quantitative process management at level 4.

2) Opinions of assessors

By participating in the assessments, the members of SEPG readily realized the effectiveness of CMM and are committed to cooperating to ensure process improvement. These are some of the assessors comments: Representative assessors opinions are shown below.

a) By conducting cross assessments, I was able to learn about the systems in other departments, and this information is very useful. At the same time, I was able to recognize the weak points of my own department.
b) It was a good experience because I was able to understand the Humphrey processes.
c) I believe I was able to understand the key to CMM assessment by asking questions and through discussions.
d) Examining data analysis, training, management of cutting edge technology, and other areas was very useful because those are not included in ISO.
e) It was helpful because I was able to use the assessment for benchmarking.
f) It helped motivate my department to pursue improvement activities.

3) Drop in level due to process changes

We found that a drop from level 3 to level 2 occurred in projects where development processes changed significantly in order to adapt them to a new large-scale system development projects. From the results of these assessments, we found two problems. One is treaceability from upstream processes could not be guaranteed because many changes to requirements happened in upstream processes. The other problem is that it was difficult to determine normal quality within processes because no base line for quality data had been defined.

We presume that this is because an organization at level 3 does not have sufficient flexibility to respond to changes in technologies and processes. What we should learn from this experience is that we should not make a lot of major changes to a project when introducing a new process. In other words, new processes must be tested one by one in a pilot project . After testing, the results must be compared to a base line, and the effectiveness should be evaluated and modifications carried out before implementation.

[7. Future activities]

As mentioned before, we found that a drop in level occurred in new projects where development processes changed significantly. Improving the flexibility to respond to changes in technologies and processes is a task for SEPG and quality control. This will require quality improvement by the SEPG in terms of strategy. In addition, to reach level 4 it will be necessary to focus on the following four points and to establish a data-based process control system.

* Implementing management cycles on a daily basis
* Undertaking automatic and high-speed data processing
* Providing data analysis and estimation
* Improving the accuracy of input data

[8. Closing]

The effectiveness of CMM and process improvement were explained using examples and by describing the characteristics of NCOS in-house SEPG assessments.

To achieve a higher level, we will improve the techniques of the SEPGs and study the conditions at other organizations.

[References]
[1] Humphrey, W. S. (supervised and translated by Touno): Improvement of Software Process Maturity, Nikkagiren 1991
[2] Nishino and others, Sub-committee of NEC Process Maturity: Introduction to CMM Assessment Methods at NEC, The 52nd National Conference for Information Technology

"Using a Software Quality Model to Improve Supplier Performance"

Abstract: Software is becoming an increasingly important component of products received from suppliers. Contractors must have assurances that the software products will be delivered on time and with high quality. A key factor in achieving these objectives is a comprehensive Software Quality Program with proven results. This paper will chronicle the development and use of a Software Quality Model (SQM) for evaluating a supplier's software quality program and generating quality improvement ideas. The SQM has evolved over the last three years from primarily audit question checklists to a quality model that contains Quality Assurance best practice ideas extracted from key software/quality standards and experiences within Cellular Infrastructure Group, an organization that produces equipment and software to support a cellular telephone network.

Craig began his career with Motorola's Cellular Infrastructure Group (CIG) in Arlington Heights, Illinois in January 1994. As a Senior Staff Engineer in the Software Quality Engineering Department, his job duties include: software process auditing, quality team support, software supplier evaluations, teaching training courses, and program coordinator a the monthly organization quality meeting. He supported the Motorola team that helped CIG obtain ISO 9001 certification in June 1996. Craig also serves as a software assessor for Quality System Reviews conducted for Motorola business units and key suppliers.

Craig's twenty five year career in quality assurance and software engineering began as a quality engineer with Western Electric in Denver, Colorado. Since that time he has held positions as: Business System Analyst, Data Center Manager, Project Administrator, Independent Software Developer, and Lead Engineer.

Mr. Smith is a Certified Quality Engineer with the American Society for Quality (ASQ). He has a B. S. Degree in General Engineering from the University of Illinois and has completed graduate work at Illinois and California State University at San Bernardino. He is a member of both ASQ and the IEEE Computer Society.

R. Craig Smith
Motorola
Cellular Infrastructure Group
IL27, Rm 2315
1501 West Shure Drive
Arlington Heights, IL 60004
Tel.: (847) 632–4758, E–Mail: smithcr@cig.mot.com

1.0 Introduction

The Cellular Infrastructure Group (CIG) designs, develops, manufactures, markets, and services cellular systems and infrastructure equipment for wireless communication networks worldwide. Most CIG software organizations hold an SEI CMM Level 3 maturity.

A major role of the Software Quality Assurance (SQA) group within each CIG software organization is to provide a process assurance function. Audits are conducted to verify compliance to organizational procedures and industry standards such as the SEI Capability Maturity Model (CMM). [Paulk 93] The SQA group seeks to provide value added audits by recommending process improvements in addition to findings that reveal product defects and process non-compliances. [Smith, "Process Assurance," 95]

This process assurance philosophy is extended to subcontractors and suppliers of software. SQA seeks to fulfill its supplier quality program monitoring responsibility as defined in the CMM's Software Subcontract Management KPA and Motorola's internal quality system evaluation program known as the Quality System Review (QSR).

A review of supplier evaluations and QSR's conducted in the last three years within CIG revealed the following potential for improvement.

- The SEI CMM is a lengthy document and needed some condensation and focus in order to be used during a one or two day supplier review.
- The Motorola QSR process provided many strong evaluation approaches. However, because its scope includes hardware evaluation and the complete software development life cycle, isolating the software quality program for evaluation is difficult.
- Audit question checklists used to elicit interview responses were useful for compliance verification, but did not give a vision of a best–in–class quality program.
- The SEI CMM criteria lacked one important dimension of evaluation, results. Is the software quality program achieving results such as a reduction in customer found defects?

We decided that a supplemental audit reference aid should be developed to improve the effectiveness of supplier software quality program evaluations. This aid, known as the Software Quality Model (SQM), would be a concise set of quality evaluation criteria covering the key elements of a software development quality program. The approach used to develop the SQM:

- Include proven CIG quality program elements
- Integrate quality factors from the SEI CMM and ISO 9001
- Adopt the quality program evaluation structure from Motorola's Quality System Review Guidelines

2.0 Software Quality Model Development

2.1 Gather Potential Quality Model Factors

A review of successful aspects of the CIG quality program produced the following two elements as deserving prominence in the SQM:

- Rigorous inspection of software work products following the Michael Fagan formal inspection process which had proven successful in CIG organizations as the primary contributor to a reduction in customer found defects. [Fagan 86] During their first three years of use, Fagan inspections had been a major contributor to almost a ten fold reduction in the number of customer found defects.
- Measure the defects escaping each of the major software development processes, e.g. requirements and design. This measurement is called Phase Containment Effectiveness (PCE) within

Motorola [Daskalantonakis 92], but usually referred to in industry as Defect Removal Efficiency. The PCE is calculated as: defects found during process inspections, divided by: the total of in–process defects found plus all defects found after the in–process inspection.

For more than five years, product software development organizations at our location had used the SEI Capability Maturity Model as the basic guide for process maturity improvement. Several of the Key Process Areas (KPA's) contain essential information useful for developing the SQM:

- The SSM KPA, Software Subcontractor Management (SSM) activity ten and the verification section provided a good scope statement of the SQA role in regard to software supplier monitoring requirements.
- The Software Quality Assurance KPA provides a useful framework for the verification role of the quality function.
- The Training Program KPA emphasizes the need for training activities at three levels: individual, project, and organization.
- The Software Product Engineering KPA provides testing guidelines.
- The Peer Review KPA defines a rigorous inspection process.
- Both the Quantitative Process Management and Software Quality Management KPA's contributed good ideas to the development of an organization metrics program.
- The Defect Prevention KPA focuses on defect cause analysis, followed by a requirement for process and product improvement implementation actions.

The ISO 9001 standard also contributed to SQM development. Our Motorola site attained ISO 9001 certification in mid 1996. One of the areas of emphasis from ISO is the need for quality records, documents proving that certain process activities occurred. We decided that the SQM must emphasize quality records as a primary evidence that quality results were achieved. Quality records helps achieve the much needed visibility into the software development process.

2.2 Development of the SQM Categories and Model Structure

After careful analysis of the experiences from the CIG quality program and a review of applicable software engineering and quality standards, the following ten categories were selected as the basis for the SQM. **See the Appendix for the completed SQM.**

The Software Quality Model decomposes into ten quality program categories for purposes of scoring. Categories one and two, inspection and test, are *Detection Categories* which have the objective of finding defects in software work products such as code. Categories three through seven primarily are *Prevention Categories* with the major goal of preventing defects from occurring and improving product and process quality. Category eight covers measurement and metrics relating to product and process quality and category nine evaluates the overall management of the software quality program. Category ten is unique in that it looks at one or two other quality program elements that the organization may be using that are not covered in the first nine categories.

1) Inspection—formal reviews conducted to find defects in work products. Two of the most important inspection process evaluation criteria are: a) Use of a Phase Containment Effectiveness (PCE) type measure to evaluate defect containment efficiency at each development process, e.g., design, and b) Evidence of inspection process improvement, primarily as a result of elimination of causes of inspection process escaped defects. A low PCE value usually signals the need to improve phase defect detection methods such as inspection.

2) Testing—a function that exercises the system or components to verify that requirements are satisfied. Important evaluation criteria necessary for a high score include: a) An independent test organization that has early software development project involvement such as verifying the testability of requirements, b) Analyzing defects found after test that should be caught in the particular test process and making test improvements, and c) A test case data base under configuration management, with considerable test case reuse.

3) Software Quality Assurance—the function that provides management with a regular, independent assessment of product and process quality primarily via an internal audit program. Key factors necessary for a high score in this category include: a) Broad software development life cycle coverage especially the SEI CMM KPA's that have an SQA verification requirement, b) Evidence of timely corrective action of audit findings, and c) An SQA Plan for each project.

4) Quality Improvement—an established program of product and process improvement with defect prevention. Key factors pertaining to this category include: a) Metrics that demonstrate software work product defects reductions, and b) A defect prevention program with activities such as defect cause analysis.

5) Process Management—a software engineering process group/function that coordinates the development, change, and improvement of software development processes. Key evaluation factors include: a) Evidence of process institutionalization, b) Process controls that signal process problems so that corrective action can be taken to improve the process, and c) Process change effectiveness measurements.

6) Quality Training and Awareness—training courses and programs that develop the necessary skills and subject knowledge relating to product and process quality. Key evaluation factors include: a) Individual employee training plans and records that show regular training in such topics as inspection techniques, problem solving, and software development processes, b) Training effectiveness measures, and c) Training at three levels: organization, project, and individual.

7) Quality Standards—internally developed specifications and external standards that define software work product and process quality requirements. Key factors for evaluating this category include: a) Documents such as templates and style guides that define the format and content of all software work products, and b) Entrance and exit criteria for software development processes that function as quality gates to help control the quality of incoming/outgoing software work products

8) Quality Goals and Metrics—both organization goals and individual project goals for such quality factors as defect reduction and system availability improvement. Supporting metrics measure progress toward the goals for both the entire organization and individual software projects. Key evaluation factors: a) Positive trends in key quality metrics such as customer found defects, b) Quality information system that provides ready access to a wide variety of quality data, and c) Evidence of data use to improve quality.

9) Quality Management—management reviews and other activities demonstrating regular management direction and oversight for the software quality program. Important evaluation factors include: a) Project management minutes/documentation that show resolution of problem effecting product quality, b) A strong risk management program, and c) Use of metrics such as those mentioned in SQM Category Eight to make quality management decisions

10) Additional Quality Program Elements—a broad quality program that gives attention to additional quality program elements such as quality teams, software quality tools, software reliability, cost of quality, and software maintenance quality control. Either one or two additional elements will be selected for scoring based on knowledge of the supplier's quality program.

2.3 Software Quality Model Structure

Since 1982, Motorola has conducted Quality System Reviews (QSR's) of each business unit for purposes of evaluating the health of the organization as relates to Motorola's prime corporate goal of Total Customer Satisfaction. The SQM scoring and evaluation strategy derives from the QSR process. [Motorola 98]

As can be seen from the Appendix, the SQM is divided into three columns which provide word phrases to describe quality factors that help in category scoring a software quality program by giving one of five possible ratings from Poor (1) to Outstanding (5).

Perhaps the most critical feature adopted from the QSR process was a three dimensional strategy for organization quality program evaluation: Approach, Deployment, and Results.

- Approach: the appropriateness of the methods used
- Deployment: quality program instituted throughout the organization, not just a few projects
- Results: software quality results that demonstrate the effectiveness of the software quality program activities

In general, an SQM category is scored **Poor or Fair** if there is some evidence of a documented approach with limited deployment, but usually no effective results. An SQM category is scored **Adequate** if there are well–written procedures with thorough deployment and some positive results. An SQM category is scored **Good** if there is a sound approach and deployment with positive results that demonstrate the effectiveness of the category. The **Outstanding** category must meet the criteria of both the **Adequate** and the **Good** Scoring plus best–in–class results and a reputation so good that other organizations seek counsel and desire to bench mark.

Additional information on the Motorola QSR process particularly approaches for evaluating a software organization's software development capability and developing quality program assessment instruments can be found in the author's technical paper on this subject. [Smith, "Quality System Reviews" 97]

3.0 The SQM Usage Guide

3.1 Usage Guide Content

To aid in understanding and using the SQM, an SQM Usage Guide was produced. The SQM has purposely been confined to one page in order to provide a simple picture of a best–in–class quality program without too much detail. However, to supplement the SQM, the usage guide provides additional details to include:

- Definition and scope of each of the ten SQM categories
- Explanation of each of the five scoring areas from Poor to Outstanding
- Definitions of key terms such as Phase Containment Effectiveness
- References to standards and publications for more information about the ten SQM categories
- Discussion of evidence needed to verify the higher ratings of Good and Outstanding

3.2 Evaluating Software Quality Programs

The guide suggests evaluation of the ten quality program categories for:

- Documented processes
- Quality records demonstrating that activities have occurred
- Metrics that aid in process control and demonstrate quality program effectiveness
- Full process deployment across all software projects
- Documented process and product improvements attributable to the particular SQM category

The SQM Usage Guide includes recommended generic questions to ask during interviews so as to identify details about the supplier's quality program. Recommended questions:

- Briefly describe from start to finish the process and approach used for this SQM category
- What documented procedures describe the categories?
- What quality goals and metrics exist for this category?
- What quality records are kept that provide evidence of process execution?
- What quantitative data or metrics show process/product improvements and effectiveness?

- What evidence exists to show that the quality category operates consistently throughout the organization, e.g., all software projects?

4.0 Initial SQM Use

4.1 Supplier Audits

The SQM was used to plan two upcoming supplier software quality program audits. Next to each of the ten categories, a mark indicated that the topic was part of the last audit at the supplier location. Another mark indicated that the topic would be included in the current audit. The SQM also served as a guide during the development of the audit interview question sheet. Key SQM phrases helped define the scope and content of the audit questions. Thus, the SQM proved to be a useful supplier audit planning tool to ensure adequate audit coverage of quality program topics.

The SQM proved valuable during and after each of the supplier audits The model formed the basis for the opening presentation dealing with the fundamental components of a software quality program. Later, the SQM helped verify sufficient topic coverage during audit interviews. Also, the SQM contributed to identifying a number of quality improvement recommendations to include in the audit report.

4.2 Lessons Learned

As a result of three years of experience in software supplier evaluations, the following lessons learned have evolved.

Lesson One: *Need a diverse set of software standards to build a comprehensive software quality model.* Although the SEI CMM contains many good topics for an SQM, other standards such as quality evaluation factors from Motorola's QSR process, and ISO 9001 all made contributions to the model. No one perspective or standard contains the total proper formula for software process and product quality. [Smith, "Software Standards" 97]

Lesson Two: *Make the foundation of the SQM quality practices that have worked for you and that you understand. Plan to build the quality model from there.* The basis for the SQM were CIG experiences with a software development quality program and implemented software standards such as the SEI CMM. We acknowledge that the SQM will need to be enhanced as more is learned about such topics as:

- How a SEI CMM Level 4 or 5 quality program operates, e.g. characteristics of an effective defect prevention program
- New quality strategies required by object–oriented methodologies
- New customer areas of emphasis, e.g., system availability

Lesson Three: *As part of software supplier evaluations, the supplier must be given a vision of what a best–in–class software development quality program looks like.* The SQM model has sought to provide this vision by including best practices within our organization, including elements from CMM level 4 and 5 KPA's and drawing on the best practices from the top quality experts in software engineering. [Jones 97]

5.0 Conclusion

This paper has been an experience report on the development of a Software Quality Model, an evaluation tool designed primarily for use in evaluating software supplier quality programs. The tool has proven useful so far as an aid for planning and executing supplier quality program audits. Further experience will result in SQM enhancements.

The initial SQM use has resulted in the following benefits:

- A concise one page depiction of the essence of a software quality program that is not lost in the details of standards such as the SEI CMM
- A planning tool for selection of topics for supplier audits

- A high standard of excellence goal for software suppliers
- A contributor to supplier audit quality improvement recommendations

The SQM provides a useful approach for software development quality program effectiveness evaluation. An organization's software quality program is a vital component to success in software production. The SQM is a tool for use in strengthening this program so that superior quality software can be built.

6.0 Acknowledgements

Thanks to the following Motorola, Software Engineering professionals for their assistance in reviewing technical paper versions and commenting on the Software Quality Model: Norine MacGregor, Perry Ozols, and Art Bell.

REFERENCES

[Daskalantonakis 92] Daskalantonakis, Michael K. " A Practical View of Software Measurement and Implementation Experiences Within Motorola." *IEEE Transactions on Software Engineering, Feb. 1992, 998–1010.*

[Fagan 86] Fagan, M.E. " Advances in Software Inspections." *IEEE Transactions in Software Engineering, July 1986, 744–751.*

[Jones 97] Jones, Capers. Software Quality—Analysis and Guidelines for Success. *Int. ThompsonComputer Press, Boston, MA, 1997.*

[Motorola 98] Motorola. Motorola Corporate Quality System Review Guidelines, *Rev. 5, April 98, MUP–06–015. Available from: Motorola University Press, (847) 576–3142.*

[Paulk 93] Paulk, M.C. et al. Capability Maturity Model for Software, *Version 1.1, SEI, Carnegie–Mellon University Pittsburg, PA, 1993.*

[Smith, "Process Assurance" 95] Smith, Craig. "Process Assurance: Signposts on the Road to World Class Quality." *Proceedings of the Fifth International Conf. on Software Quality, Austin, TX, American Society for Quality, 1995, 385–96.*

[Smith, "Quality System Reviews" 97] Smith, Craig. "Quality System Reviews—A Software Perspective." *Quality Audit Conference Transactions, Los Angeles, CA, American Society for Quality, 1997, 383–94.*

[Smith, "Software Standards" 97] Smith, Craig. "Software Development Process Standards: Challenges for Process Assurance." *Proceedings of the Third IEEE Int. Conference on Software Engineering Standards, Walnut Creek, CA, IEEE Computer Society, 1997, 180–86.*

APPENDIX: Software Quality Model

Category	Poor(1) to Fair(2)	Adequate(3)	Good (4) to Outstanding(5)
1. Technical Reviews and Inspections	Few reviews. Perhaps, a basic review process in place. Quality records on reviews not always complete.	Documented inspections with quality standards occur for all software work products. Timely defect correction following inspections.	Reviews/inspection processes improved by analyzing recurring defects and inspection escapes and taking corrective action. PCE > 80%
2. Testing	Unit testing of code with at least integration or system testing. Records sparse.	Fully documented multiple testing levels with plans, test cases, and results reports. Test fault tracking to closure.	Sophisticated, config. managed test data base with significant test case reuse. Long term results show test effectiveness.
3. Software Quality Assurance	Few or some audits of software development activities with few audit findings corrective actions.	Thorough software dev. life cycle process SQA audit coverage with timely audit finding corrective action.	Timely action on SQA found problems and improvements. Measurable, positive SQA impact on quality.
4. Quality Improvement Program	Some attention to improving product and process quality.	Formal quality improvement program in place. Defect root cause analysis produces reasonable defect reductions.	Process/product improvement & defect prevention programs widespread. Metrics demonstrate major defect reductions.
5. Process Management	Some or most processes documented, but little evidence of process improvement	Dedicated org. to manage process work. Software development processes documented with good process impr.	Quantitative process mgmt using statistical process controls. Superior process improvement results.
6. Quality Training and Awareness	Employee training generally adhoc with little or no formal training plans developed.	Implemented org. & project training programs with individual plans adequately covering quality training topics.	Substantial evidence of training at individual, project, and org. levels: covers all aspects of quality training with results.
7. Quality Standards	Some evidence that Software Work Products (SWP) are produced following standards.	All SWP's have quality standards. Processes use entry and exit criteria (quality gates).	Evidence of continuous improvement to product and process standards
8. Quality Goals and Metrics	Some evidence of quality goals and a few project quality metrics produced.	Comprehensive org. quality goals set with supporting metrics in areas of defect reduction, cost of quality, etc.	Quality information system. Positive trends in product and process quality goals to include customer satisfaction.
9. Quality Management	Some senior management and project management attention to quality.	Established quality policies & an org. quality plan. Regular mgmt quality program review.	Timely resolution of mgmt detected quality issues. Sound risk mgmt program.
10. Additional Quality Program Elements	Attention to at least one other quality activity, e.g. quality teams	Several documented additional quality initiatives with some results.	Significant, positive results from at least two additional quality initiatives

Improved Software Quality by Adopting Control Charts

Anders Subotic andsu@ida.liu.se
Department of Computer and Information Science
Applied Software Engineering Laboratory
Linköping University, SE-581 83 Linköping, Sweden

Niclas Ohlsson niclas.ohlsson@gratistel.com
GratisTel International
Gjörwellsgatan 22,
SE-112 60 Stockholm, Sweden

Ola Blomkvist olabl@ikp.liu.se
Division of Quality Technology and Management
Linköping University, SE-581 83 Linköping, Sweden

Abstract

The paper discusses issues of control charting that are often forgotten or neglected in software literature. Examples, using defect data from a large-scale telecommunications project, illustrate some of the issues discussed. It is found that control charting can be more difficult than suggested elsewhere. The paper discusses issues related to control charting, e.g. models of quality characteristics and data collection for process control.

Author Biographies

Anders Subotic is a PhD candidate in the Applied Software Engineering Laboratory, ASELAB, at the Department of Computer and Information Science of Linköping University, Sweden. He received the MS degree in computer science from Linköping University, in 1996. His research interests include software quality assurance, process improvement and software quality engineering. His main area of research is software inspections, which he pursues in strong collaboration with industry.

Niclas Ohlsson received the PhD degree in computer science from Linköping University, Sweden, in 1998. He received both the MS and the Licentiate degree computer science from Linköping University. His research interests include quality assurance, process improvement, software metrics and software quality engineering. He was affiliated with the Applied Software Engineering Laboratory, ASELAB, at the Department of Computer and Information Science of Linköping University and is now Technical Director at GratisTel International.

Ola Blomkvist is a PhD candidate at the Division of Quality Technology and Management, Linköping University, Sweden. His main area of research is design of experiments and robust design methodology. In 1993 he received his MS degree in applied physics and electrical engineering and was employed at ABB, ASEA Brown Boveri, in Ludvika, Sweden. Since 1994 he is absent on leave from ABB to finish his PhD degree.

Improved Software Quality by Adopting Control Charts

Anders Subotic[1] andsu@ida.liu.se
Niclas Ohlsson[1] nicoh@ida.liu.se
Ola Blomkvist[2] olabl@ikp.liu.se

[1]Department of Computer and Information Science
Applied Software Engineering Laboratory

[2] Division of Quality Technology and Management

[1, 2]Linköping University
SE-581 83 Linköping, Sweden

Abstract

Society's high dependence on software demands increased attention to software quality. There is not only a need for improved quality, but also controlled quality. This necessity has lately led to a shift in focus from product to process. Reports from a number of organizations developing software suggest that development processes are now becoming repeatable, and as such enabling statistical control. One of the most popular tools of statistical process control is the control chart. In this paper, we discuss various issues of control charting, often forgotten in software engineering literature. We discuss applicability of control charts to software defect data, as well as assumptions, problems and meaning. Empirical data from a large-scale industrial development project is used in examples that domesticate control chart techniques and illustrate important issues. The authors observe a number of problems with earlier applied techniques. The implication is that the application of control charts often is not as simple and straightforward as earlier work suggests. Successful use of control charts demands insight from the user in both the underlying statistics and the application domain.

Keywords: SPC, statistical process control, control charts, software processes, defect analysis

1. Introduction

Walter Shewhart introduced statistical process control (SPC) and control charts already in the 1920's. Since then, control charts have been successfully applied within manufacturing industry. In a control chart, the variation of a product, process or service is monitored in a diagram with control limits, derived from past projects, which facilitate detection of unusual events. Control charts support the monitoring, detection and analysis of process variables related to quality and process control. Even if these activities are not successful, control charts play an important role in increasing the awareness of processes, quality and variation.

Control charts play an important role in evaluating different processes, techniques and tools, and in gaining a better understanding of these. Even though many have proposed that statistical process control can and should be used for software development there are, unfortunately, few well-documented applications of this in the literature. Furthermore, it is not very clear which methods are the most appropriate. For example, He et al. (1996) showed how C_p and C_{pk} can be used to monitor defect rates. Kirkham and Roberts (1996) rejected *c*-charts in favor of *u*-charts for plotting defect data. Kitchenham and Linkman (1990) pointed out the problem with non-Gaussian distributed data and argued for the use of boxplots and scatter plots. However, Pyzdek (1992) gave examples of data transformations that solve this type of problem. Poulin and Brown (1994) used different process diagrams for different defect injection rates to monitor the defect rate appearing in different phases. Wadsworth et al. (1986) described a number of classic and new applications of control charts that could be adopted to software engineering. However, e.g. Porter and Caulcutt (1992) suspected that failures of many SPC initiatives is due to an "over simplistic model of process variability", attributed to "standard texts on SPC [which] offer simple procedures … [that] do not produce useful charts in all situations."

Since 1993, the telecommunications company Ericsson have been committed to a process improvement initiative aimed at continuous reduction of maintenance costs and increase in product quality. An essential part of this is to bring software processes into statistical control. Thus, the potential benefits from using control charts were identified early. However, it was not clear which of the techniques proposed in literature were most appropriate. Therefore, a better understanding of control charts and their underlying assumptions was sought. To gain better understanding of earlier work, some analyses were replicated. In addition, data from a large-scale industrial project, developing telecommunications software, was used in evaluating various control charting techniques. The data was gathered from inspection, testing, and operation. Defects disclosed were classified by phase of injection, phase of detection, severity, technical category, and cause of error.

In conclusion, there is a need for a better understanding of what techniques can be used for quality control in software engineering. Therefore there is a need for empirical studies that validate earlier reported results and studies exploring the applicability of various statistical techniques. The focus of this paper is to discuss often forgotten or neglected issues in the application of control charts. Specifically, the assumptions, meaning and problems of control chart techniques are discussed. Many issues are illustrated in examples, using empirical data from a large-scale industrial development project. Central issues include assumptions of data distribution and models of process and product quality. Our experience from this review suggests that application of conventional control charts techniques is not always as simple and straightforward as it may appear from reviewing software engineering literature. A number critical assumptions need to be fulfilled, e.g. assumptions about distributions, the validity of which often are not justified. The simplicity of the models is also questionable. Thus, more advanced models, such as Bayesian and regression models, that consider more relevant factors, need to be considered. We also discuss which software processes need to be stable.

This paper is organized as follows. Section 2 provides an overview of SPC and control charts. Section 3 gives a systematic evaluation and application of various control charts to our empirical data set. In section 4 the results are summarized and discussed.

2. Statistical Process Control

Statistical process control is an attitude toward variation. Every process is subject to variation in its output. The variation is either systematic or random. Systematic variation has assignable causes, which can be determined, and eliminated. The variation that is left, after removing the systematic sources of variation, is the random, or chance, variation. A process that has no systematic variation is considered to be in statistical control, or stable. In the words of Shewhart (1931), "… a phenomenon will be said to be controlled when, through the use of past experience, we can predict, at least within limits, how the phenomenon may be expected to vary in the future. Here it is understood that prediction within limits means that we can state, at least approximately, the probability that the observed phenomenon will fall within the given limits." Observations of a stable process contain only random noise. Unusual observations, with low occurrence probability, indicate that the process has not operated as normal. First, this enables early assurance that defective products do not exit the process. Second, it is worthwhile to look for assignable causes since identification and removal prevents reoccurrence. Thus it is possible to move from acceptance towards prevention.

Statistical process control is aimed at finding the causes of variation and eliminating them. SPC has a primary set of tools, the seven quality control tools (Bergman and Klefsjö 1986; Montgomery 1996):

- Data collection or check sheet
- Histogram or stem-and-leaf diagram
- Pareto chart
- Cause-and-effect, Ishikawa, or fishbone diagram
- Stratification or defect concentration diagram
- Scatter plots
- Control charts

The main focus of this paper is evaluation of various control charts. However, the other tools are powerful and should be used in conjunction with control charts. The next part of this section explains the meaning of the control

chart, and lays the ground last part of this section. In the last part, important issues of control charting are discussed. Some of these issues reoccur later in the paper, in examples and the concluding discussion.

2.1. Control Charts

A control chart is graphical presentation of data, in a time perspective. Control charts are on-line process control tools (Montgomery 1996), designed to quickly detect changes in process mean or dispersion (Bergman and Klefsjö 1994). Typically, some important quality characteristic is measured and plotted at certain times. The quality indicator can be a product property, a process parameter, or any combination of these. "Anything" that indicates process quality and fluctuation is a candidate. Often, many quality indicators are charted in order to control a process. Bergman and Klefsjö (1986) suggest that process parameters be charted as it gives an earlier warning than charting product properties.

According to Bergman and Klefsjö (1994), a control chart should:

- Facilitate fast detection of systematic change.
- Create only a few false alarms.
- Be easy to use.
- Be of help in locating the time and kind of change, in order to assist in the defect analysis.
- Vouch for the stability of the process.
- Strengthen the awareness of process, quality and variation.
- Provide information for evaluation of process variation.
- Generate information for process improvement.

Figure 1 shows a typical control chart. It has a horizontal central line (CL), which represents the mean value of some quality indicator; the quality indicator is a metric based on either central tendency or dispersion. Above and below are two parallel lines representing the upper and lower control limits (UCL and LCL). The control limits are calculated from earlier process data so that, if the process is in control, a datum very seldom plots outside of the control limits. Typically, the accepted risk of false alarm is a few per thousand. If the distribution of data is known, determining the control limits is an easy task. Unfortunately, this not always the case with software engineering data.

A point above the UCL or below the LCL is a signal of an assignable cause of variation, i.e. that the process is out of statistical control. The sensitivity of the control chart can be increased so as to signal when points inside the control limits exhibit certain patterns. This is accomplished by interpreting the control chart using a set of sensitizing rules, such as eight consecutive points above CL (Montgomery 1996). However, higher sensitivity is achieved at the cost of increased risk of false alarms, see e.g. (Wetherill and Brown 1991).

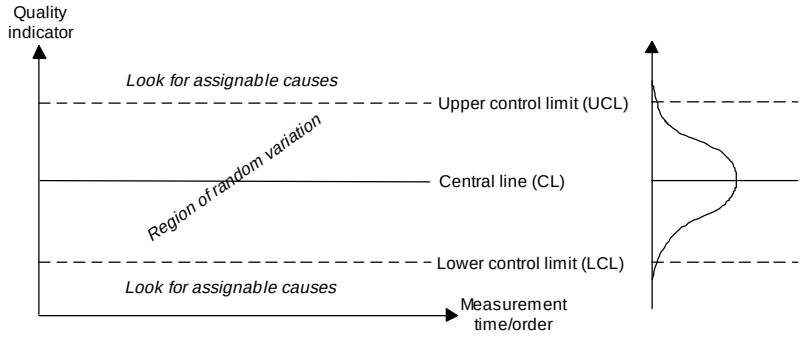


Figure 1. An example of a control chart and an example of a distribution it may represent.

Montgomery (1996) makes the analogy between errors in control chart with those in hypothesis testing. Type I errors occur when the control chart falsely signals an out of control situation. Type II occurs when the control chart fails to signal when the process is out of control. There is a trade-off situation between errors of types II and I. This trade-off is most commonly addressed in the following two ways (Bergman and Klefsjö 1986). First, by moving UCL and LCL apart type I error decreases while type II increases. Typically, the distance between UCL and LCL is set to six standard deviations (6σ). Second, charting values based on groups of observations, rather than single

observations, reduces the effect of random variation in the quality indicator. In addition, sensitizing rules can be used to lower error of type II at the expense of type I.

2.2. Choosing Control Charts

The purpose of the control chart is to separate observations with information from random noise. The control chart is a proven practical way to visualize what is considered as natural variation, the underlying distribution of the quality characteristic, as well as a baseline for classifying further observations. There should only be noise in the control chart if the process is stable. The more information we have on the nature of the distribution the sharper the tool gets. Thus, it is necessary to understand that the control chart is intended to mimic the underlying distribution of the quality characteristic, which is the main point of this section.

The performance of a control chart is often expressed in the operating characteristic curve, OC-curve, and the average run length, ARL. The OC-curve illustrates the probability that an observation will fall within the control limits as a function of the change in the underlying distribution and is therefore a measure of type II error. In this way the OC-curve illustrates the sensitivity of the control chart. However, the ARL is a more direct measure of the sensitivity since it is the average number of samples between a change in the underlying distribution and an alarm in the control chart. For calculations of ARL or OC-curve, the distribution has to be known. Even though both ARL and the OC-curve are important issues, they are not investigated in this paper due to the nature of the later examples (see section 3.2).

The choice and design of control chart depends on factors such as: the *objective* of the work, the *type* of the quality indicator, the *distribution* of the quality indicator, the *independence* of measurements, the *sample size* and its *variability*, the *number of samples*, the *sampling frequency*, and the *control limits*.

Various charts or chart configurations can be used for different purposes. There are different charts that can be used for detecting small process shifts or for large shifts. Standard control charts, such as $\overline{X}$ *-R*, only use information from the last sample to determine whether the process is in control or not. Cusum charts use information from earlier samples in order to be more sensitive to small shifts in the process, without increasing the risk of false alarms (Bergman and Klefsjö 1986). However, cusum charts are not good at detecting large shifts in the process, but combining them with a standard control chart (Montgomery 1996) can solve this. In this and other cases of several simultaneous charts there is always a trade-off between type I and II error and especially the multiple test effect must be accounted for. Montgomery also describes the EWMA chart, the performance of which is comparable to that of the cusum chart. Charts for detecting small shift are not further discussed.

The charted quality indicator is of either *variables* or *attribute* type. Pyzdek (1990) defines variables data as "data that are obtained from a continuous scale of measurement…that can take on any value on a given scale". However, Pyzdek (1992) gives a less strict definition where data can be considered as variables "if at least ten different values occur and repeated values make up no more than 20% of the data set." Burr and Owen (1996) define attribute as a feature that may or may not be present, typically counts such as software defects. According to Griffith (1996), attribute charts are less sensitive than variables charts. The reason is that attribute data contains less information than variables data. In fact, Burr and Owen (1996) go as far as to recommend that "defect data should always be accompanied by a variable[s] measurement…that describes why the [defect] value should be believed…"

The design and choice of control chart depends on the *distribution* of the data to be charted. The data distribution affects the control limits and their meaning, as seen in Figure 1. Standard variables control charts rest on the assumption that the charted characteristics are identically normal and independently distributed (Montgomery 1996). However, from Pyzdek (1995) we learn that normality is uncommon, especially in management and engineering processes. Montgomery (1996) suggests obtaining percentiles from the underlying distribution or transforming to an approximately normal distribution. Furthermore, some common sensitizing rules, or run tests, depend on data belonging to a normal distribution. However, more robust, non-parametric run tests exist (Pyzdek 1990). Returning to distributions, standard attribute control charts assume either binomial or Poisson distribution (Pyzdek 1990). In contrast with normal distribution, these can seldom be considered symmetrical, and there are no underlying mechanisms, as the central limit theorem, which remedy erroneous assumptions. Obviously, for attribute data, run tests have to be customized to account for the non-symmetrical properties.

According to Montgomery (1996), the second assumption behind standard control charts, the *independence* of observations is very important. However, neither e.g. Pyzdek (1990; 1992) nor Burr and Owen (1996) discuss autocorrelation. As argued earlier, independence is often considered to be guaranteed if the time between samples is sufficient. Montgomery states that standard control charts give too many false alarms even for "low levels of correlation". This is due to an underestimation of the total process variation and consequently to narrow control limits since subgroups do not contain all variance components to the extent that the process does. Montgomery continues, "the assumption of uncorrelated or independent observations is not even approximately satisfied in some manufacturing processes." Certain run tests can efficiently detect correlation among the observations but there are also more formal tests, see e.g. (Box et al. 1994; Draper and Smith 1981). However, often the visual appearance of the control chart is enough to reveal patterns, which look non-random. Patterns, like all types of alarms, require further inquiry to determine cause, as any alarm can be triggered by random variation.

Another important issue is the *sample size*, the base of the charted values. Roughly, three fixed sample sizes can be distinguished: 1, around 5 and above 10. The rationale for grouping observations can be both statistical and practical, related to normality, sensitivity, availability and cost. Sample size can also be variable, and charts exist also for these instances. The sample size affects the performance of the control chart, as e.g. the sensitivity of detecting small shifts increases with sample size. As argued before, the issue of knowing the distribution is crucial when the number of observations is very small. Specifically, the central limit theorem does not provide an underlying distribution in these cases. It should be kept in mind that the control charts must represent the underlying distribution in order to be reliable.

Further, the *number of samples* used to establish the control chart is vital. Traditionally, setting up a control chart requires about 20 samples to establish the central line and control limits. It should be noted that it is necessary to use as many observations as it takes to represent the underlying distribution in a fair manner. However, as Griffith (1996) notes, some environments have problems with "insufficient or untimely data for control limits." This has been termed small or short run production. Control charts have been developed to handle these challenges, adaptations of standard charts as well as new ones. Typically, focus is put on the process rather than the product. Alternatively, product families are considered rather than single products. The assumption is that the underlying processes are identical except for certain shifts. Anyhow, the number of samples needed to establish a control chart depends on the sample size. Pyzdek (1992) provides procedures and tables to be used down to a single observation, but the recommended number of samples larger, especially for small sample sizes. From a practical perspective, most of the short run charts are more complicated to use than standard ones.

Sampling frequency, the intensity with which data is collected, is related to sample size. Often, the process and practical considerations determine the frequency. Higher sampling frequency often means increased effort. Still, according to Burr and Owen (1996), the sample size shall preferably be "sufficiently large to reflect [e.g.] the error rate." If the level of the sampled indicator is very low, sampling may be inadequate and every item then has to be inspected. Sampling can also be hazardous, e.g. when defects are unevenly distributed in products, a point made by Burr and Owen (1996). Consequently, not all people in the software industry do trust sampling. For example, often large documents are inspected in their entirety rather than sampled pages. One reason being that quality assurance techniques like inspection and testing often have not been used for quality control but for defect detection and correction at late stages of software processes, i.e. acceptance testing. Further, every item is usually inspected or tested.

The choice of control limits should reflect the user's attitudes toward type I & II error and the sensitivity that is demanded of the control chart. The positions of the control limits are straightforward to determine if the underlying distribution is known. However, it is important to recognize that if the underlying distribution is skewed and the risk for false alarms is to be the same in both directions, the control limits should be asymmetrically distributed around the central line. A compromise and an approximate solution that often appears in literature is to set the limits to $\overline{X} \pm k\hat{s}$. A rationale for this is Chebyshev's inequality, $P\{|\overline{X} - \mu| \geq k\sigma\} \leq 1/k^2$, which gives the maximum probability of a false alarm when *only* μ and σ are *known*, but not the distribution. For the usual 3σ-limits the maximum chance of false alarm is 11.1%, if μ and σ are *known*, which can be compared with 0.3% when the

distribution is known to be normal. Furthermore, for skewed distributions with symmetrical control limits, the chance of false alarm is not the same on both sides.

3. Case Study

The objective of this section is to illustrate some of the issues raised in the previous section by applying control charts to empirical data. The section begins with a short overview of our data, which is used in later examples, specifically numbers two to five. Example one is an alternative analysis of an example in (He et al. 1996).

3.1. Data collected

The data presented in this section is based on one sub-system within a major release of a legacy system. The sub-project is responsible for 15 modules, which were all included in this study. Totally, 311 defects were reported and classified as major, and thus included in the study. Of these 311 major defects, 179 were found during development, 126 during testing, and 6 in operation (see Table 1). The modules ranged in size from approximately 1,000 to 3,500 LOC. One module was new and the other enhancements of existing modules. The modification degree, i.e. the percentage of lines of code that has been changed, ranged from 1 to 67 percent. The defect data was collected from the phases: function specification, function design, module design, coding, function testing, system testing and six months of operation.

Development phase	Detection method	Defects discovered
Pre-design	Inspection	53
Design	Inspection	27
Code	Inspection	99
Test	Testing	126
Operation	Operation	6
	Total	**311**

Table 1. The number of defects detected per phase and method.

3.2. Application of Control Charts

This section contains examples where the issues discussed earlier are illustrated on a practical level. Due to nature of the data and the format of the paper, no investigations into assignable causes are performed.

Example one

Two of the most common ways of charting variables data is the $\overline{X}$-R chart and the X-moving-R chart. The $\overline{X}$-R chart is based on sample *means* of observations while the X-moving-R chart is based on *individual* observations. Both charts assume normally distributed data, but the $\overline{X}$-R chart can tolerate moderate violations while X-moving-R is more sensitive (Montgomery 1996). This example contains an alternative analysis of an example in (He et al. 1996), based on project progress data from (Grady 1992). He and colleagues made use of Shewhart-like individuals charts for variables data. The individuals chart is not accompanied by an analysis of process dispersion. The individuals chart had control limits 0.69 and 0.32, based on the standard deviation. Their analysis suggested a stable process. Using this example, we begin with a normality test using the normal probability plot, see Figure 2. As can be seen in Figure 2, the observations adhere well to the line, and suggests control limits 0.70 and 0.30, which is in agreement with He et al. In Figure 3, a corresponding standard X-moving-R chart (Pyzdek 1990) is shown. The moving range chart indicates stability. However, in the individuals chart, with control limits 0.59 and 0.42, several points fall outside the control limits, starting week 24. The difference between the control limits lies in the estimation of process dispersion, a most important issue in designing individuals charts (Roes et al. 1993). While He et al. used the standard deviation of the total sample, the average range was used here, as suggested in (Pyzdek 1990). The control chart in Figure 3 is of help in trying to understand the difference. Visual inspection of the individuals chart gives several different out-of-control signals. First, the most obvious being 11 consecutive points below the central line. Second, there is a positive trend, with six steadily increasing points, at the end of the chart. In addition, much of the chart exhibits a conspicuous pattern, with alternating means. These patterns also reveal themselves in the autocorrelation coefficient for the data, lag-one being 0.68. This high value suggests that the observations are not

independent, and the last observation is a good predictor of the next. In turn, this explains the difference in the estimation of process dispersion since the sample standard deviation measures long run variation and moving range measures short run variation.

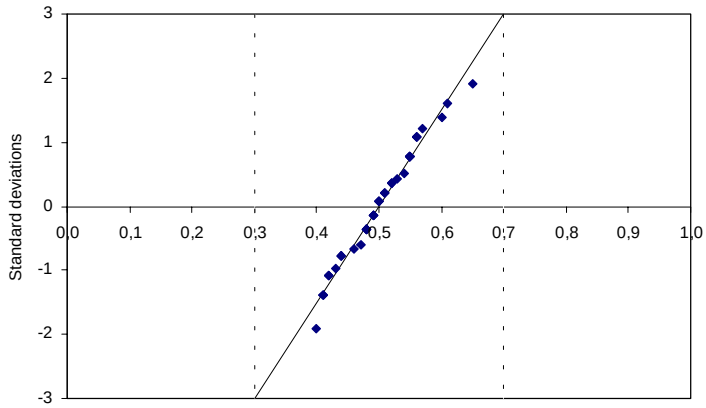


Figure 2. Normal probability plot for project progress.

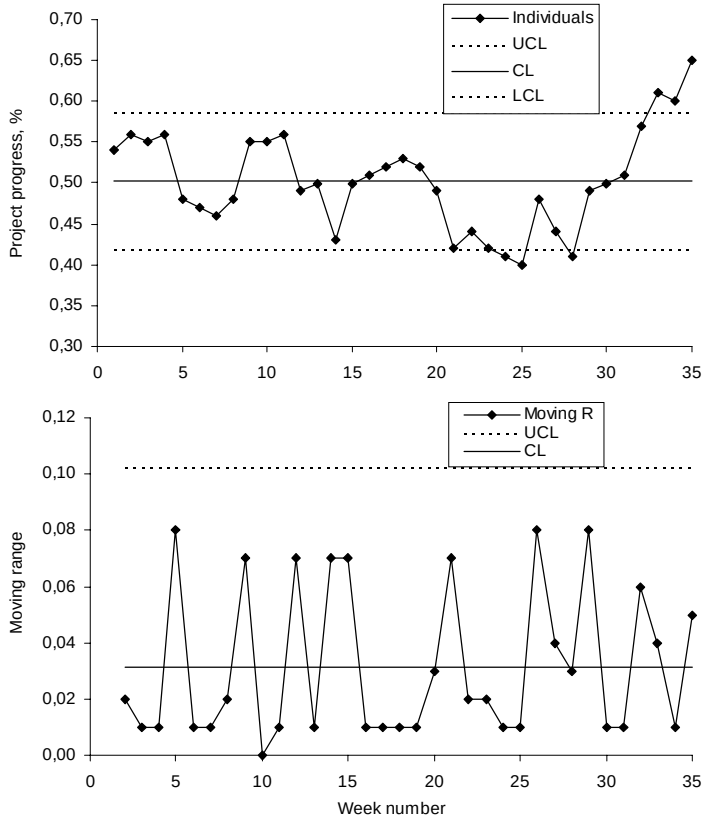


Figure 3. *X*-moving-*R* chart for project progress.

Example two

In this example, the total number of defects, i.e. from all phases, is used. Defect count is attribute data, but if it is normalized against module modification size, the data can be considered to be of type variables. However, the available data does not show a resemblance to the normal distribution. A log transformation gives data that is approximately normal, as seen in Figure 4. Scatter plots reveal no relations between the transformed data and module size, module modification size or module modification grade (not shown). The observations are assumed to be

independent as the modules were developed largely in parallel, by different teams. As there are only 15 data points, the sample size one is chosen, resulting in an individuals chart. The rationale for not grouping observations is to preserve information (Wheeler 1991). Furthermore, it is necessary to use a short run version of the *X*-moving-*R* chart, e.g. (Pyzdek 1992).

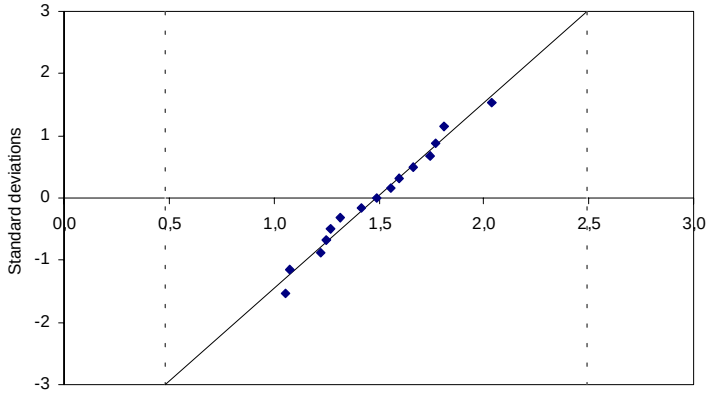


Figure 4. Normal probability plot of log transformed defect rate.

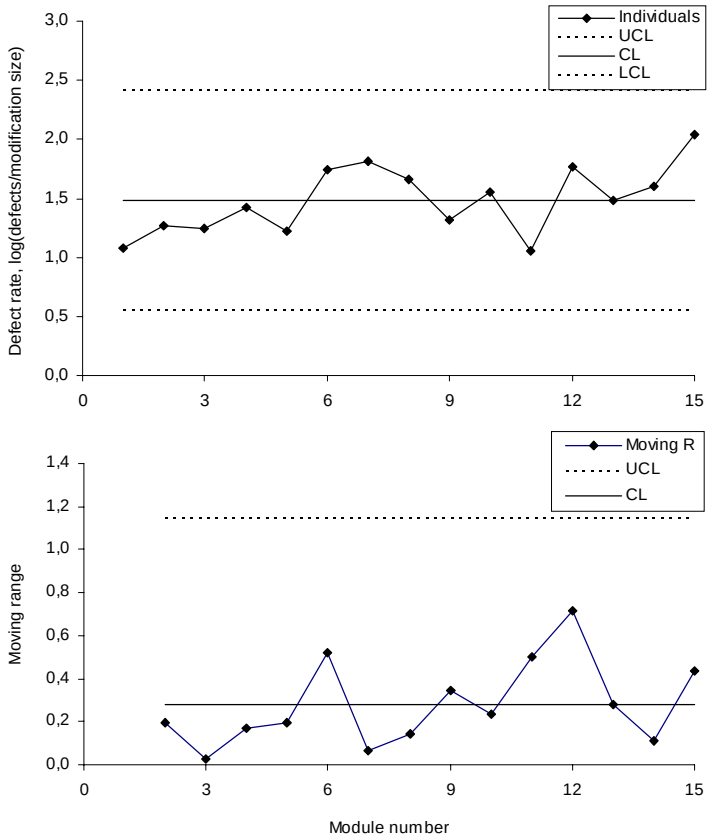


Figure 5. Exact method *X*-moving-*R* chart for log transformed total defect rate.

The exact short run *X*-moving-*R* approach produces the control limits 2.36 and 0.61, for the individuals. These limits are slightly wider than those from straightforward calculations of mean and standard deviation, which are 2.35 and 0.62. The limits extracted from the line in the normal probability plot in Figure 4 are wider, 2.49 and 0.48. The difference between the estimates does not seem significant. The first approach is the hardest, and the resulting charts are shown in Figure 5. Both charts indicate that the process is stable. However, the value of moving-R charts has been contested, e.g. by Roes et al. (1993), who argue that “displaying a moving range chart has no real value and is,

therefore, ill-advised." The main arguments being that "moving ranges are correlated", individual observations contain the information, and a moving range can signal out-of-control for a pair of individual values that are inside their control limits. As for the individuals chart in this example, interpretation is not trivial, as the meaning of the transformation is not quite clear at this stage. A common rationale for the log-transformation is its stabilizing effect on variance when uncertainty increases with the magnitude of observations.

Example three

In the previous two examples the control charts have been established using "quite a lot" of data. However it might be desirable to start earlier, using only a small number of observations. The problem with this approach is that the distribution of the quality characteristic is quite uncertain and this must naturally show in the control chart. In the beginning, as knowledge is small, control limits are wide. As more data is accumulated, the estimates of process mean and dispersion become more accurate as uncertainty decreases, which is reflected in the chart. Obviously, the procedure requires iterative refinement of the control chart. To illustrate this and other points, a new variables short run chart for individuals is utilized. A family of variables charts for short runs is the code value charts or difference charts, e.g. see (Pyzdek 1992) and (Wheeler 1991). These charts are obtained by plotting the difference between measurements and nominal or target values. However, as the target value for defect rate is zero, this results in charts that look largely the same as the previous example. If we go one step further by subtracting the process mean from each observation and divide by process dispersion, the result is a stabilized control chart for individuals.

The chart is based on the transformations: $(\overline{X} - \overline{\overline{X}}) / \overline{R}$ and $R / \overline{R}$. The resulting statistics have no unit and correspond to the number of average ranges. As we are concerned with individuals charts, we remove one bar from each *X*. In the earlier examples, raw data was plotted. For stabilized charts, the data is transformed to suit the chart. Stabilized charts illustrate the deviation from process mean whereas standard charts plot the actual values. The exact method (Pyzdek 1992) used in this example requires constant subgroup size. That is, the documents, or changes, used to form the individual values should be of roughly the same size so that the individual values have the same variance. Even though this is not true, we proceed as normal and use the log-transformed values from example one. The resulting stabilized *X*-moving-*R* is shown in Figure 6.

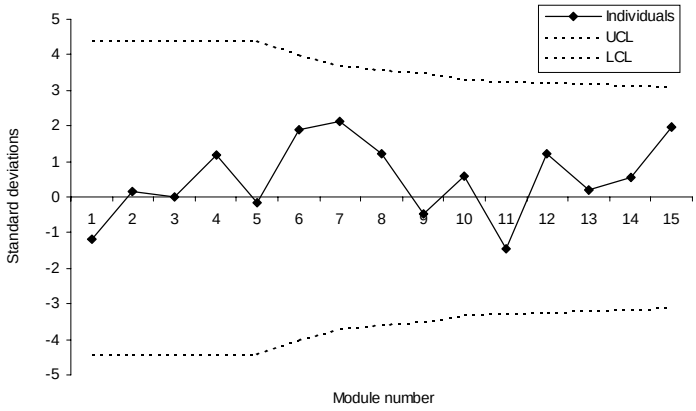


Figure 6. Stabilized *X*-moving-*R* chart for log-transformed total defect rate.

The first five observations were used for calculating the start-up control limits, which are constant for these points. The limits are recalculated for each additional observation, which causes the control limits to vary. This kind of start-up problem also occurs every time an assignable cause is identified and actions are taken to remedy its effects, since the underlying distribution then changes.

Example four

The data analyzed in this paper is mainly related to defects. Defect data is of attribute type, a fact largely neglected so far. There are four standard attribute control charts, the *np*, *p*, *c* and *u* charts. Briefly, the *np*-chart is for charting the number of non-conforming units, *p* for the fraction of non-conforming, *c* for defects per unit, and *u* for average defects per unit (Pyzdek 1992). In this example, defect data from the test phase is used. The data includes the number

defects found in each module. The tested modules are of varying size. Only the *u*-chart and *c*-chart are applicable as there is no standard for non-conformance. The *c*-chart is based on the assumption that the number of defects is a random Poisson variable (Montgomery 1996). In order to validate the assumption, the empirical density function is compared with the Poisson probability density function, see Figure 7. One plausible explanation for the lack of resemblance is the varying module size. The *c*-chart is consequently rejected in favor of the *u*-chart. Since it is difficult to test if the assumptions of the *u*-chart are fulfilled, these are assumed to be fulfilled.

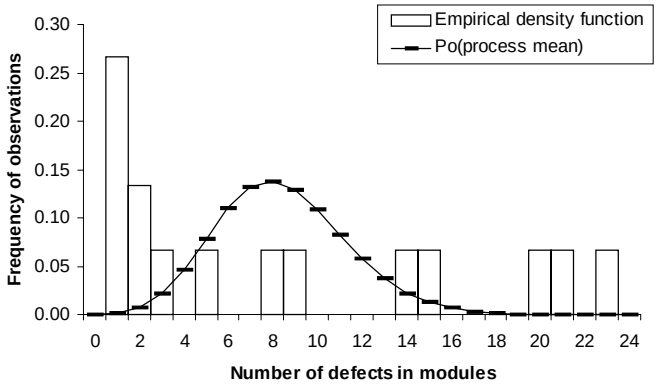


Figure 7. Empirical and Poisson probability density functions based on defects found in the test phase.

The use of *u*-charts for controlling software processes was suggested already in 1987 by Gardiner and Montgomery (1987). They suggested that each development phase should be monitored with defect rate *u*-charts. A later example is Kirkham and Roberts (1996), who analyzed inspection data using both standard and stabilized *u*-charts. Standard attribute charts require about 20 observations for establishing the control limits. The stabilized *u*-chart (Pyzdek 1992) is not affected by this problem, and in addition varying control limits are avoided. The stabilized versions of all four standard attribute charts are based on the same transformation:

$$Z = \frac{\text{sample statistic} - \text{process average}}{\text{process standard deviation}}$$

Here, process average and standard deviation are estimated utilizing the assumptions concerning the underlying distribution, see e.g. (Montgomery 1996; Pyzdek 1992). The *Z*-score, which corresponds to the number of standard deviations between the sample statistic and the average, is then plotted in the control chart. Normally, UCL and LCL are set to +3 and –3 standard deviations respectively.

In addition to constant control limits, the stabilized *u*-chart allows plotting different quality indicators of a process in the same chart, and it allows varying unit size. All of these properties are utilized in Figure 8, where defect data from the test phase is plotted. Figure 8 contains three different charts, or Z-scores, representing three different quality characteristics. The difference between the *Z*-scores is the chosen base of unit: module size, modification size, and modification grade. All three charts display points outside the control limits; a point above UCL indicates a defect discovery rate higher than normal. However, even if the out-of-control observations differ in number for the charts, the three *Z*-scores seem quite correlated. In this example, it is not possible to determine if any individual metric is the better indicator of test process performance, or if all have useful properties. It is however important to note that the signals given by the three test statistics are quite similar even though not identical. One strength of the stabilized approach is that interpretation is somewhat simplified as the control limits are constant. However, some of the meaning of the raw data is not conveyed by the transformed values. In addition, manual use of stabilized charts is cumbersome, which is a deterrent. If they are to be adopted, some automation is advisable.

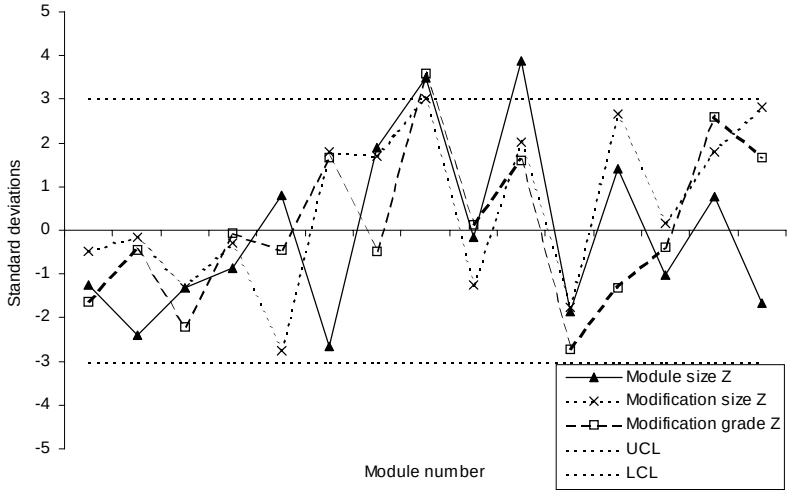


Figure 8. Stabilized *u*-charts for three models of relative defect rate in testing.

If the quality indicator is trustworthy, an out-of-control point is an unusual value, the result of an unusual occurrence, unusual measurement, or both. As Burr and Owen suggested, preferably these attribute data ought to be accompanied by some metric describing testing quality, possibly effort, coverage etc. The data should be of help in explaining the cause of out-of-control points.

This example shows that the choice of quality indicator is not trivial, and requires domain knowledge. At minimum, the plotted characteristic has to make intuitive sense. Preferably, historical data and theory should support the model represented by the quality indicator. The models in this example are univariate models, and may not be sufficiently accurate. Assume that it is known that small and large modules are especially defect prone. In this situation, a better model could be to multiply the *Z*-score with absolute difference between module size and mean module size. The result is a multivariate chart. Multivariate charts are not further discussed in this paper.

Example five

Figure 9 contains stabilized *u*-charts for the phases design to operation, as suggested by Gardiner and Montgomery (1987). Accurate size data for all phases could only be obtained for seven of the modules. The plotted values correspond to Z-transformations of average defects found per total document or code module. One module is plotted on each (imaginary) vertical line. Modules 1, 2, 6 and 7 seem well behaved. Module 3 has a higher than normal rate of defects discovered during code inspection. Module 5 behaved badly in design inspection but slightly too well in code inspection. Module 4 seems to be the most problematic module. However, all modules performed well in operation. If the quality indicator is well chosen, this view is useful as it contains a lot of information and allows for many comparisons. However, this view does not offer an intuitive time perspective with respect to module refinement in subsequent development phases.

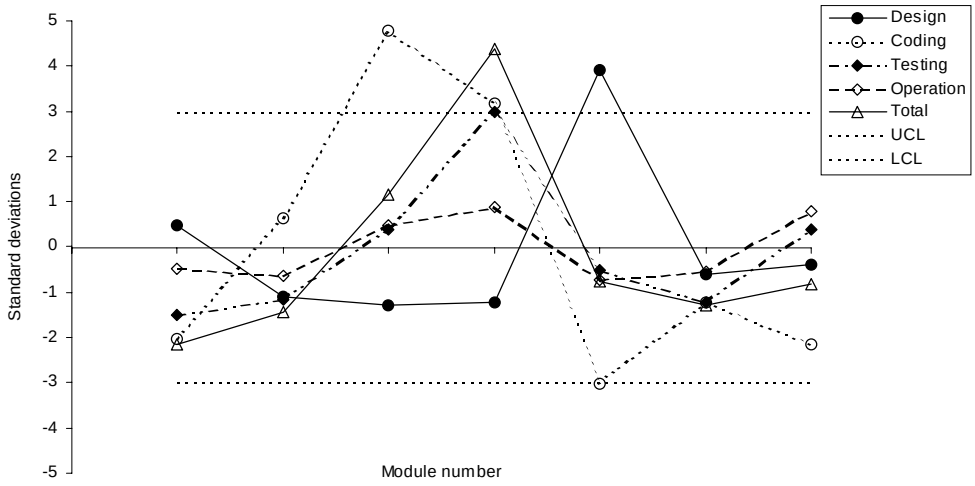


Figure 9. Stabilized *u*-charts for defect rate per development phase.

The time perspective, missing above, is realized by plotting the same data with development phases on the horizontal axis (design, coding, testing, operation and total) and each module as a line of connected points, as in Figure 10. This chart may be used to identify patterns that indicate problems in development. For example, it may be that the defect detection pattern represented by module 4 is typical for "bad" modules, or stinkers. Similarly, pattern for "nice" modules may exist. This data set is too small to examine if representative patterns really exist. Further, the quality characteristic (or model) is likely to vary between organizations.

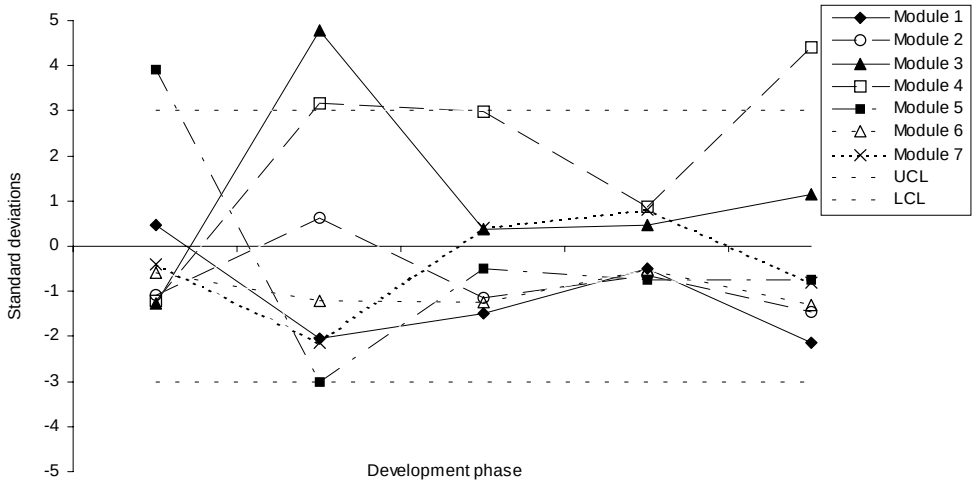


Figure 10. Stabilized *u*-charts for defect rate per module.

4. Discussion

Society has become increasingly dependent on software-based products and services, which have increased the attention to software quality. Statistical process control and especially control charts have played an important role in manufacturing industry in evaluating and gaining a better understanding of process, techniques and tools. The applicability of control charts has also gained increased attention within the software engineering community. However, it is not clear to what extent conventional techniques are applicable to the software engineering field. In this paper we have evaluated and discussed these issues with respect to underlying assumptions and prerequisites for applying various control charts. We have for example shown that the distribution of data is probably the most important issue in control charting. We have also shown that control limits should not be arbitrarily set to +/- 3σ, as σ may be estimated in a number of ways, and in some situations false alarms are relatively inexpensive. We have also verified that stabilized *u*-charts can be applied to short run defect data. Further, we have identified a number of

essential aspects that need to be further investigated, such as the meaning and effect of transformations. Finally, we have suggested using defect rate profiles to identify troublesome modules in early stages.

Possibly, the biggest problem identified in this paper is the lack of data, which forces the use of short run techniques. The underlying problem is that measurement in software engineering environments usually takes place at the end of processes. Typically, completed sub-products are inspected or tested in their entirety at the end of each development phase, exemplified by the data used in this paper. As sub-products can take considerable time to develop, data is scarce, forcing similar artifacts to be compared. Unfortunately, the notion of time is lost and thus much of the point with control charts. For the practitioner, the techniques needed are complicated and the results non-intuitive and hard to interpret. One way forward could be regular measurement within software processes. This would produce more data and focus could be turned from product acceptance testing toward process monitoring and control. In addition, more data allows use of simpler and more intuitive techniques. In the words of Haworth (1996): "These two attributes, ease of use and support for longitudinal studies, are central to the usefulness of control charts."

In many settings, approaches alternative to conventional control charts may be more appropriate. Often, the models discussed above are too simple and have to include more contributing factors in order to be applicable to software processes. For example, we need to include process, product and resource data to truly normalize the data. One interesting example is the regression control-chart described by Haworth (1996). Neil and colleagues (Neil and Fenton 1996; Neil et al. 1996) stress that there are a number of diverse factors implicit in defect detection that need to be accounted for to achieve accurate prediction. They describe how the relatively new but rapidly emerging technology Bayesian Belief Networks (BBNs) can unify diverse aspects, such as size, complexity, inspection effectiveness, and process maturity, to improve current prediction models for improved fault prediction. A BBN represents probabilistic relationships among variables, which each have a conditional probability table showing the probabilities for different node state-input combinations. This technique offers an alternative approach to conventional control charts but fulfils the same objectives by considering, in an intuitive way, more relevant factors. As they are modular, BBNs can be developed for each sub-process. For example, a BBN model of inspection performance, based e.g. on (Sauer et al. 1996), could be used to facilitate interpretation of a inspection defect-rate control chart. This is in line with the suggestion of Burr and Owen to support defect data with metrics that increase their believability.

Traditional quality improvement techniques aim at stable and repeatable processes meeting targets with low variation to produce products. This has influenced a large number of methods in software engineering such as CMM. Whether or not these methods reflect correct assumptions about what constitutes a stable process and what aspects should be stable is still an open research issue. However, given the rapidly changing state of software engineering, both with respect to technology and market, it may be unrealistic to assume stable processes in the most traditional sense. Instead, there may be other types or levels of processes that should be focused on. For example, it is rational to focus on bringing processes that provide measurements into control, i.e. verification processes. In addition, the variation in measurement processes must be small in relation to the measured processes. The rationale is that otherwise measurement noise will obscure the search for assignable causes. Another example is the technology used in the development process, which might never be stable. However, the process of introducing new technology might be a monitorable process. The introduction of new technology ought to have visible consequences, which are to be compared to the expected.

References

Bergman, B., and Klefsjö, B. (1986). *Statistisk kvalitetsstyrning*, Studentlitteratur, Lund.

Bergman, B., and Klefsjö, B. (1994). *Quality from Customer Needs to Customer Satisfaction*, Studentlitteratur, Lund.

Box, G. E. P., Jenkins, G. M., and Reinsel, G. C. (1994). *Time Series Analysis - Forecasting and Control*, Prentice Hall, Englewood Cliffs.

Burr, A., and Owen, M. (1996). *Statistical Methods for Software Quality - Using Metrics for Process Improvement*, International Thomson Computer Press, London.

Draper, N., and Smith, H. (1981). *Applied Regression Analysis*, Wiley-Interscience, New York.

Gardiner, J. S., and Montgomery, D. C. (1987). “Using Statistical Control Charts for Software Quality Control.” *Quality and Reliability Engineering International*, 3(1), 15-20.

Grady, R. B. (1992). *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, New Jersey.

Griffith, G. H. (1996). *Statistical Process Control for Long and Short Runs*, ASQC, Milwaukee.

Haworth, D. A. (1996). “Regression Control Charts to Manage Software Maintenance.” *Software Maintenance: Research and Practice*, 8(1), 35-48.

He, Z., Staples, G., Ross, M., and Court, I. (1996). “Measuring the Software Process Stability and Capability.” *Proceedings of the Sixth International Conference on Software Quality*, Ottawa, 99-110.

Kirkham, D. M., and Roberts, M. A. (1996). “Experiences in Analyzing Software Inspection Data.” *Proceedings of the Sixth International Conference on Software Quality*, Ottawa, 271-282.

Kitchenham, B. A., and Linkman, S. J. (1990). “Design Metrics in Practice.” *Information and Software Technology*, 32(4), 304-310.

Montgomery, D. C. (1996). *Introduction to Statistical Quality Control*, John Wiley & Sons, New York.

Neil, M. D., and Fenton, N. E. (1996). “Predicting software quality using Bayesian Belief Networks.” *Proceedings of the 21st Annual Software Engineering Workshop NASA/Goddard Space Flight Centre*, 217-224.

Neil, M. D., Littlewood, B., and Fenton, N. E. (1996). “Applying Bayesian Belief Networks to system dependability assessment.” *Proceedings of the Safety Critical Systems Club Symposium*, Leeds.

Porter, L. J., and Caulcutt, R. (1992). “Control Chart Design - A Review of Standard Practice.” *Quality and Reliability Engineering*, 8(2), 113-122.

Poulin, J. S., and Brown, D. D. (1994). “Measurement-Driven Quality Improvement in the MVS/ESA Operating System.” *Proceedings of the Second International Metrics Symposium*, London, 17-25.

Pyzdek, T. (1990). *Pyzdek's Guide to SPC: Vol. 1, Fundamentals*, Quality Publishing, Tucson.

Pyzdek, T. (1992). *Pyzdek's Guide to SPC: Vol. 2, Applications and Special Concepts*, Quality Publishing, Tucson.

Pyzdek, T. (1995). “Why Normal Distributions Aren't [All That Normal].” *Quality Engineering*, 7(4), 769-777.

Roes, K. C. B., Does, R. J. M. M., and Schurink, Y. (1993). “Shewhart-Type Control Charts for Individual Observations.” *Journal of Quality Technology*, 25(3), 188-198.

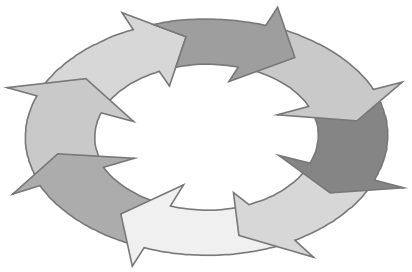
Sauer, C., Jeffrey, R., Land, L., and Yetton, P. (1996). "A Behaviourally Motivated Programme for Empirical Research into Software Development Technical Reviews." *Report 96/5*, School of Information Systems, University of New South Wales, Sydney.

Shewhart, W. A. (1931). *Economic Control of Quality of Manufactured Product*, D. Van Nostrand Company, New York.

Wadsworth, H. M., Stephens, K. S., and Godfrey, A. B. (1986). *Modern methods for quality control and improvement*, John Wiley & Sons, New York.

Wetherill, G. B., and Brown, D. W. (1991). *Sampling Inspection and Quality Control*, Chapman & Hall, London.

Wheeler, D. J. (1991). *Short Run SPC*, SPC Press, Knoxville.



An Integrated Software Audit Process Model to Drive Continuous Improvement

by

Neda L. Gutowski, Software Quality Manager

Global Systems for Mobile Communications (GSM)

Base Station System (BSS)

Motorola

1501 West Shure Drive (3201AR)

Arlington Heights, IL 60004

Email Address: lunich@cig.mot.com

Author Biography:

Neda L. Gutowski is a 10 year Motorolan.

Graduated with a Bachelor's Degree in Electrical Engineering

from Marquette University, Milwaukee, WI.

For the past year held the position as Software Quality Manager of the GSM BSS,

previously a Software Engineer in the same organization working on software development

and new technology prototyping.

Published and presented papers at two Motorola Software Engineering Symposiums

as well as customer conferences in the areas of Knowledge Based Systems and

Quantitative Process Management.

Member of the Software Quality Council at Motorola

and the Union League Club of Chicago Engineering Foundation.

ABSTRACT

The following paper details the Software Audit Process Model that is used in the Global System for Mobile Communications Base Station System (GSM BSS) organization at Motorola. This model is integrated into the existing project life cycle and promotes continuous improvement by feeding back into the Process Improvement and Technology (PIT) team. The project life cycle spans from requirements planning to general availability of a software release. The objective of the audit process described is to ensure that the documented process is followed in every phase of the software development. In addition, the audit ensures the appropriate documentation is collected and organized for future needs, such as customer audits.

KEY WORDS

software audits, integrated, continuous improvement

INTRODUCTION

The GSM project started in the late 1980's, at which time there was no disciplined audit process. Through the years, as the organization matured on the SEI SW-CMM scale, the software audit process followed suit. Our software audit process matured from its once ad hoc state to a planned and defined, data driven, and currently an optimizing state. The audit process that will be described reflects a Level 5 culture and demonstrates Level 5 disciplines and behaviors.

This paper covers the following areas: our organizational structure and our culture, the quality management function, making the organization aware of the software audit process, a timeline of our feature development lifecycle with the software audit process stages noted, a description of the software audit process stages, the software audit process model, and the continuous improvement activities that result from the audit data. The key words in the title of this paper are 'Integration' and 'Continuous Improvement'. Both are core beliefs that the organization recognizes and lives by daily. These beliefs are ingrained in our culture. Integration is about having the processes integrated. For example our quality management, metrics program, and defect prevention activities are all integrated into our project life cycle. Continuous Improvement is about ever evolving efforts to improve. Continuous Improvement is part of everyone's job, and is fully supported and sponsored by our management. Opportunities to identify improvements are integrated throughout the project life cycle.

Throughout this paper terms may be used that represent specific language used by our organization. A glossary in appendix A contains some definitions to aid in understanding the meaning of these terms.

Organization Structure and Culture

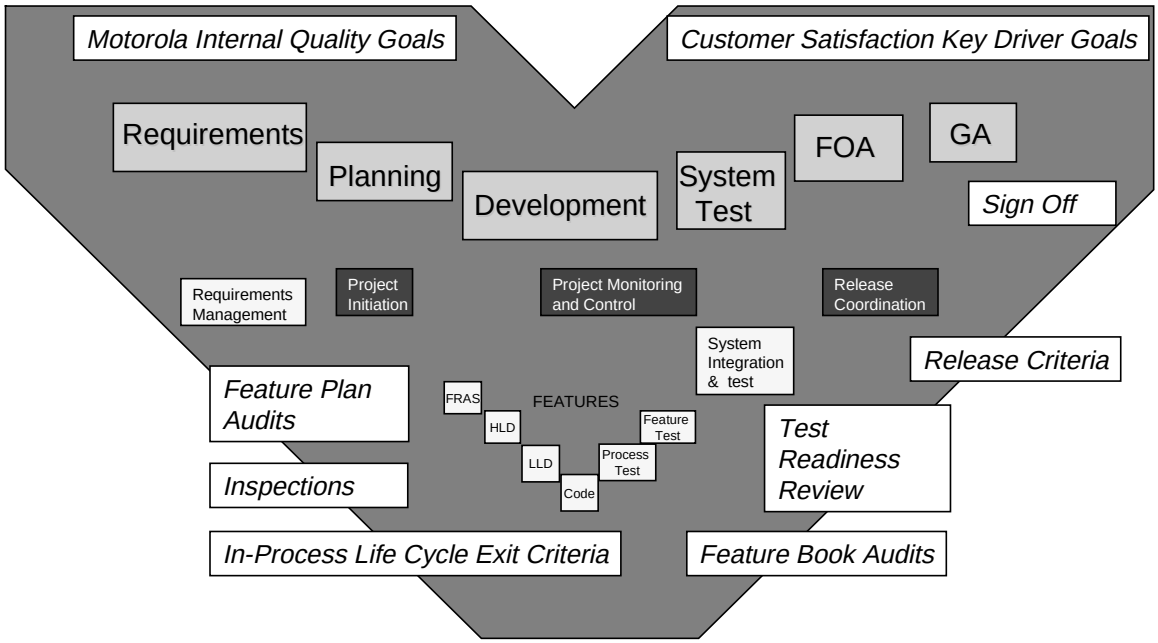
Our Software organization has approximately 240 engineers. 86% of the engineers are located in Arlington Heights, IL, while 14% are located in Swindon, UK. Our engineering teams include Requirements, Software Development, System Integration & Test, Quality, Training, Project Management, Metrics, and Customer Support. The teams communicate regularly, there is a small quality team, and everyone owns their own quality. Some critical relationships the Quality Engineering and Assurance (QEA) team has during this audit process are with the feature development and project management teams. As the software audit process model is discussed, these relationships will be noted.

Our project is software development for the GSM BSS. GSM is a European cellular system standard. Our customer base currently stands at 80+ worldwide. There are millions of assembly equivalent lines of code in the product. The GSM BSS software organization is the first Motorola organization to be assessed at SEI SW-CMM Level 5 (October 1997). Although our culture has many aspects, this paper will key on the 'Integration' and 'Continuous Improvement' dimensions.

Our software quality audit process is integrated into our life cycle and has the ability to promote continuous improvement by using data obtained from these audits. The software audit process data is used to continuously improve both our development life cycle processes as well as the software audit process.

The Quality Management Function

The QEA team function covers many roles, responsibilities and key activities ranging from customer activities to quality improvement activities as well as planning quality activities and conducting audits, reviews and assessments. In the area of audits, reviews and assessments, the QEA team is responsible for process, product and supplier audits. The organization uses the Fagan inspection methodology which ensures essential participation, consultation and follow through at the end/start of each phase of the development life cycle. A Quality Management Plan (QMP) exists which details the product quality goals and quality management activities for each phase of the development life cycle. A QEA handbook which details the QEA function also exists. The QEA members follow the handbook to perform their job function.



The above figure shows various Quality Management (italicized) activities that occur during the Project lifecycle. The top five boxes (Requirements through GA or general availability) represent the project life cycle. The boxes with white text represent the project management process. The small boxes represent the development life cycle. The development life cycle is a subset of the entire project life cycle. This paper will concentrate on the feature audit, which is an audit of the development life cycle. Although the QEA team performs other activities related to the project life cycle such as release sign-off, the focus of this paper is on the development life cycle.

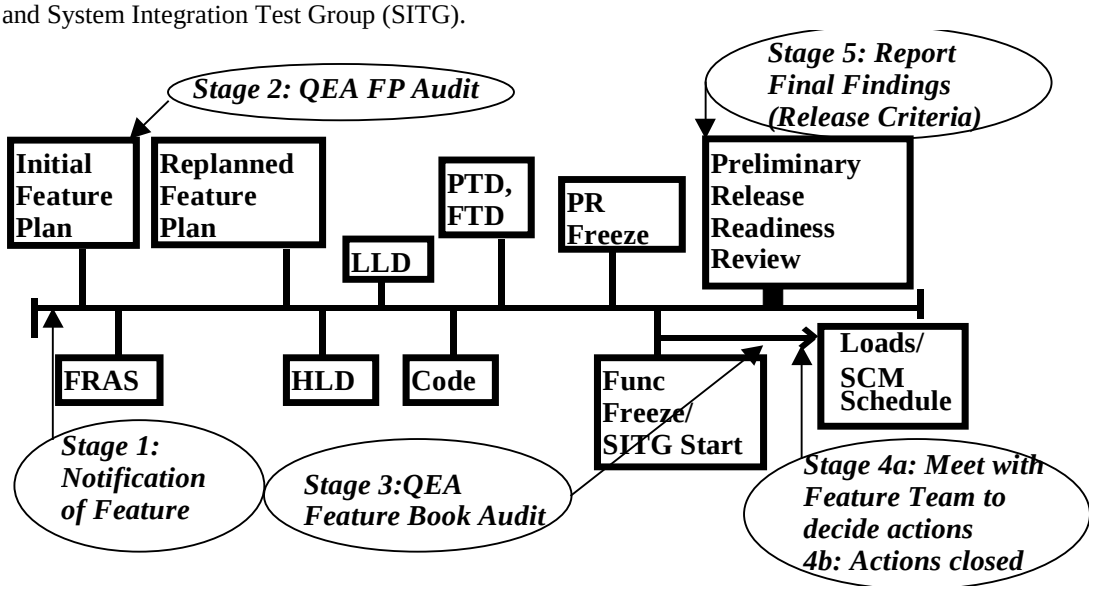
Awareness of the Software Audit Process

Having been a software developer for 9 years on the GSM BSS product prior to my role as a QEA team member facilitates my job working with the developers. I know how they think and I know how they perceive the QEA team. Number one in importance is sometimes when Feature development gets pretty intense all else that's out of sight, is out of mind.

The QEA function needed to be communicated to the development engineers. They were given an opportunity for feedback which is another of our core beliefs – 'collective knowledge'. This was important to achieve their buy-in. Basically this entailed presenting an awareness 'Road show' at each of their section meetings. Also development engineers do not want to listen to anything they feel is a waste of time. Our road show is nothing special because they have these meetings anyway, the spotlight this week just happens to be the software audit process. This was a successful way of making our engineers aware of what is expected of them. Before the road show people didn't know when we were going to do the audit, what they needed to do and when they needed to do it. They didn't take time to study the detail in the paragraphs of our hundred page QEA handbook document which talks about the audits. They were often late turning in their feature books to be audited because some people actually waited until we asked for the audit to happen tomorrow to put their book together today. This road show details the procedure, who is involved, when it happens, and what is looked for. This information is presented in easy to understand tables for each stage of the audit process. That's all. Twenty minutes maximum of their time saved a bundle come feature audits for that particular release. Feature Champions create their feature books throughout the development life cycle and are prepared to deliver the feature book to QEA when the audit schedule calls for their feature.

Software Audit Process Stages

The next figure is the development life cycle timeline with the software audit process stages identified. The development life cycle is identified in the boxes: starting with the initial feature plan and ending when the code is in the load (SCM = software configuration management). The acronyms in the figure stand for in order Feature Requirements and Architecture Specification (FRAS), High Level Design (HLD), Low Level Design (LLD), Process Test Design (PTD), Feature Test Design (FTD), Problem Report (PR), and System Integration Test Group (SITG).



The software audit process stages are identified in the outlying ovals marked as 'stages'. A description of the stages are detailed below. They include the awareness of the feature, to auditing the initial feature plan as a proactive measure, the auditing of the feature book, the feature team meeting, and sharing the results at the preliminary release readiness review.

The feature audit described in this paper is at the project subsystem level. All features characterized as customer visible are audited, the remaining internal features are audited if time permits. The feature audit is a quantitative assessment of conformance to the project's defined development life cycle. The audit data is used to plan and estimate as well as to improve the process. This will be discussed below.

Stage 1: The audit process starts upon notification of a feature. This is done via an e-mail sent to all interested parties, managers and developers, project management, and quality. Therefore the feature champion does not have to do anything extra to notify the QEA team that the feature is planned.

Stage 2: The Feature Plan audit is to ensure tailoring compliance. Tailoring guidelines are defined in the development life cycle. An example of tailoring the development life cycle is as follows: the HLD phase is not needed, only LLD and code is required. The feature plan would indicate that the HLD phase of the development life cycle is not necessary for the development of the feature.

The Feature Plan audit is a separate exercise than the inspection of the feature plan. An inspection involves the appropriate development and project managers and focuses on the resources identified, estimated dates, timeframe of delivery, etc.

The Feature Plan audit is proactive vs. reactive catching potential problems during the planning stages of the feature. The feature plan data is used to estimate the audit schedule of the complete feature, or feature book. The schedule estimation technique is based on the number of audited items developed with historical data from audits. From this data the number of days to audit a feature can be determined. Initially the feature audit schedule was not based on historical data, audits were performed as called for by the QEA team. The audit schedule now allows two weeks after the latter of the following, either the feature test complete date, or the date the feature is included in a load or SCM schedule. Each feature included in a load is scheduled in sequence based on the above criteria and also the calendar availability of the QEA team member performing the audit. The audit schedule is only revised if the feature test complete date changes or the SCM schedule changes.

Stage 3: The Feature Book audit is to ensure the feature was developed as planned in the feature plan. This audit is done by the QEA team independent of the feature team. The supporting evidence is included in the feature book. The audit examines that the feature followed the development life cycle and that the Fagan inspection methodology was performed at each deliverable stage of the life cycle. Data is collected of the common audit issues from the feature audits. Data is also collected for the audit timeliness. The use of this data will be described in the metrics section of the paper. The audit results can be viewed on our intranet. The results are always current and the issues are clearly marked with a yellow or red color, while the non-issues are green. This allows valuable feedback for the feature team in a timely manner.

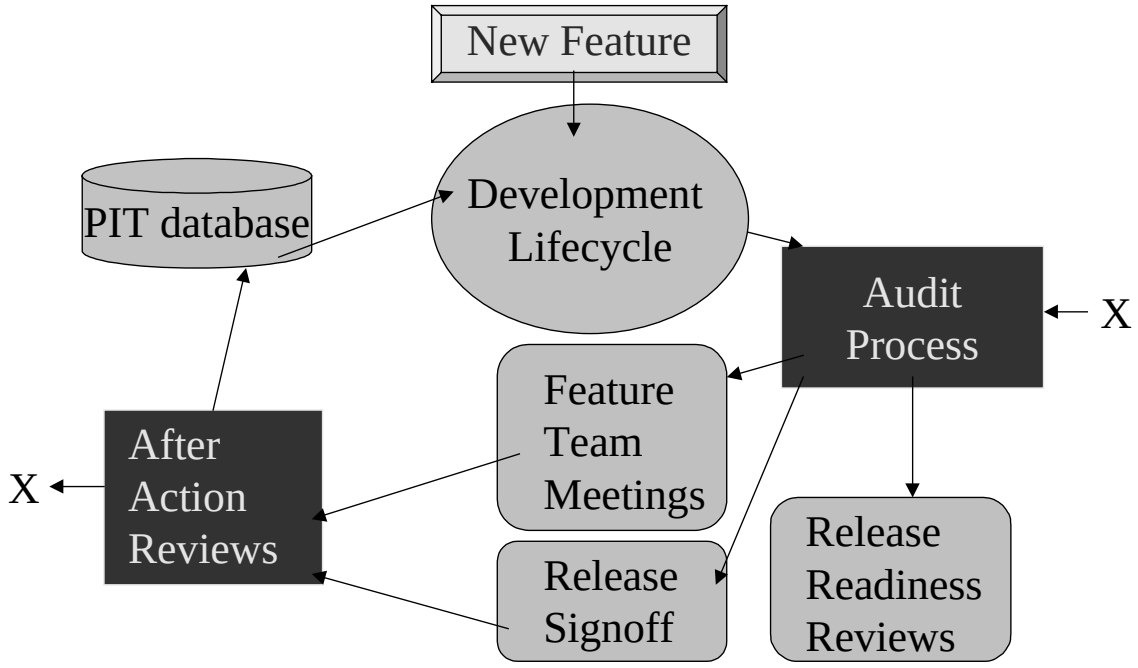
Stages 4a & 4b: The Feature Team Meeting is where the audit results are shared with not only the feature owner, but the entire team. The Feature Team owns the feature, and they take responsibility to resolve the actions from the audit. It is all about harmony and about working well together without walls. Problems are worked without casting blame. Any questions that the QEA team has that may or may not be issues are easily resolved in the meeting because the entire team is there to discuss them. Two weeks is allowed for the team to resolve the issues identified in the audit and shared at the feature team meeting.

Stage 5: The data from the audits are shared at the preliminary and final release readiness reviews. These reviews are phases in the project life cycle and are planned and scheduled by the project manager responsible for the release in which the features are included. These reviews are attended by all development managers and project management as well as quality. The adherence to the process is presented for each feature audited based on a scale of 0-100% process adherence. This value is calculated based on the number of outstanding audit issues divided by the total number of audited items. This data is stored in the feature profile which is a compilation of various data collected for a feature including test results and number of problems opened against the feature. Also presented is the number of passes it took to get to the process adherence percentage. This data is important to the QEA team because each pass beyond the initial pass of the audit and the follow up after the feature team meeting takes additional unplanned time of the QEA team.

Software Audit Process Model

The next section describes the software audit process model. The following diagram is the software audit process model. It starts with a new feature. This feature is implemented following the documented development life cycle. The audit process is performed on this feature. Data from the audit feeds into the following: feature team meetings, release readiness reviews, and the release signoff process.

After the release is complete and available, the QEA team holds an after action review. An after action review is held for the quality function, which includes the QEA audit and the signoff function, and includes a handful of developers who were on feature teams for the release being evaluated, and project management responsible for the release. This review brings all together to discuss things that went well and things that can be done differently for the next release. For example, a pareto chart of the audit issues is generated and discussed. The chart is compared to the issues of the previous release to determine if process changes made since the last release prove effective. See the metrics section of the paper for details on this metric. The actions from the review are handled the following ways: any systemic defects are entered into the Process Improvement and Technology (PIT) team database for the process improvement team to review and resolve, while software audit process actions are directly used to improve the audit process for the next release (the 'X' in the diagram).



An example of the use of the model follows: in a recent after action review it was noted that some of the common audit issues had not been corrected from one release to the next. After the PIT team evaluated some of the problems, the engineers asked for the QEA team to create a checklist that they could fill out while they were completing their feature book. The key point here is that those who develop the product are the ones responsible for improving it. If the QEA team were the ones who decided a checklist would be the best means to reduce issues and then try to change the process for the engineers, they would probably not buy into it as smoothly. The PIT team requested this checklist. Everyone in this organization owns their own quality. They feel responsible for what they deliver and were asking for means to check their work. In our next release we will measure the effectiveness of the checklist by examining the pareto of audit issues. Even though this checklist is not required until the next release, an engineer who had a feature which came in late in the previous release actually filled out the checklist and submitted it with the

book. This person stated that the process of preparing the feature book for audit went quicker than usual and they also felt confident about what they were turning over to the QEA team. Although there were some issues with this audit, the number of issues for the number of items to be audited was much lower than the average issues for similar audits in the past. An example of the feature book checklist can be found in Appendix B. The QEA team created the checklist and reviewed it with the PIT team. The PIT team representatives then took it back to their section meetings to share it for comments with their development team. This way there were no surprises to the development community. The QEA team enhanced the feature checklist to include a signoff of the feature champion. The project management team formerly had to complete a release checklist at the end of the release. They used to make their rounds to all section managers and have them sign that the feature books were in place. This step is now eliminated since the signoff by the feature champion is sufficient for this purpose.

When the QEA team signs off on a release, the data from the audits is used as part of a release criteria package. The release criteria includes an entry which verifies that all scheduled audits have been conducted and the issues identified were communicated to the feature team. As part of the audit results QEA documents the path to and the document id of the deliverables for each feature in the release. This audit data is also used by project management to generate the document roadmap for the release. The project managers are quite pleased we have made their lives a bit easier at the end of the release when things get hectic.

The software audit process model promotes continuous improvement by using various data collected during the audit process. The process is quantitatively managed for continuous improvement.

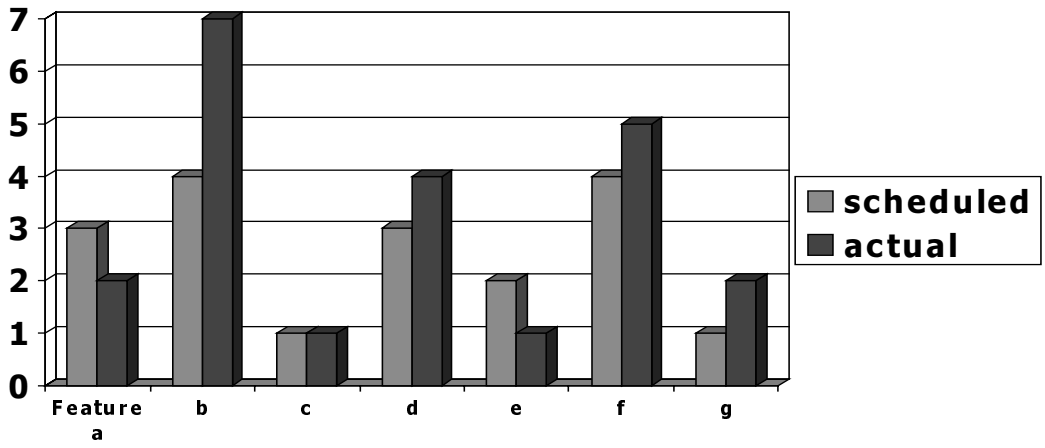
Metrics for continuous improvements

The software audit process is measured with the following metrics: audit coverage and audit timeliness. The QEA team wants to know how well they plan the audit schedule and estimate the effort of each feature audit. This data is used to better plan and estimate for future releases to be audited. These metrics are listed below.

Quality Audit Process:

Audit coverage – the number of audits completed vs. planned in the published schedule
Audit timeliness – the number of days the audit was conducted vs. scheduled days
- the planned start date of the audit vs. the actual start date

The following diagram is an audit timeliness metric which measures the number of days the audit was scheduled vs. the actual days. This data is analyzed for reasons why scheduled days may not be near the actual days.



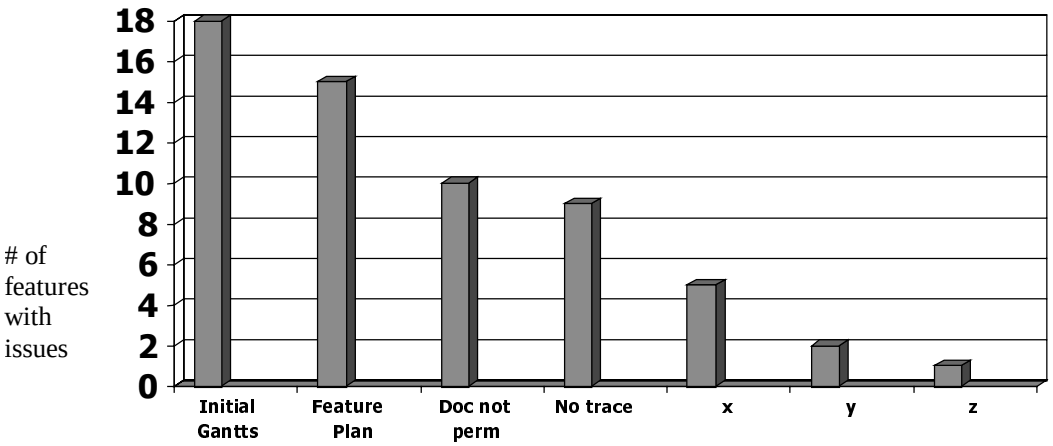
In the case of this metric chart, analysis showed that Feature b was missing the feature plan, somewhere along the way it went missing into an electronic black hole, thus it took longer to do the audit since they had to recreate the feature plan. Other late times were due to feature books arriving late. A feature champion process training class has been created in response to a need for common feature champion roles amongst the various development areas. One major finding was that initially we did not (in our audit schedule) allocate time for the feature team meetings. When planning the schedule for the next release, we allocated time for these meetings to improve our accuracy of audit timeliness.

Development Process:

Pareto of Audit issues – feature plan and feature book issues

The development process is measured with the following metric: Pareto of Audit issues. A pareto is a bar chart showing items ranked from the most to least frequency of occurrence. A chart is created for both the feature plan audit and feature book audit. This chart is distributed to all the managers. It is also shared at the preliminary and final release readiness review which is held by project management. And finally the data is brought to a PIT team meeting where it is analyzed and actions recommended by a problem report being opened in the PIT database.

The following chart is an example of the pareto chart for feature book issues. The y-axis data is the number of features with the issue and the x-axis data is the types of audit issues. Common problems such as initial gantt charts (a gantt chart is a tracking chart which shows, for example, effort and duration of tasks) and feature plan not being updated at the end of the feature with actual values, documents were not being moved from the working area to the permanent area, traceability missing, etc. In the next release a comparison will be done with the previous release audit issues to see what kind of impact the feature book checklist has.



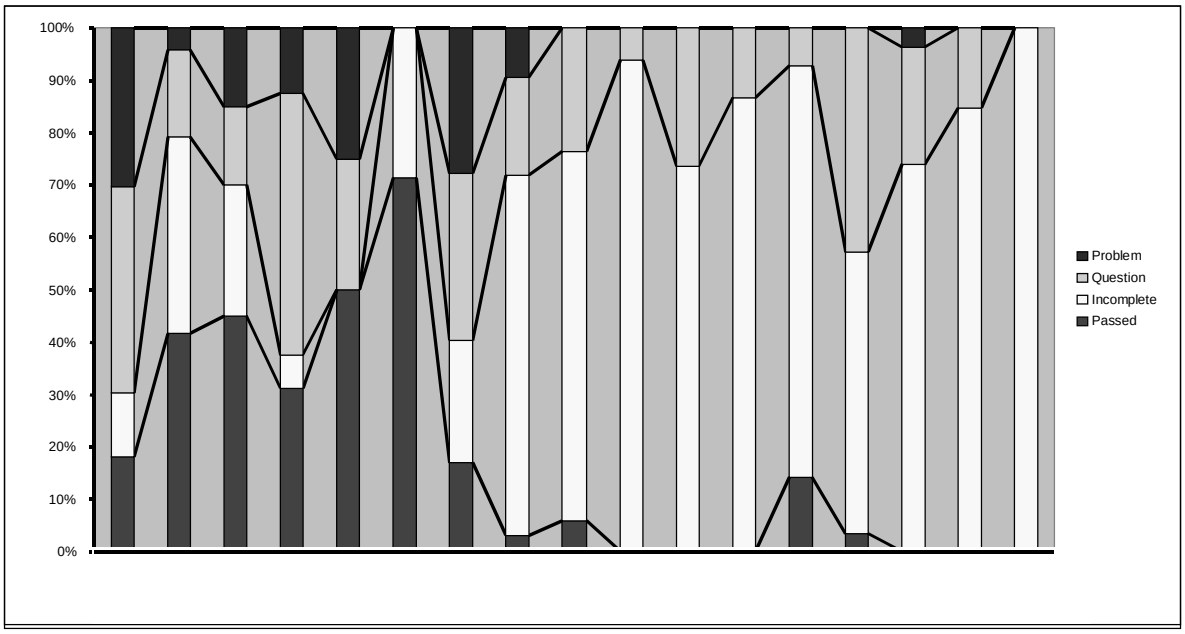
Some examples of changes to the development process due to systemic problems found during audits are as follows: updates to the templates for feature planning and design, enhancements to the process documentation including the development life cycle process, the feature planning and tracking process as well as others. These items are handled through the PIT database and the PIT Team.

Audit Results:

Percentage of audit issues – the audit status of each feature

Other data available during the software audit process are the audit results to date. The chart below shows for each feature, the percentage of problems, questions, incomplete and passed audited items. The chart is described as follows: the y-axis is the percentage of audited items that are problems, questions, incomplete and passed items (see legend), and the x-axis has a bar for each feature being audited (note the x-axis is blank due to data confidentiality).

This data allows the feature team to see the current status of their feature and how many issues still need to be resolved following the audit. A feature which has resolved all issues will reflect a solid bar with all items 'Passed'. The QEA team ensures that all features are 'Passed' at the release of a software load.



Summary

Communication of the software audit process and achieving the development organization's buy-in was the first important activity in introducing the audit process. Sharing the audit results with the entire feature team is essential to achieve timely resolution of issues. The after action reviews provide a means for the people QEA works closely with to help create actions for things that were identified as systemic defects. It also to covers recommended improvements in both the software audit process and the signoff process. It involves key people from the organization, focusing on the QEA team's customers.

As a result of common issues occurring across multiple releases, the QEA team assembled a Feature Book Checklist for the feature champion to submit with the feature book. This serves many purposes: reducing the time it takes for the feature champion to prepare the information in the feature book, reducing the cycle time for feature audits, aiding in defect prevention by reducing the number of audit issues, and eliminating a step in the signoff process The checklist was reviewed by the process improvement and technology team and also shared at all the development section meetings before approved.

The work done to ensure a quality product prepares our organization for visits from our customers. When the customer requests to see that we follow our development life cycle, the feature book presented to them has been audited and is in proper order.

In summary, the integration of our audit process into our development lifecycle is a means to drive continuous improvement! Our organization lives with a commitment to improve & a pervasive process culture.

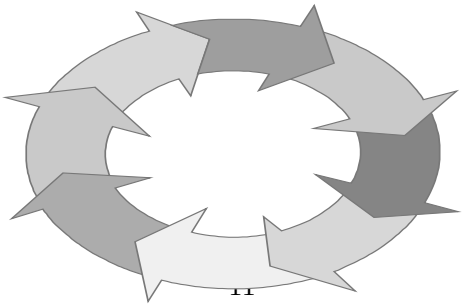
Acknowledgements

The author wishes to acknowledge the support received from Corinne Miller, Director of GSM Products Division Quality. She has been a great mentor and a key player in this organization's successes.

References

[1] "Productivity Improvement Through Defect-Free Software Development". Michael Fagan Associates, June 1, 1992

.



APPENDIX A

Glossary

Development life cycle – a model of the phases of the software development component ranging from feature planning to designing to code to feature testing.

Fagan inspections – the process used to inspect all deliverables (documents and code) [1]

Feature – new functionality that is added to the system.

Feature Plan – a detailed plan for a feature documenting estimates in software size, resource allocation, scheduling and critical resource requirements.

Feature Book – one of several binders containing evidence of all deliverables noted in the feature plan, as well as copies of the inspection forms for each deliverable.

Feature Champion – the person responsible for the planning and delivery of new feature. functionality into a release. This person is usually the technical lead of the feature team.

Feature Owner – the person ultimately responsible for the feature's delivery and quality and monitors it throughout the feature development. This person is usually a manager.

Feature Team – the team of engineers responsible for completing all the development work for a feature.

FOA - First Office Application (of a software load, occurs before GA)

GA - General Availability (of a software load)

Load - the integration of the software feature functionality and software problem fixes developed for a release.

Project life cycle – a model of the phases of the life of a project ranging from requirements to the general availability of all subsystem releases and customer documentation.

QEA – Quality Engineering & Assurance team (renamed from the traditional SQA).

Release – the work product which combines new feature functionality and/or maintenance work to form a new version of the work product for distribution to the customer base.

APPENDIX B

Feature Book Checklist

- □ Index of paths to all documents included
- □ All inspection forms have paths (if not code)
- □ All inspection forms have version numbers (if not code)
- □ All inspection and reinspection forms included and signed and dated
- □ Version number in inspection sheet matches document or explanation if discrepency:_____
- □ All documents in Permanent (including Traceability Matrix)
- □ Traceability Matrix included and complete
- □ Code Coverage output included as part of Test Results Summary Form
- □ For any Blocked/Failed tests include Status
- □ Feature Plan completed & included (with signature form or copies of electronic signatures) in Feature Book
  - □ Actuals
  - □ Resides in proper directory
  - □ Tailoring of Life Cycle deliverables indicated
  - □ Documentation Tree is accurate
  - □ Lines of Code match inspection sheets or explanation given
  - □ if Code Actuals >> or << Estimates (25%), give explanation:

    _____
- □ Copy of completed Gantt in feature book and in directory
- □ Feature Book ready 2 weeks after Feature Test completion or Feature Delivery (in load). If delinquent state why:_____
- □ FRAS updated at end of Feature - if applicable check box
- □ Feature Champ has attended Feature Champion Process Training
- □ Any Notes/Comments to QEA:_____

Feature Name:_____
Feature Champion Name:_____
Feature Champion Signature:_____DATE:_____

The Effects of the Year 2000 Problem on the Wintel Duopoly's Control of Internet Commerce in the International Marketplace for Multimedia Mindshare

or

How Do We Get These Damn Things to Work?

By Lincoln Spector

Lincoln Spector was introduced to the computer industry in 1983, when he bought his first computer, an Osborne 1, and within weeks watched it drop in value from $1500 to $200. After that experience, there was no option left but to become a humorist.

Gigglebytes made its debut in the August 12, 1986 issue of Northern California Computer Currents. The column now appears in 13 publications in 4 countries. Lincoln is also a Contributing Editor for PC World, where he writes the allegedly serious Answer Line column.

Lincoln Spector is the speaker's real name, appearing on his birth certificate and driver's license. He spent his youth going to movies and performing in amateur theatrics, and has been trying to return to that lifestyle ever since.

When he isn't writing or speaking publically, Lincoln spends most of his time looking for affordable health care.

The Maltese Penguin

A man's man amongst the Unix

by Lincoln Spector

Note: The following article has nothing whatsoever to do with my speech. It was originally published in the June 16, 1998 edition of *Bay Area Computer Currents.*

The fog was as thick as a Windows error message. I was returning to the office after another difficult case: The Feds had just discovered a two-digit year field in an obscure, Department of Agriculture database. If it wasn't corrected by the year 2000, they were gonna bomb Norway. I fixed the problem by strong-arming a systems administrator in Justice into keeping 1999 going for a little longer.

The name is Rowe. Mack Rowe. Private Consultant.

I entered the office and tossed my hat at the coat rack. It missed, hit the heater and instantly burst into flames. That reminded me: I had some work to do in Windows.

I was opening a bottle of bourbon when the door swung open and a large penguin waddled into the room. "Mr. Rowe?" he asked, jumping into a chair.

"Yeah. What's it to you?" I offered him a Lucky Strike, but he declined.

"Have you ever heard of Linux?" he asked, producing a gold cigarette case and pulling out a sardine.

"Of course. It's a free version of Unix."

"Someone's holding it back from mass acceptance. I want you to find out who it is. We believe his initials are BG."

"My fee is $100 a day, plus expenses," I told him.

He raised an eyebrow. "This is Linux. You're supposed to do it for the love of technology."

We negotiated. I agreed to take the job for the love of technology, added self-esteem, a warm and fuzzy feeling all over, and the ATM pin number for Andy Grove's Swiss bank account. He tossed a CD-ROM onto my desk and waddled out.

Bundled Nerves

The next morning I got up at the crack of noon. I started my investigations at the office of an old friend, Dell Gateway. He wasn't happy to see me.

"BG didn't send you, did he?" he asked, nervously drumming his fingers and typing "sefasdfasdfasdfsdfasef" into the annual stockholder's report. "I asked him if we could put the name of our computers on the screen somewhere. He absolutely forbid it, of course. Not that I'd do anything to cross him, but I'm scared he'll drop by and notice our logo on the box."

I assured him that BG hadn't sent me. "Know anything about Linux?" I asked, sinking comfortably into his couch.

He turned whiter than Michael Jackson. "BG did send you, didn't he?"

I sighed, got up, and let the cat out from under me. "For the last time, BG didn't send me. I'm working for a penguin who wants to do why more people aren't using Linux."

He smiled with all the sincerity of a politician in November.

"And what does this…penguin want from me?"

"He wants you to bundle Linux with some of your computers."

"Every computer we sell comes bundled with one of BG's operating systems. That's all I can say."

“Ever think of offering an alternative?”

Gateway pushed a button on his desk. Two powerful men came in and started beating me with printer cables.

When I came to, I was in a large cardboard box filled with Styrofoam peanuts. I climbed out, tipped the FedEx man, and went on my way. Dell Gateway was scared--scared as an Apple employee watching Steve Jobs talk about vision.

Back to School

I next dropped in on Anna Conda, a system administrator for the local university. I knew she’d talk if I made the right moves. I know how to handle women.

“Okay, okay, I’ll tell you,” she said. “Just stop begging!

“Linux is great,” she continued. “It’s the best OS in the world. Look, I maintain a network for 53 linguistic professors, and they used to put all sorts of things on their Macs. I couldn’t tell what those programs were…they were in, like, all sorts of languages.

“Then I gave them all Linux; removed that other OS entirely. I found some great point-of-sale software on the Web which I converted to handling grades and Ph.D. theses.

“Now I don’t get no trouble from those professors. In fact, as far as I can tell, none of them are even using their computers.”

Cruel and Unusual

When I got back to the office, I gave the penguin’s CD-ROM a whirl. Things weren’t fitting together. Why was BG muscling the vendors? Was Linux really scaring Linguistics professors? What was the real reason they’d cancelled *Ellen*? The answers, I knew, were on the CD-ROM.

The installation program got off to a good start, asking me if I wanted to wipe everything off my hard drive. The options were XY-14 and Q40. I picked the later and soon knew that I didn’t have to worry about my corrupted Windows registry anymore.

Next, it asked me to pick an X-server. The help file didn’t say what an X-server was, but it did tell me where I could find out on the Web--nice thing to know after I’d formatted my hard drive.

In the end I picked an X-server at random, and was able to boot Linux in a one-color mode. Black on black.

I was about to douse the keyboard with bourbon when my client waddled in. “Well,” he said, “have you found out why Linux isn’t catching on?”

There’s nothing like the sound of a penguin going through a second story window. It enhanced my love of technology, increased my self-esteem, and gave me a warm and fuzzy feeling all over.

Hierarchical Organization of Test Cases

Michael Ensminger
Partes Corporation
10635 NE 38th Place, Suite B
Kirkland, WA 98033
Email: mensminger@asqnet.org

About the Author:

Michael Ensminger is currently Manager of Quality for Partes Corporation in Kirkland, WA. Partes develops applications and data services for the financial community. Previous experience includes directing the testing of retail banking systems and shrink-wrap graphics software. He earned an MS in Computer Science from the University of Texas at Dallas with a concentration in software engineering. He also holds the American Society for Quality (ASQ) software quality engineer certification (CSQE).

Prepared for and presented at the 1998 Pacific Northwest Software Quality Conference / Eighth International Conference on Software Quality

Abstract

A three-tier organization for test cases is presented. The criteria for inclusion in a tier are based on the pragmatic needs of the tester. The quick test tier aims at testing the most commonly used functionality in common situations. The full test tier fleshes out the quick test tier to fully test the functionality of a feature. The detail test tier discovers those defects that would not prevent shipping but allows the developers to fix them as time permits. The benefits of the approach are many. The organization allows the project to develop objective release criteria. It also provides a roadmap for test organization.

Introduction

From a practitioner's point of view, there is a wealth of information available on how to document test cases. These include the formal IEEE and government standards, semi-formal references in the test literature, and informal formats distributed on the web or homegrown solutions (Encyclopedia, 1994). These sources advise the practitioner how to document the tests but not how to organize the individual test cases. In fact, there is little or no information as how to classify and prioritize the test cases for execution.

In this paper, I propose a three-tier organization for organizing test cases. The ideas presented in this paper are not new. This organization is meant to document one method of classifying test cases. It does not suppose to replace the systems already in place for many organizations. The three tiers are easily adapted to work within an existing system.

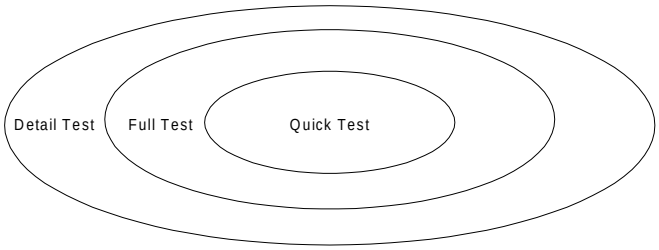


Figure 1: The three tiers

The three tiers encompass all of the test cases that comprise a test suite for a project. Each tier's criteria are based on pragmatic criteria - always keeping the tester in mind. If more granularity is required additional tiers may be inserted. The three tiers can be thought of as three concentric circles. Working outwards from the center, each subsequent tier encompasses the results of the earlier tier. Once the outer rung is reached, all test cases are executed.

"Quick Test" Tier

The innermost tier is the quick test tier. This tier contains the test cases that will most likely turn up defects in common functionality.

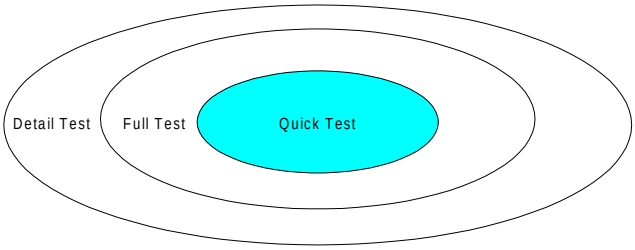


Figure 2: Quick test tier

Criteria for Selection

The general principle behind selecting the test cases in the quick test tier is to find those defects that prevent the product from shipping. Based on this principle, consider the following criteria for selecting test cases for inclusion in the quick test tier:

- Length of time to execute
- Usage profile
- Error handling
- "Must Have" functionality
- Coverage Criteria

Length of time to execute determines the size of the quick test tier. The upper bound on the number of test cases in the tier is based on how long the project team decides is acceptable between completion of the build and first feedback to the developers. The quick test tier "qualifies" the build for further testing. This is the principle behind the "Smoke Test" (McConnell, 1996) processes. This length also depends on the frequency with which code is handed off to the test group. In a nightly build and test scenario, the quick test should be completed shortly after completion of the build, ideally before anyone comes in the next morning. If code is handed off less frequently, say biweekly, spending a day on the quick test is not unreasonable.

Once the upper bound on the size of the tier is determined, the usage profile for the product provides the largest set of test cases. Usage (operational) profile is used in the same sense as in reliability measurement (Musa, et al., 1996). The basic concept of a usage profile is to understand what activities are performed and how often by the different types of users of the product. Once the usage profile is in hand, test cases for the most common operations are first included in the tier. Test cases are added generally according to the Pareto principle (Kan, 1995). The test cases should cover the 80% of the functionality that is most commonly used. Note that the test cases selected do not exhaustively test the feature in question. The test cases selected should verify that the normal use of the feature is functioning properly.

The test cases so far have exercised the product in a normal fashion without any errors. For the quick test to truly verify the build, it is necessary to exercise some error handling. Similar to the usage profile, an "Error Profile" should be developed to identify those errors that are most common. Test cases are then selected to examine the error cases that are most likely to occur.

So far, the quick test tier includes the most commonly used functionality and those errors that are most likely to occur. However, there is another class of functionality that needs to be verified often. This "must have" functionality is included in the quick test tier. For example, end of year processing may not show up in the usage profile but it is crucial to the product. Similarly there may be some features that are not used often but critical to the marketing of the product (so-called checkoff items for magazine reviews).

Finally, a set of test cases are collected and classified as the quick test tier. Based on the size of this collection and the time limit set on execution, test cases may need to be eliminated. The opposite is also true if the time to execute the set of tests is less than the time limit selected. One means of adding test cases is to examine the coverage obtained by the existing tests. Generate additional test cases to increase the code coverage (Beizer, 1990) of the quick test. Keep in mind that the time it takes to execute the quick test may decrease over time through automation, tester ability and performance enhancements to the product. Periodically, it is important to examine the quick test and see if any more value can be added.

This set of criteria works for many product types. Additional criteria should be created based on the needs of the product. For example, in the shrink-wrap software business, installer testing could be added to the quick test tier. Also, some projects include contractual obligations that could generate quick test criteria. An example is year 2000 compliance tests.

In all cases, the test cases in this tier are aimed at finding those defects that prevent the product from being tested. In particular, notice the user interface tests are not included in this tier. This is because a cosmetic problem is not likely to prevent the product from being tested. For example, exhaustively testing the text, font, size, etc. on all dialogs, while useful, is unlikely to uncover any defect that needs to be corrected immediately.

Also, the test cases should identify those defects that are blocking further testing. A common frustration among developers is waiting for feedback on a fix only to find out that the test team is being blocked by some other defect. Execution of the quick test gives the development organization speedy feedback on the latest build.

Application to Other Phases of Development

The criteria listed above are based around test cases for system testing. A similar set of criteria can be developed for other phases in the development cycle, such as unit testing. Consider the following criteria for use during unit testing:

- Length of time. Probably much shorter, measured in minutes. This is a quick verification that a developer can do to tell if a change has had any obvious adverse effects.
- Test most common calls using the most common data
- Invoke common errors that the unit is likely to occur
- Coverage criteria

Similar criteria can be developed for other steps in the development lifecycle.

Example Application

To provide an example of applying the criteria, it needs to be in the scope of a project. For the examples in this paper, consider the following simple utility. The utility generates lines of code metrics for a source code file. For each development language, there is an input file defining the end of line character, line continuation, beginning and ending quote delimiters, etc. The program outputs to the screen the number of physical lines in the source file, number of executable statements, and number of comments. The utility identifies the source language by the file extension. The utility may use wildcard characters to process more than one file at a time. A command line switch is available to have the utility traverse the directory structure.

The time limit and usage profile must be identified first. Since this is a simple utility, the team decides that the maximum time for executing the quick test is thirty minutes. The usage profile is for the utility to be invoked on files of only one source language, usually C or C++. Single or multiple file execution are both common. Traversing an entire directory tree is not common.

The following test cases hit the basic usage: 1) Invoke utility with full filename.c; 2) Invoke utility with *.c in directory with 10 files 3) Repeat test cases for C++ source files. The following common error / unusual conditions are identified: 1) File not found; 2) Source language does not have input file; 3) Zero length file; 4) Extremely large file; 5) File on a remote system. The marketing group has determined that the ability to traverse directories is a must have feature. Because of this, the basic traversal functionality test cases are added to the quick test tier. Suppose that with these test cases, we have achieved 55% statement coverage and 40% predicate coverage. Additional test cases are added to increase the coverage until the time limit of thirty minutes is reached.

"Full Test" Tier

The middle tier is the full test tier. This is the tier where the functionality is most fully tested.

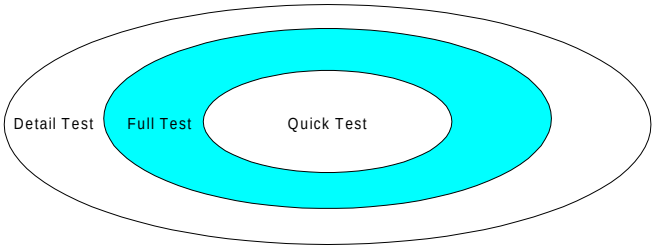


Figure 3: Full test tier

Criteria for Selection

The test cases in the full test tier are an extension of the tests in the quick test tier. The guiding principle in selecting test cases in this tier is to flesh out the tests in the quick test and fill in the functionality gaps. The tests in this tier are the ones most people think of when developing a test suite for a product. The criteria are:

- Add additional tests to fully test all features found in the quick test
- Add functional tests for all other features
- Add tests which must be executed to release the product
- Add test cases for reasonably achievable error conditions
- Add test cases to achieve an "acceptable" level of coverage

The test cases in the quick test only touch on the most common features and uses. To fully test any individual feature, many additional test cases are needed. For example, in a word processing program, the text formatting options that are most commonly used are bold, italic, underline and a small range of point sizes. These would all be tested in the quick test. However, to fully test these formatting options (along with all of the other options available in the same section of the program) additional tests are needed. The test cases are created using all of the common test strategies.

In addition, the quick test ignored much functionality on purpose. The full test tier's goal is to test all of the functionality. Test cases are added to test all of the functionality in the product. The test cases are created using all of the common test strategies.

Requirements for release will change from project to project. Some products will be localized after the initial release for foreign markets. One set of tests that need to be executed before the domestic version is released is the set to ensure that the product can be localized. Also, a contract may specify some needed performance level for the product. The necessary tests to measure this performance need to be included in the full test tier.

Error handling is a fruitful area for defects to hide. Many inexperienced test teams will neglect the large number of test cases that are derived from error handling. On the flip sides, some errors are so unlikely or so difficult to simulate that the value from them is not worth the effort in executing. For the full test tier, test cases are created for all reasonably achievable error conditions. (The other error conditions are delegated to the detail test below.)

Finally, the project may set some level of required coverage for the project as a whole. Since the full test is executed in its entirety before the release of the product, the coverage level required needs to be obtained with a combination of quick and full tests. Hopefully, the test cases generated from the prior criteria will achieve close to the desired coverage and additional test cases will be few.

As in the quick test, contractual obligations may require a set of tests to be executed before release, but are too lengthy to include in the quick tests. A prime example is performance testing. Performance testing before defects are shaken out may mislead the team. Performance gains identified could easily be reversed after more robust error handling is introduced. Also, performance tuning usually occurs late in the development cycle. It is wise to defer execution of the full performance suite until later in the product lifecycle.

In the quick test tier, the goal of the test cases was to identify those defects that prevented testing. In the full test tier, the goal of the test cases is to find all of the defects that prevent the product from shipping. There is a class of defects that the team may deem as acceptable to ship without fixing. Test cases designed to uncover those defects do not belong in the quick or full test tiers.

Application to Other Phases of Development

Once again, these criteria are focused at systems testing. Criteria can easily be developed for the other phases of testing. Criteria for integration testing (assuming the quick test criteria for integration includes the most common interfaces, common data, common errors, etc.) could look like this:

- Test cases for all interfaces
- Plausible call sequences
- Data test cases
- Remaining readily achievable error conditions

Example Application

Using the example utility described in the quick test above, here is an application of the criteria for the full test tier. First, add additional functional tests. This would include adding tests for each of the parameters in the language input file, invoking the help system, syntax testing of command line options, etc. Next, assume that the marketing for the product is promoting it as the "speediest" tool of its kind. Because of this assertion, a set of performance tests is added to the full test tier. Now, consider the error handling for errors not included in the quick test. Some of the test cases derived from these errors are corrupt file, non-termination of a comment, no files in a directory, non-text file, character set other than the default, low memory, etc. Finally, suppose the team has a goal of 80% statement and predicate coverage. Test cases are added to achieve that goal.

"Detail Test" Tier

The final tier is the detail test tier. This tier generally begins with few test cases. As the project progresses, test cases are added which do not meet the criteria for the other tiers.

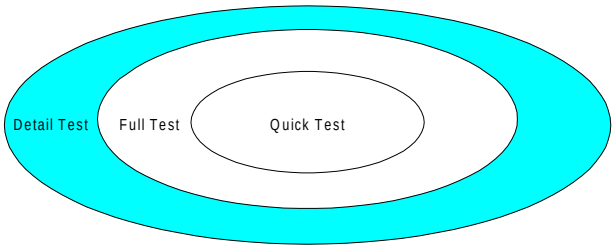


Figure 4: Detail Test Tier

Criteria for Selection

The criteria for the detail test can easily be summed up as "everything else". However, a more orderly approach to adding test cases to the detail test tier is accomplished using these criteria:

- All test cases already developed which do not meet the criteria for the quick test and full test tiers
- By reviewing defects and identifying new test cases which need to be created. These new test cases might belong to one of the first two tiers. However, many test cases will be too specialized and have a place in the detailed test tier.
- Error conditions which are extremely unlikely or not worth the effort to simulate. A judgement must be made as to whether a defect in the error handling is worth the risk of not including it in the other two tiers.
- Additional tests to further improve test coverage
- Additional stress and performance tests not required for the release of the product

These criteria are fairly self-explanatory. Other test cases may also be included. Likely candidates are tests that only need to be executed once. It is at the discretion of the project team as to where the tests are included. The full test tier cases will probably be executed multiple times whereas tests within the detail test tier may be discretionary.

The characterization of the test is twofold. First, test those cases likely to find defects that may go unfixed. Second, test those cases whose expense to execute is higher than the benefit derived from finding a defect. Within the tier, a ranking of test cases could be implied. Some test cases could be identified as more desirable to run then others.

Application to Other Phases of Development

As above, these criteria were developed with systems testing in mind. For other stages of the lifecycle, these same criteria can be applied with few changes. The concept of the detail test containing those test cases with the least return on execution investment applies during all stages of development. However, the size of the detail test tier is much smaller for the other phases. This is due to the fact that test cases are generally easier to execute before full systems testing.

Example Application

Once again, consider the following test cases from the example utility discussed in the quick test section. First, add all of the test cases developed that do not fit into the other tiers. The cases include detailed output tests of the formatting, traversing large directory structures across machines, etc. Now, examine any reported defects that did not result from the execution of a test case. If the new test case does not apply to the other tier, insert it into this tier. Common examples of these are defects that appear only on one customer's machine. Now add, the error handling cases that are not easily invoked. An example is a low memory condition at a specific point in the code. Finally, additional test cases may be added to further increase coverage.

Special Types of Testing

There are several types of testing that generate test cases that do not neatly fit into the three tier criteria. One way to handle these test cases is to generate additional tiers (or perhaps a wedge within a tier) for the special type of testing (see the flexibility section below). Another way is to scatter the test cases throughout the tiers. This was alluded to in the criteria above. Here are some suggestions on how to classify the test cases generated for compatibility, configuration, stress and performance testing.

Compatibility Testing

When considering how to classify compatibility tests, the project must decide what the compatibility priorities are for the product. The structure I prefer is to generate "A" and "B" compatibility lists. The product must meet items on the “A” list. Items on the "B" list are opportunistic goals.

Since the "B" list items are not required for the product to ship, those test cases generated from the "B" list can automatically be relegated to the detail test tier. Once work has been done to achieve compatibility, the project may decide to promote the particular test cases to full test tier.

What about the test cases generated from the "A" list? Start by determining the frequency that a particular compatibility item will be required. For example, compatibility with the file format of a previous version is probably a common need. (Besides the fact that users upgrading to the latest version will get their first impressions on the quality of the product by how well it works with the data already generated.) The basic compatibility tests for these features are candidates for the quick test. More detailed tests for the feature are included in the full test. Now consider compatibility with a format that is no longer in wide spread use. The majority of tests will probably be included in the full test tier.

Configuration Testing

One of the hardest aspects of testing a shrink-wrapped system is ensuring that it will work with who knows what hardware available to the buying public. The different approaches to configuration testing are beyond the scope of this paper. However, I would like to touch on one approach to configuration testing and how it fits into the classification scheme.

In the quick test tier, test cases for the minimum and recommended configurations are included. Depending on the size of the target market, common configurations available from the major vendors may need to be included. If specific test cases are needed for other hardware / software combinations, these usually fit into the full test tier. I recommend that a procedure be set up to ensure that the execution of test cases rotates among machines, drivers and operating system revisions. As for the various permutations of possible combinations, if the

team decides it needs to document these, the test cases fit well into the detail test. The reason for this is that the effort required finding a defect associated with a very particular configuration may cost more than the consequences of releasing the product with the defect. (So called "Good Enough" testing (Bach, 1998).)

Stress Testing

Stress testing, seeing how the software behaves under heavy load or in unusual circumstances, may already be covered by existing test cases. Some teams generate test cases, as a matter of course to exercise the software in stressful situations. For many teams, this testing is an additional task.

If there are certain stressful situations where the project must be sure to work, then those test cases should be included in the quick test tier. In most cases, stress testing is begun in the full test tier. The various stress tests can easily be prioritized using the criteria for the three tiers. Situations that are most likely to occur have a higher priority than other situations. These test cases are added to the full test tier. Situations that are extremely unlikely are assigned to the detail test tier. The remaining test cases can be divided up at the team's discretion.

Performance Testing

Measuring the performance of a software system may or may not be a regular testing task. Some teams execute performance tests on a regular basis. Others will execute the tests only during performance tuning. The approaches to classifying the performance test cases will differ with the two approaches.

In the case where performance tests are regularly executed, assigning the test cases to the tiers can be accomplished via the same mechanism as other test cases. For teams who execute performance tests only once or a few times during the development cycle, no test cases will be assigned to the quick test tier. I would tend to place the tests in the detail test tier. This is due to the fact that the full test tier is executed multiple times, most likely before the team is ready for performance results.

Benefits

There are many benefits to this approach. Of course, there are others ways to achieve the same results. However, I believe the following benefits derive very naturally from the three-tier approach.

Development of Release Criteria

The structure of the three tiers lends itself to the development and support of a set of release criteria. Before each phase in the development lifecycle (each milestone), it is useful to define a set of criteria by which the product is judged. Many times, these criteria include minimal levels of testing which must be completed for the phase to be met.

Consider the case where a feature is trying to meet the "alpha" milestone. Besides requirements for feature design and implementation documentation, code complete, etc., a requirement may be that the test cases for the feature are developed and classified among the three tiers. To reach the "beta" milestone, the requirement could be that the current build has passed all of the quick tests and a subset of the full test. To reach the "release candidate" milestone, the build must pass all of the quick and full tests. There could also be a requirement that some subset of the detail test has been executed within the last 30 days.

Test Automation

The three-tier organization provides a roadmap for test automation. The test cases that are executed the most often, thus offering the greatest return on investment, are grouped together in the quick test tier. Since the quick test is not user interface oriented, it avoids the common automation pitfall of testing the interface over the functionality. In many automation projects, the first tests developed are those verifying the details of the user interface. These tests are likely to break as the user interface changes and thus become a maintenance burden. Furthermore, the defects discovered are not likely to prevent further testing or the release of the product. Thus the defects found by a user interface oriented automation effort are equivalent to a detail test failing.

Flexibility

This three-tier structure is meant as a framework. By no means does it suppose to work for all projects. The benefit here is that it is easily adaptable. The three tiers are probably the minimal stratification. Additional tiers can easily be defined to meet the needs of the project. I envision some teams adding additional tiers or wedges of a tier dedicated to the special types of testing discussed above. These tests may reside between the full test tier and the detail test tier indicating that the test cases within are more likely to find a certain class of defects. I could also see some teams creating a "standards" tier. This tier could contain the team coding standards verification, user interface guidelines, etc.

Statistical Sampling

Finally, this organization lends itself to statistical sampling method of selecting test cases. One way to think of the tiers as a distribution of test cases ranging from those most likely to catch an important defect to those least likely. If time does not allow for the entire test suite to be executed, a sampling of test cases could be taken. By executing these test cases, the team has some level of confidence based on the test suite. For example, say a beta release is needed to be released to external customers tomorrow morning. The team could use the following method: Execute the entire quick test (say it takes two hours with the full test team and automation). This will ensure that the 80% of common functionality / error cases are executed. Given an eight-hour workday, this leaves six hours left to test. The full test suite can easily take weeks to execute. One way to select the test cases is a random sampling of test cases from the full test tier that will take five hours to execute. In the last hour of the day, take another random sample of test cases from the detail test tier. If this process is repeated multiple times, the test cases selected during the previous sampling cycle could be excluded from the sample space for the current sampling exercise.

Conclusion

The three-tier organization presented here was developed with the tester in mind. The quick test tier is aimed at finding those defects that prevent further testing. The full test tier discovers those defects that will prevent the project from shipping. The detail test tier finds those defects that would be nice to fix (and many will be fixed before ship). The organization prioritizes the tasks for the test team. This provides the most valuable feedback for the project in a timely fashion.

References

Bach, James. "The Challenge of Good Enough Testing." www.stlabs.com Bellevue, WA: 1998

Beizer, Boris. *Software Testing Techniques, 2nd ed.*. NewYork: Van Nostrand Reinhold, 1990

Encyclopedia of Software Engineering. "Test Documentation" Ed. John J. Marciniak. New York: John Wiley & Sons, 1994

Kan, Stephen. *Metrics and Models in Software Quality Engineering.* Reading, MA: Addison-Wesley Publishing Company, 1995

McConnell, Steve. *Rapid Development: Taming Wild Software Schedules*. Redmond, WA: Microsoft Press, 1996

Musa, John, et al. "The Operational Profile." *Handbook of Software Reliability Engineering*. Ed. Michael R. Lyu. New York: McGraw-Hill, 1996

So You Think You Know Objects?

Ray Lischner
Tempest Software

Abstract

Inheritance is the least important aspect of object-oriented programming (OOP). More important is that the design and implementation focus on the external behavior of the system and of the objects.

About the Author

Ray Lischner teaches Computer Science at Oregon State University, writes books and articles about software, consults on occasion, and avoids anything resembling a real job. His next book, to be published in early 1999, is *Shakespeare for Dummies* (you read that correctly).

Ray Lischner
Tempest Software, Inc.
lisch@tempest-sw.com

Object-oriented programming (OOP) is either a natural, easy-to-learn programming style, or a radically new paradigm shift that is difficult to master. Which version you hear often depends on what someone is trying to sell you. To a certain degree, both are true. OOP is easier than the doom-sayers think it is, but it can require some effort to understand objects fully. This paper helps novice and experienced programmers get inside objects and understand what they are and how to use them.

Start with a test: What is the single most important aspect of an object?

A. Fields (also called instance variables or data members)

B. Inheritance

C. Methods (also called member functions)

D. All of the above

If you are like most people, you picked D. Some people pick B because that's the primary difference between most object-oriented languages and other kinds of languages. The right answer is C. Keep reading to learn why.

What Is an Object?

The most fundamental issue is the nature of objects. What, exactly, is an object, in programming terms? To answer this question, consider another question: What is the difference between a door and a window? You read that correctly, and "window" means a window in a building, not on your video display. Think about it for a couple minutes. It might help to write down five ways you can distinguish a door from a window.

No, really. I'm serious. Write down a list of five ways to tell the difference between a door and a window. I'll wait.

Have your list now? Good. Look at the items on your list. Can you find any similarities between the items? Try to organize the items into a small number of categories. What categories have your listed? For example, your list might say that windows tend to be smaller than doors. Put this in the category of *physical attributes*. If you wrote that you can always see through windows, but only sometimes through doors, that, too, is a physical attribute. Maybe your list describes windows as something you look through, and doors are something you walk through. That is a difference of *usage*.

Everyone comes up with their own lists of differences and a variety of categories. Nonetheless, two categories dominate the lists: attributes and usage. Most other categories can fit under these two categories.

Now decide which category is most important. Which one best describes the most basic differences between doors and windows? If you chose usage, you understand the fundamental nature of objects.

An object is defined by how a program uses the object. Another way to put it is to say that an object is a collection of behaviors. That is, an object has a set of behaviors, or things it can do, and the list of things it can do tells us what the object is.

The physical attributes of an object are also important, but they do not define the nature of the object. Given a collection of physical attributes, such as size, transparency, and location, you know nothing about the object. It might be a door or a window. The usage is the only way to tell the difference. That's why usage is most important, more important than attributes. Attributes, on the other hand, differentiate between distinct occurrences of a particular kind of object. This door is big, and that door is small. The next section explores this concept.

Classes and Objects

Upon closer examination, you can see that the behavior of an object applies to all similar objects. For example, our use of doors and windows differentiates the class of things called "doors" from the class of things called "windows."

A person can open a door, possibly see through a door, enter a door, and so on. This use of doors applies to every door. A person can also open a window, usually see through a window, etc., so windows and doors are similar. Entering a building through a window, however, is not part of the normal behavior of a window (or a person, except for that subclass of person, burglar).

With this understanding, we can be more specific about the nature of objects. The way we use objects applies to the entire class of similar objects. Look at this the other way around: the class to which an object belongs determines

how to use the object. If someone tells you it's a door, you know you can walk through it. Perhaps object-oriented programming should be called class-oriented programming.

Thus, the class determines how to use an object. What, then, is an object? Consider what differentiates one door from another. This is when physical attributes are important. One door is red and another is blue. One is here, another is there. You can use all doors the same way, but each individual door has its own particular attributes.

An object, therefore, combines the behavior of its class with a particular set of attributes. When I say "door" you immediately think of the entire class of doors, as characterized by their behavior. When I point to a particular door, you can associate that specific door with the class of doors, which tells you how you can use the door, and that it's a door, not a window. You can also interact with the specific door, which can have a knob on the right or the left, have its locked attribute be true or false, and so on. Some people refer to the attributes of the door as the door's *state*.

Defining Objects

Now that you understand the nature of objects, the next step is to define objects in a specific language. Talking about doors and windows is one thing, implementing the *Door* class and the *Window* class is a different matter.

Remember that the behavior is the important part of an object, so you should start by defining the methods. The fields will come later. Begin by determining which public methods you need. List the actions that your object must perform. Include anything that other objects can or must do to or with the object you are designing. These are the behaviors that define your object, or more precisely, your class. (Remember that the behaviors apply to all objects of the same class.)

The list of actions or behaviors gives you the public methods for your class. Some actions are actually interactions with other objects. These other objects might be the callers of your object's methods. For example, a person can open a door, so a *Person* object can call the *open* method of a *Door* object. Or the other objects might be arguments to your object's methods, as in the case of the *rekey(Lock lock)* method, which changes a *Door*'s lock to match the argument *lock*.

Don't worry too much about the methods' arguments. Write down the arguments you know about, and as you work, you will add any other arguments you discover later that you need. The same goes for the return types. You should know the return types for most of the methods, but you might be fuzzy about some of them. That's fine. It's still early in the design stage, so you don't want to get hung up on details, at least not yet.

Begin with the public methods, which define the external view of the object. When thinking about doors and windows, the important aspects of their usage were how others interacted with them. The internal mechanism of a door's hinge or lock is not important, except when it affects the user. The public methods correspond to the external usage of the object.

The next step is to implement the public methods. Follow your favorite programming methodology. Use pseudocode, a program design language, or whatever else strikes your fancy. Invent new, internal methods as you see fit. Perhaps you find some internal actions that many methods duplicate. If so, pull that common behavior into an internal method, and call the method from your public methods.

Internal methods can be private, protected, or use package-level access (the default if you don't specify public, private, or protected). Which access modifier to use is a tricky subject, beyond the scope of this article. As a rough guideline, use protected whenever possible. Use private only when necessary for security or to ensure data integrity. Use package-level access for methods that should be private, but need to cooperate with other classes. Make sure you define all the cooperating classes in the same package. Instead of using package-level access, try to use inner classes instead, another important topic that is beyond the scope of this paper.

When you implement the methods, do not access any fields. Instead, invent accessor methods for the information you need. For example, say you are implementing the *open* method. This method must first test whether the door is locked, so invent the *isLocked* method, which returns a *boolean*. Without knowing anything else about *isLocked*, you can finish the *open* method. When you implement *isLocked*, you might decide that the locked status should be stored as an attribute of the *Door* object. Do not invent a *locked* field at this time. Instead, use accessor methods: *boolean getLocked()* and *void setLocked(boolean is_locked)*.

When you are finished defining all the methods, external and internal, look at the accessor methods you invented. Now is the time to define fields for these methods. By waiting until the last possible moment to define your fields, you give yourself the greatest flexibility. For example, your might discover that your class has many Boolean accessors, so you decide to use a bitmask to implement the various Boolean flags. If, at a later date, performance becomes a problem, you can change the bitmask to separate *boolean* variables. This is an easy change if you use accessor methods, but difficult if your internal methods access fields directly.

Remember that the fields serve only to support the methods. Do not be seduced by the dark side of the source. Many novice programmers are tempted to define fields early, based on knowledge of the real world object. You might think, "a physical door has a height and width, so the *Door* class should define *height* and *width* fields." Maybe. Maybe not. Write your methods in terms of accessor methods (*getWidth*, *setHeight*, etc.). At the last possible moment, implement the accessor methods. At that time, you will know whether you need fields for the height and width. Perhaps you will store this information in a single field, *size*, which might be an array that contains the height and width. Don't limit your options by choosing fields too early.

Inheritance

This paper has not yet discussed inheritance. Surely inheritance is fundamental to the nature of objects? Well, yes and no. Inheritance is a real time saver and helps reduce the amount of code you must copy from one class to another. On the other hand, remember that the most important aspect of OOP is behavior. You don't need inheritance to define objects and classes that implement the behavior you need. Inheritance is just a trick that helps you reuse code.

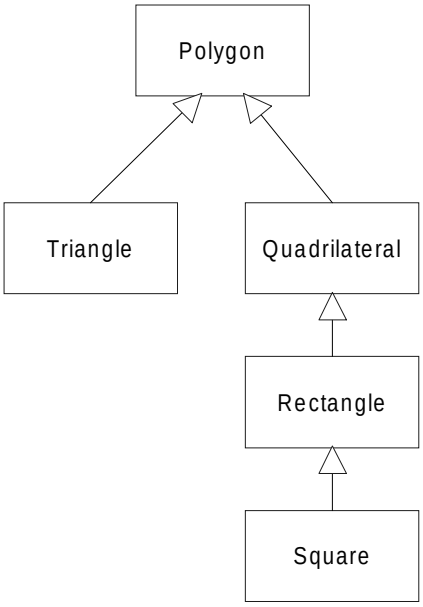


Figure 1: Ideal inheritance tree for simple shapes

The trap that inheritance sets for the unwary programmer is that inheritance in a design is not always the same as inheritance in a program. Textbooks often describe inheritance as the "is-kind-of" or "is-a" relationship. Consider the inheritance tree shown in Figure 1.

A *Square* "is-a" *Rectangle*, which "is-a" *Quadrilateral*, which "is-a" *Polygon*. This is a fine inheritance diagram for a pure OO design. The problem is that it doesn't work well in most OO languages. Consider the same diagram, this time listing each class's fields, which store the lengths of the sides, as you can see in Figure 2.

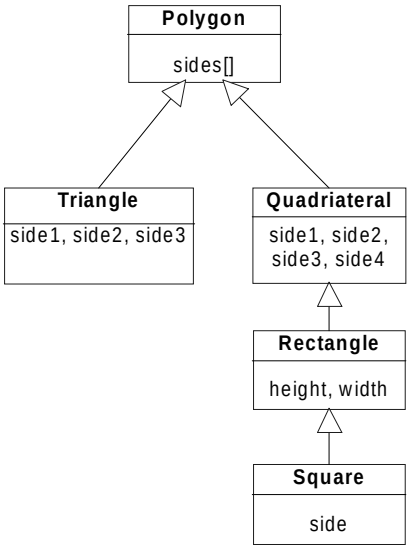


Figure 2: Inheritance tree with fields

Typically, a subclass inherits the fields of the superclass. The problem is that as you descend this inheritance tree, each subclass wants to store less information than its superclass, and you can't do that.

The solution to this dilemma is to discard the pure OO inheritance tree. In most OO languages, inheritance is not really an "is-a" relationship. Instead, it's a "gets-the-fields-and-methods-of" relationship. The "is-a" model is a decent approximation, and it serves many useful purposes, but it can fail. When the "is-a" model fails, don't worry about it. Just discard it and get on with the programming job at hand.

That's not to say you should ignore the inheritance relationships. If you focus exclusively on the

inheritance of fields, you will get a perverse inheritance tree. Figure 3 shows an inheritance tree that optimizes fields inheritance, at the expense of common sense.

The problem with the inheritance tree in Figure 3 is that it does not consider another relationship that a class hierarchy implements, namely, the "is-subtype-of" relationship. Many people have written many papers about subtype *vs*. subclass, and what that means. It's a fascinating topic for the designers of OO programming languages, but is a little abstract for our immediate needs. The reality is that inheritance defines a type hierarchy in most OO languages. This means a *Square* object is type compatible with *Rectangle*, *Quadrilateral*, and *Polygon*. It also means a *Rectangle* is type compatible with a *BiMetricPolygon*, which is type compatible with a *UniMetricPolygon*, but knowing this doesn't make us better programmers.

In other words, a pure OO inheritance tree is a better fit for a type hierarchy, but it does not work well for field and method inheritance. You can change the inheritance tree to work well for field inheritance, but that doesn't make any sense for the type hierarchy.

As you can see, it's a mess. The inheritance tree in Figure 3 is hard to understand, hard to work with, and is illogical. On the other hand, each class has exactly the fields it needs and only those fields. You must compromise to arrive at a meaningful class diagram. How you compromise depends on your particular needs. Figure 4 depicts one possible solution.

This solution keeps the advantages of the type hierarchy and field inheritance. It requires more careful planning than the pure OO model, but it works, unlike the pure model. The essence of the new model is the use of abstract classes, such as *BasePolygon*. The abstract classes define the behavior, and the concrete classes implement the behavior appropriately for the class. Each class inherits only the fields it needs, and you get the benefits of code reuse through inheritance.

In Java, Delphi and other languages, you can use interfaces instead of abstract base classes. Use a hierarchy of interfaces to define a type hierarchy.

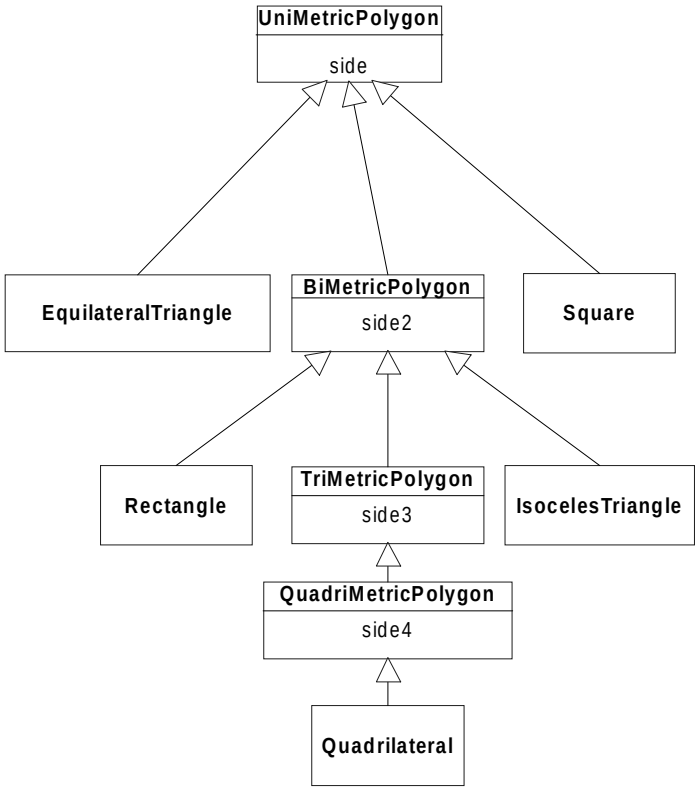


Figure 3: Inheritance tree that optimizes fields

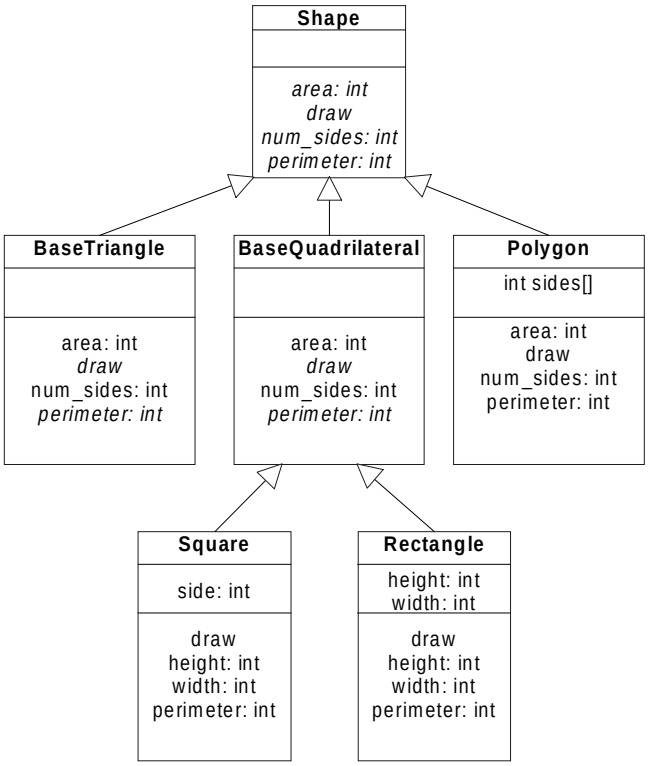


Figure 4: Practical inheritance tree

Define a hierarchy of classes according to your implementation needs, and let each class implement the interfaces that make sense for that class.

Incidental Inheritance

One last problem with inheritance is that inheritance diagrams change. Changing the inheritance tree often changes the methods that a subclass inherits. If your application depends on methods that were once inherited but are no longer, you have a problem. You can minimize this problem by focusing on behavior, not inheritance.

When you think of a class as a set of behaviors, and you document your class in terms of those behaviors, you can use the objects of that class according to the documented behaviors. When you use the objects in your application, you don't usually care about the inheritance, just about the functionality each object offers.

Inheritance matters when you implement a class: you might be able to inherit some methods, override others, and write still others from scratch. When you change the inheritance tree, go back to the original list of desired functionality for your class and see what methods, if any, must be written or modified in the new hierarchy.

The most common cause of problems in a changing inheritance hierarchy is when a subclass inherits methods that it doesn't need, that are not part of its logical, public interface. If you use one of these methods, you increase your risk of problems when the inheritance tree changes. The new inheritance might not provide that method. Because it was not part of the specification for the class, but was merely an incidental inheritance, you have no reason to implement the method under the new hierarchy. No reason, that is, until the other parts of your application cease to compile.

The solution is not to depend on code browsers and similar tools to learn which methods are available. Instead, make sure you write clear, accurate documentation for each class. Use only the documented methods, and do not use any other methods that a class might inherit.

This seems like a strange restriction, but it will save you time and headache in the long run. The corollary to this rule is not to spend too much time trying to define the perfect inheritance tree. Instead, spend your time identifying, defining, and documenting your public methods. If you get the public methods right, you can freely change the internal structure of the classes without affecting how your classes are used.

Inheritance is an implementation trick. Use it when it helps. Avoid it when it doesn't.

Conclusion

The very name of object-oriented programming implies that the programmer should focus on the objects that make up a program. This is misleading. The programmer must instead concentrate on the behavior of the program: identify the actions in the program, not the actors. From the actions (methods), you can determine the appropriate actors (objects). The tendency is to focus too early on the internals of the objects: defining fields, setting up class hierarchies, and forgetting about the fundamental reason for the program: to do something useful, solve a problem, accomplish work.

The proper way to approach OO programming is to focus on the work and the individual actions that make up the work. This will tell you the desired behavior of your objects. Implement the behavior as methods, which are the most important aspect of OOP. Treat data access as a low-level behavior. Only when you have completely defined the behavior should you consider the data implementation.

Inheritance is useful, but a much-overhyped aspect of OOP. The reality is that OO languages do not match the pure OO theory of "is-kind-of" relationships. Instead, think of inheritance as a coding optimization that helps you reuse code. Don't worry too much about the pure, OO nature of your class hierarchy. Just implement what works.

Reengineering the Software Test Process

Northwest Software Quality Conference (October 12-15, 1998)
Oregon Convention Center Portland, Oregon)

Lawrence E. Niech
Fareed Z. Shaikh

Presentation Overview

- Business Challenge
- Technology Challenge
- Business Results
- Project Organization
- Strategy Details
- Test Planning
- Test Automation
- Meeting the Challenges
- Optimizing the Challenge
- Lessons Learned

The Challenge

- Provide customers with a World Class Payroll & Human Resource Management system
- Utilize state-of-the-art PC Windows technologies
- Ensure significantly higher quality
- Reduce time to market
- Achieve higher software ROI

Project Profile - Development

- Target Platforms:
  - Microsoft Windows 3.11, 95, NT
- Development Environment:
  - Centura SQLWindows
  - Borland C++
- Mutilingual (US English, Canadian French)
- Server Database
  - Oracle (multi-user)
  - SQLBase (multi/single user)
- Estimated Code Size : 1.2 mloc
- Number of Windows : 250 + 350
- Database fields : 1800 + 3600

Project Profile - Testing

- Test Plans Developed : 130 + 50
- Number of Test-cases : 58k + 20k
- Number of GUI Test-cases : 45k
- Client Platforms to Certify : 5
- Server Platforms to Certify: 4
- Mainframe Interfaces to Certify: 2
- Team Size : 35
- Estimated QAPartner Code Size : 200k(p), 180k(hr)
- Average Functional Integration Testing Time : 40 days
- Average System Integration Testing : 20 days (4)
- Average Final Acceptance System Test Time : 10 days (2)
- Defect Detection Effectiveness : 97%+ v1.0

Business Results

- Award winning product v1.1
- Satisfied customers
- Low support activity
- SEI 3+ compliant test program
- Defect leakage rates lower than industry norm
  - ✓ 97% detection effectiveness vs. 80% (typical industry c/s) [v1.0]
- Overall test cycle duration reduced by 3x
- Extensible & maintainable deliverables

Business Results

- Enhanced effective and documented test process
- Enhanced staff skills in quality & testing
- Acceptable budget and delivery slippage
- Effective metrics program to gauge pre and post delivery status and provide process improvement feedback
- Fair return on investment
- Proud project team
- A consistent, repeatable, continuously improving process

Formula for Success

The Formula is simple:

+ Focus on the key business initiatives

+ Make the right **investments**

+ Achieve development effectiveness thru **SEI CMM**

+ Demonstrate our personal **commitment** to our C/clients and the IT objectives

= Success

Each element of this formula is required for our success.

Test Group Management/Strategy

- Leverage cross sectional skills from Staff Analysts, Programmers and Automation Experts
- Analyze and improve upon best of breed processes
- Utilize state of the art test tools and techniques to validate GUI technologies
- Maximize unattended test automation to meet stringent delivery deadlines
- Position project group for future technology initiatives

Project Team

- Functional Alignment, Organizational Independence
- PM responsible for bringing together all functional groups

Test Team Structure

PA - STAR

Interactive

- Validate Business Requirements.
- Ensure 100% requirements coverage.
- Ensure GUI compliance to standards.

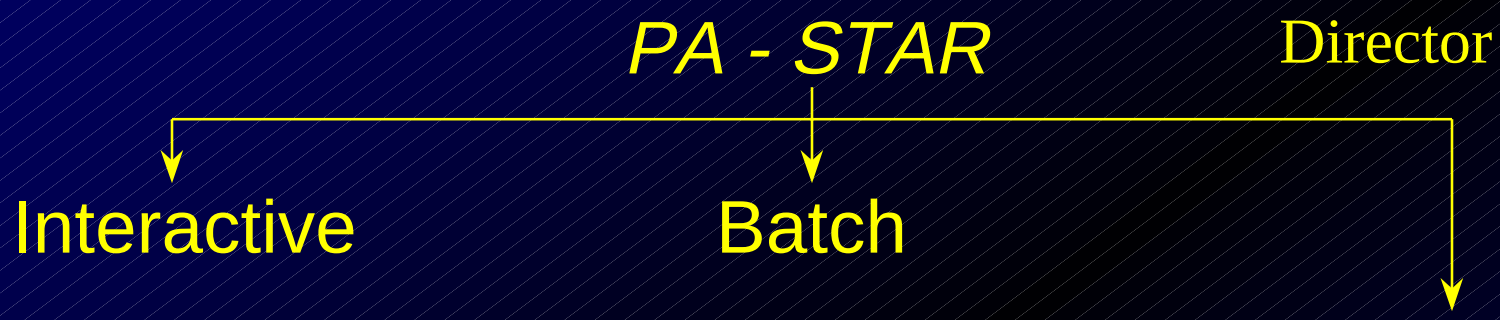
Batch

- Validate Business & Architectural Requirements.
- Ensure 100% requirements coverage.
- Utilize batch functions as test tools.
- Validate database schema.
- Ensure valid field values.

Automation

- Enable aggressive Interactive & Batch schedules.
- Identify & leverage commonality between Functional, System & GUI Testing.
- Boost productivity (unattended testing).
- Optimize test design and execution.

Test Team Structure

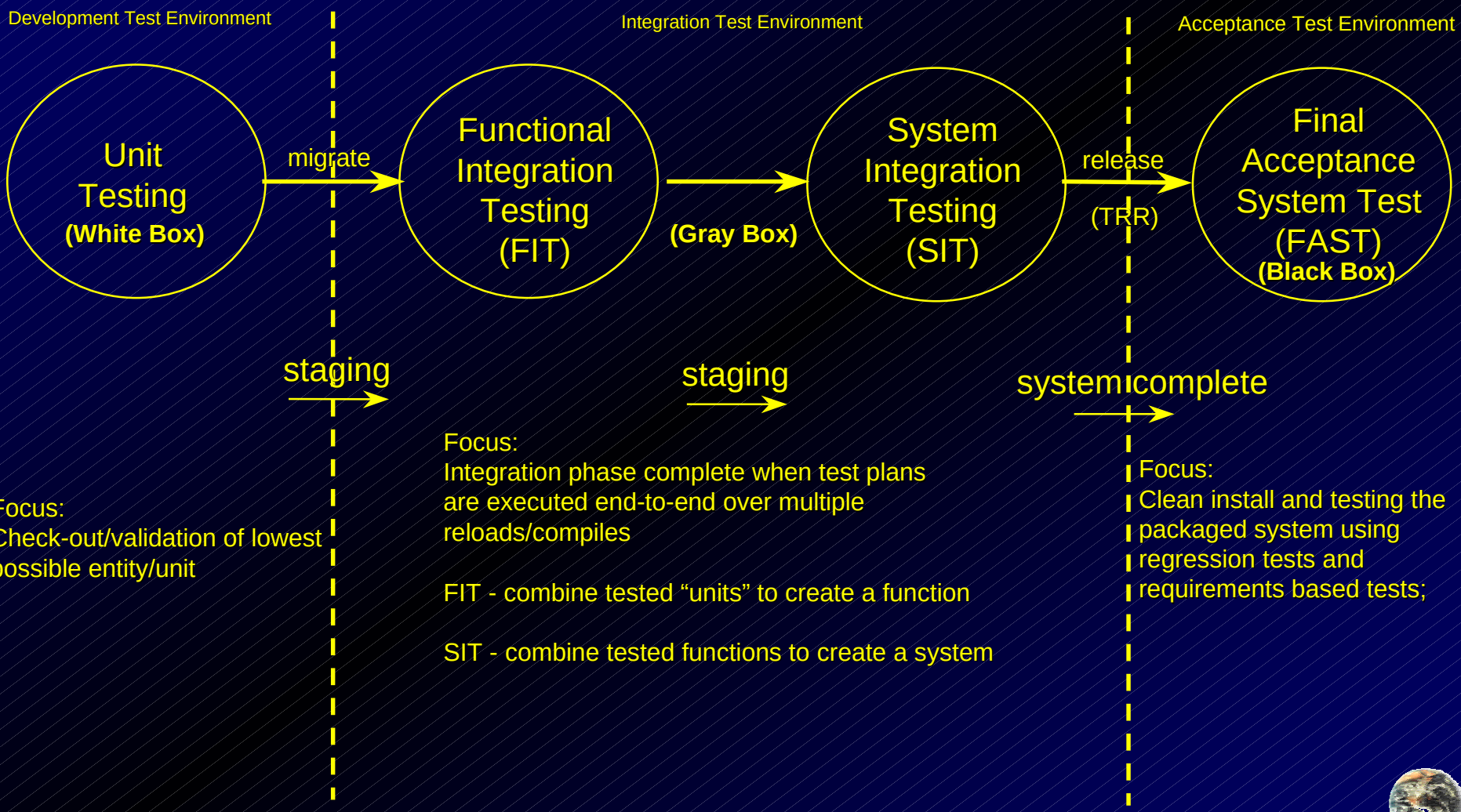


Associates
Consultants
Coops

Development to PA ratio [v1.0]
1 : 1

Strategy Details

Integrated Testing Process



Focus:
Check-out/validation of lowest possible entity/unit

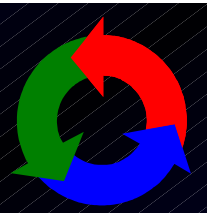
Focus:
Integration phase complete when test plans are executed end-to-end over multiple reloads/compiles

FIT - combine tested "units" to create a function

SIT - combine tested functions to create a system

Focus:
Clean install and testing the packaged system using regression tests and requirements based tests;

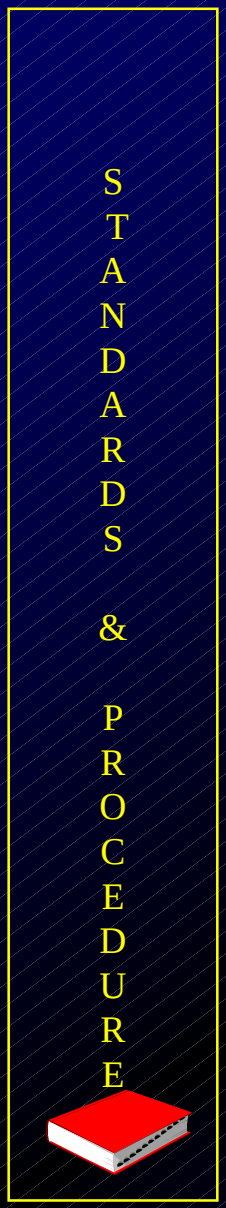
Evolutionary / Iterative Testing
Feature-Oriented Integration Testing

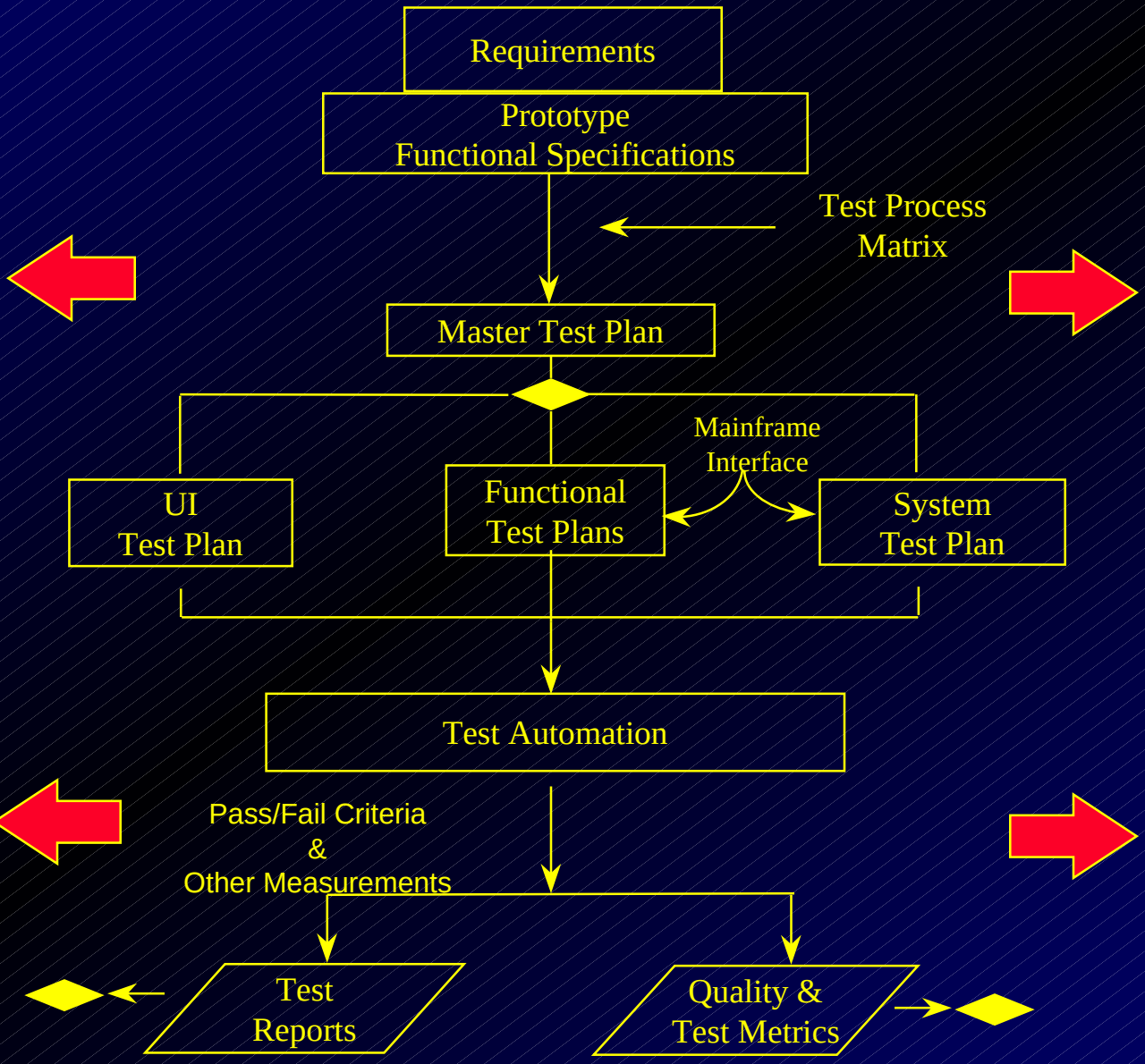


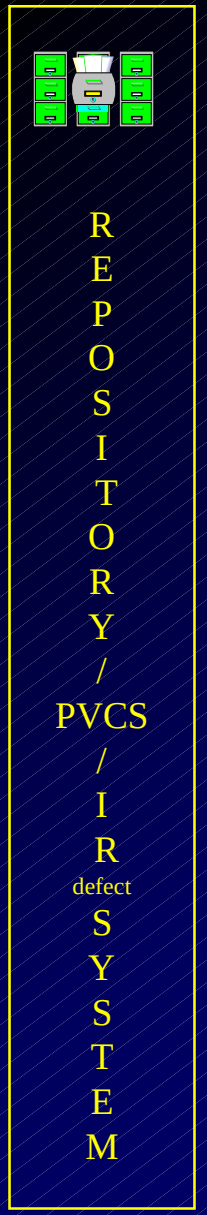
- Required more planning and realistic schedules for all teams
- Increased technical overhead
- Required enhanced communication
- Increased coordination of effort
- Team performance improved with each iteration
- Process assessment/improvement at the end of an iteration
- Improved estimates; schedules/expectations updated each round
- Improved morale - a steady stream of deliverables
- Software stability improved with each iteration

Strategy Details

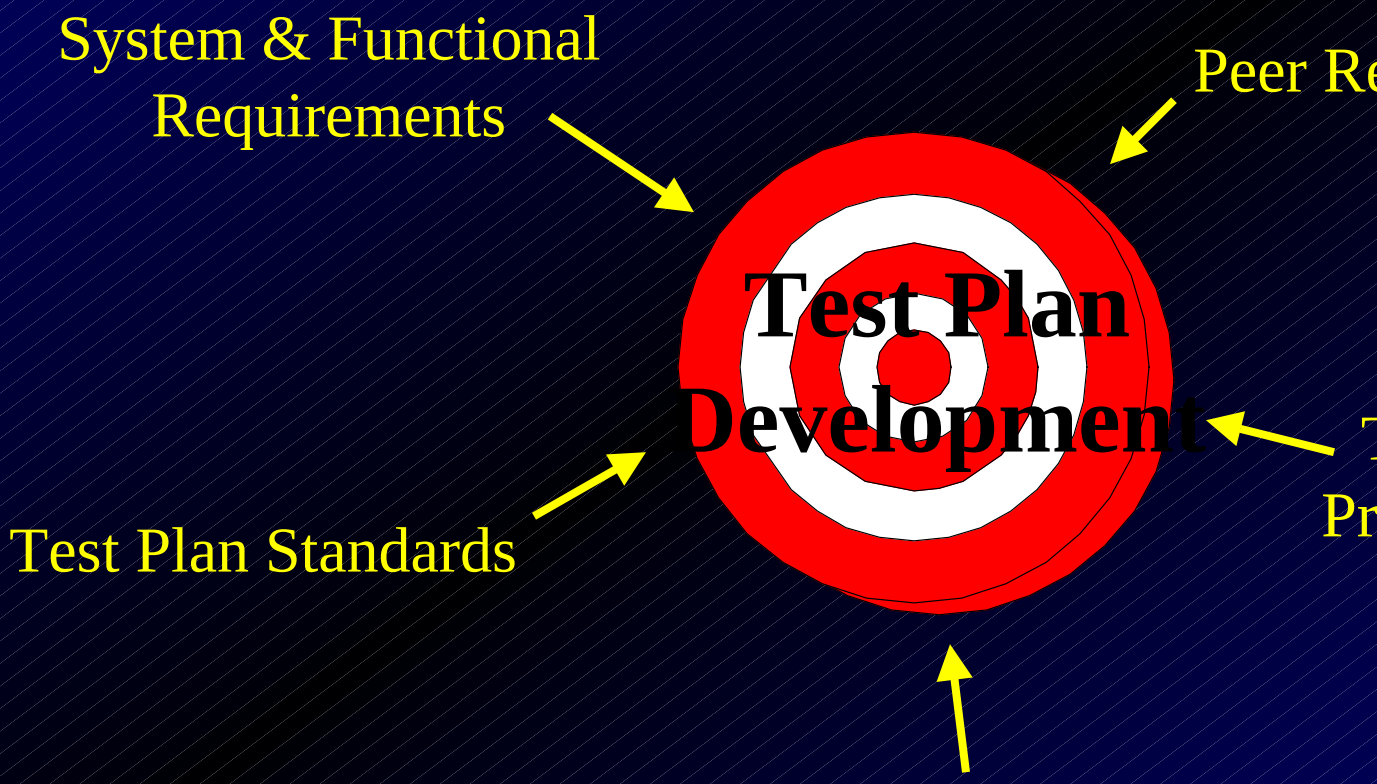
Infrastructure







Test Development



Why we needed to Automate Testing

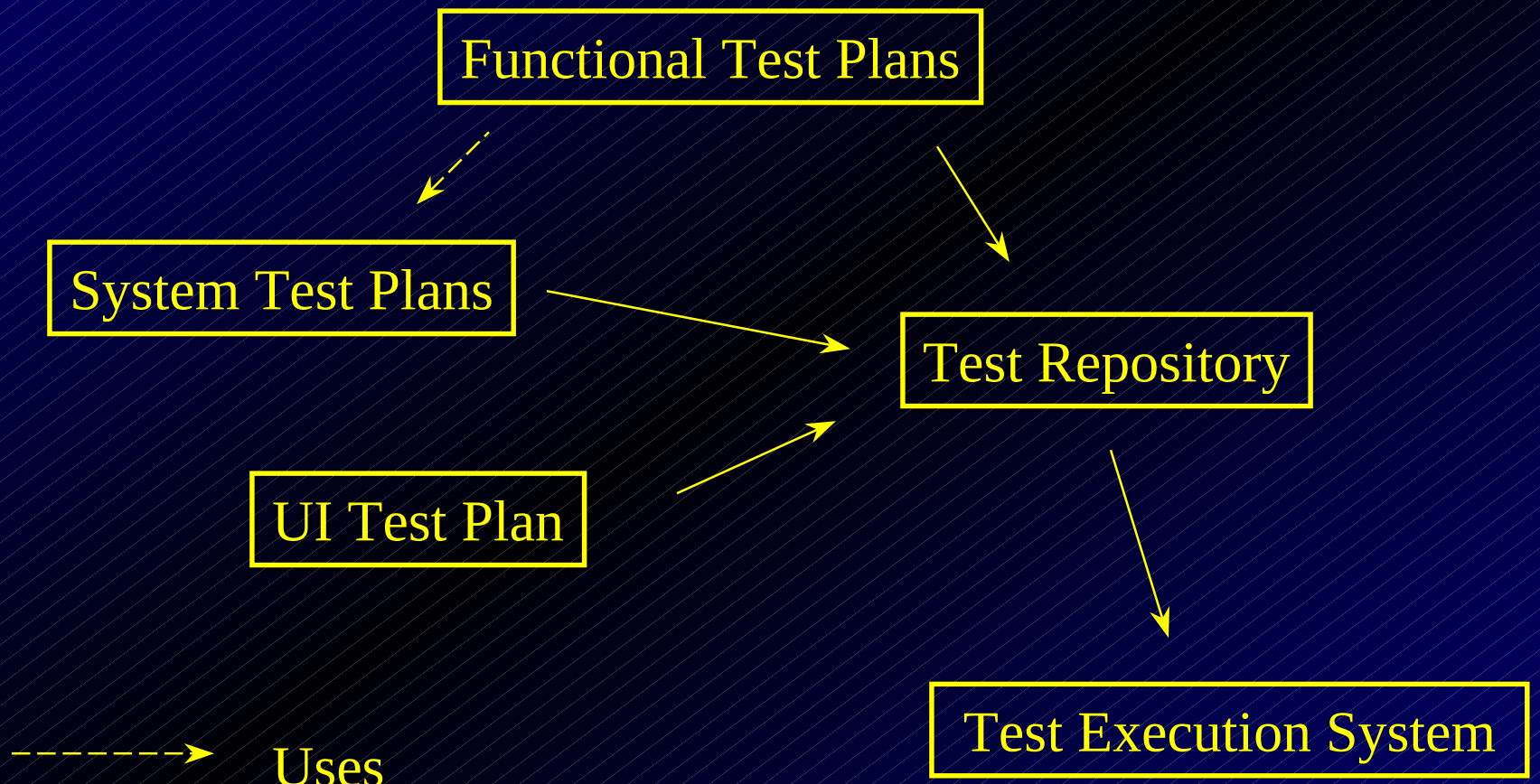
- Volume of regression testing every change/environment would have been staggering if automated testing is not in place and combined with a systematic testing methodology
- To maintain and execute a consistent & repeatable test process
- To more effectively utilize test resources
- Because the “necessary set” of tests to be executed grows with the complexity of the application (functional & UI)
- Because we did not have forever to test
- Reduce test maintenance

- ✓ Test automation is a must to provide a competitive quality edge
- ✓ Automation must be planned and only then implemented

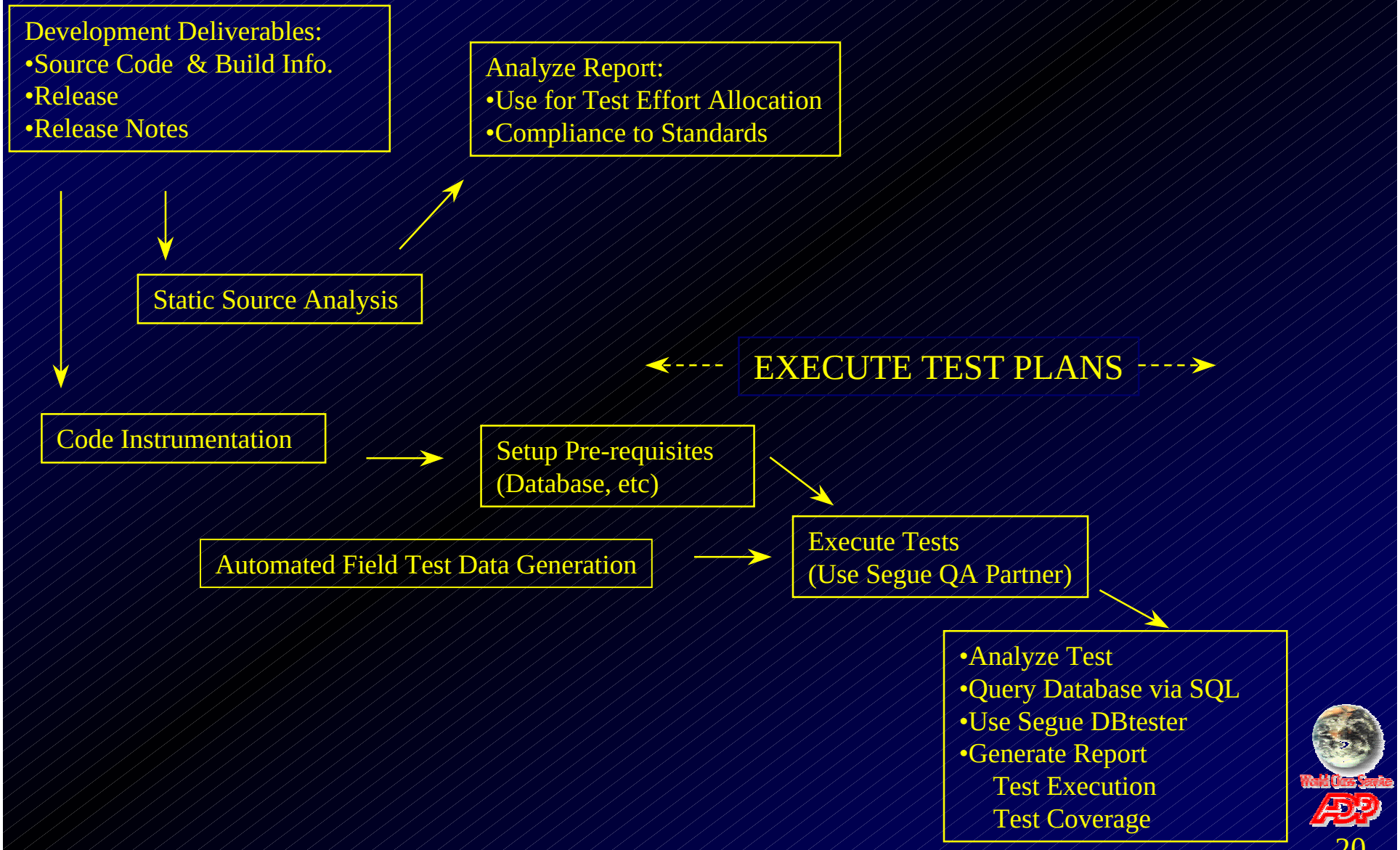
Automation Strategy

- Deploy state-of-the-art GUI & database testing automation tools
- Develop automation standards
- Provide timely training
- Deploy outside expertise for startup
- Aggressively implement automation of system and functional test plans that comply with established standards (test development & automation)
- Continuous process improvement - optimize
- Peer reviews
- Paired business analysts / automation experts

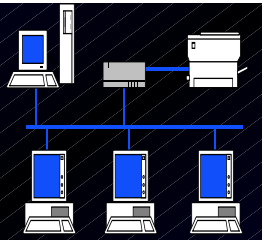
Test Automation Deployment

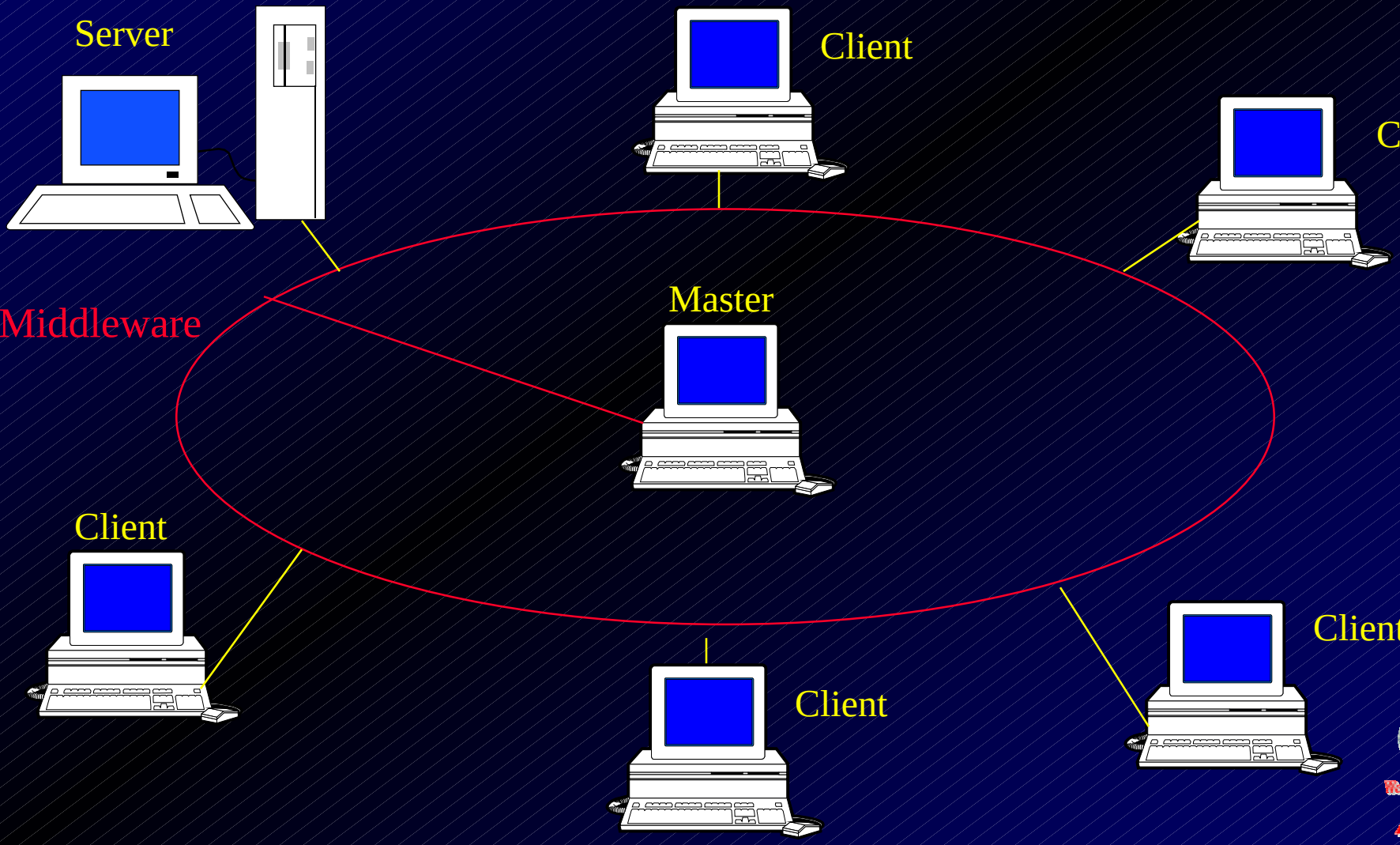


Test Automation Deployment



Automating Multi-user Scenarios





Test Tools Deployed

- Segue QAPartner
- Segue DBTester
- T - Test Generation Tool
- Visual Basic
- C++
- SQL
- MS Project / Powerpoint / Excel / Access
- Defect Tracker

Graphical User Interfaces
The Test Perspective

- Global real estate coverage
- Global consistency issues
- Standards
- On-line help and error message handling
- Window level attributes
- Field level validation
- Other controls

Test Cycles

- Formally planned FIT, SIT, and FAST cycles
- 20 day SIT and 10 day FAST cycles
- Detailed test schedules
- Project group meetings scheduled and also held on demand
- Metrics used to track product progress
- High reliance on oriental and italian food

Test Cycle Schedule Example

Test Plan	DB	Executor	Est Exe Time (hrs)	Exe End Date	Analyst	Analyst End Date
Audit Reports	PR SQL	KK*	8	10/8/96	NC/SC	10/11/96
Audit Reports	PR ORACLE	KK*	8	10/8/96	NC/SC	10/11/96
Auto Calc Rate 2-	PR ORACLE	PK	4	10/7/96	MF	10/10/96
CCDCreate-	HR-PR SQL	JE/AV	24	10/14/96	JE/AV	10/14/96
CCDCreate-	PR ORACLE	JE/AV	24	10/14/96	JE/AV	10/14/96
Change Company-	PR ORACLE	PK	7	10/11/96	CW	10/12/96
CheckMate-	PR ORACLE	MC	24	10/18/96	CV	10/19/96
Company Grouping-	PR ORACLE	PK	1	10/9/96	CW	10/10/96
Company Setup-	PR ORACLE	MC(ZK*)	16	10/17/96	NC	10/18/96
Company Totals-	PR ORACLE	PK	2	10/9/96	CW	10/10/96
Custom Grids Setup (P & E)-	PR ORACLE	PK	4	10/8/96	NC	10/9/96
Date Mapping***-	HR-PR SQL	MC	15	10/14/96	LP	10/17/96
Date Mapping***-	PR ORACLE	MC	8	10/16/96	LP	10/17/96
Employee Changes This Cycle-	PR ORACLE	SK*	56	10/10/96	GW	10/11/96
Employee Changes This Cycle-	HR-PR SQL	SK*	56	10/17/96	GW	10/18/96
Employee Checkview-	PR ORACLE	WB	6	10/11/96	CV	10/12/96
Employee Cume & Adjustments-	PR ORACLE	PK	8	10/10/96	TW	10/11/96
Employee Custom Grid Entry-	PR ORACLE	PK	4	10/14/96	NC	10/15/96
Employee Deductions-	PR ORACLE	PK	10	10/10/96	CV	10/11/96

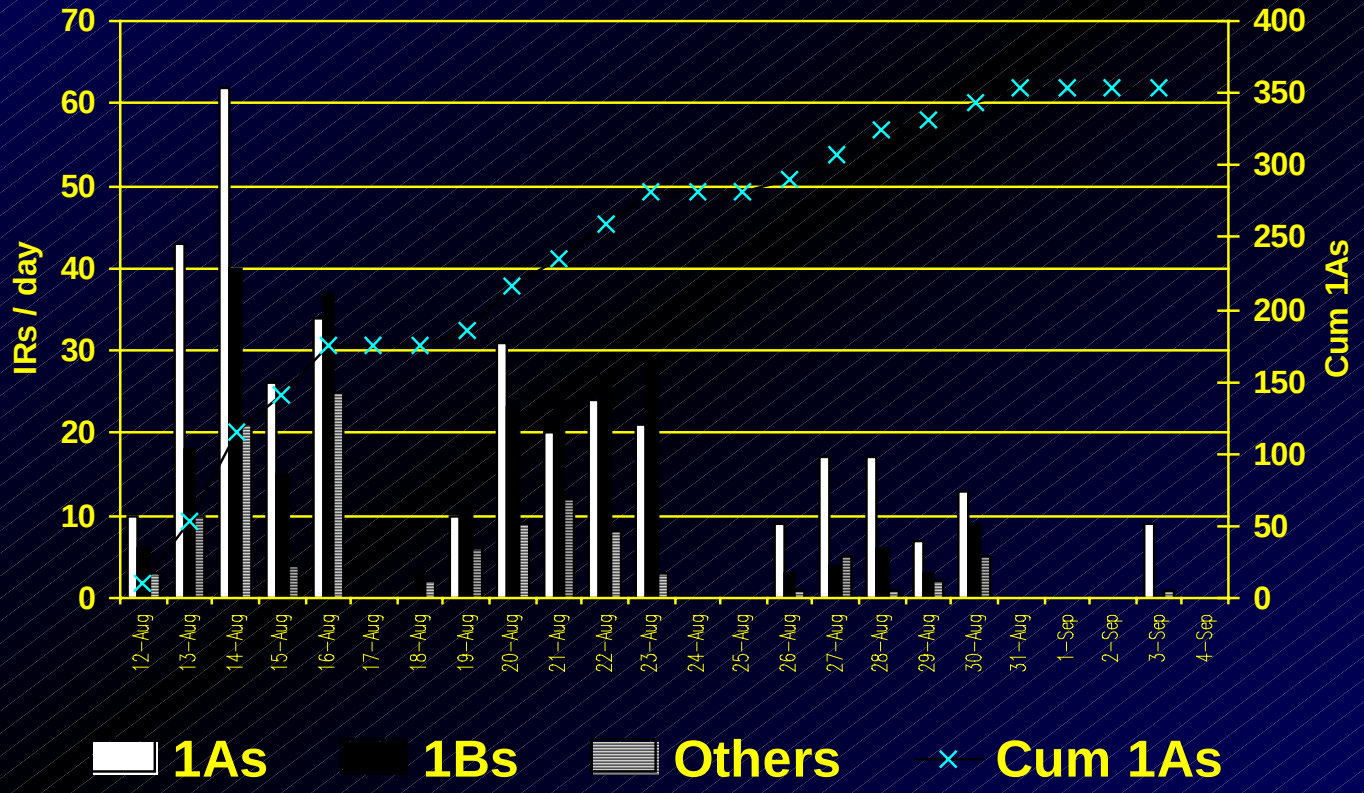
Status Tracking

- Weekly Senior Management meetings to address open issues
- Prioritization based on stability vs business priority
- Test-case metrics used to track test progress
- Defect data formally tracked and reviewed weekly
- Defect data used to determine functional and system stability
- Predictive metrics used to determine expected number of defects

Defect Metrics

System Integration Testing 8/12 to 9/4
as of 9/4, 11:15 Am

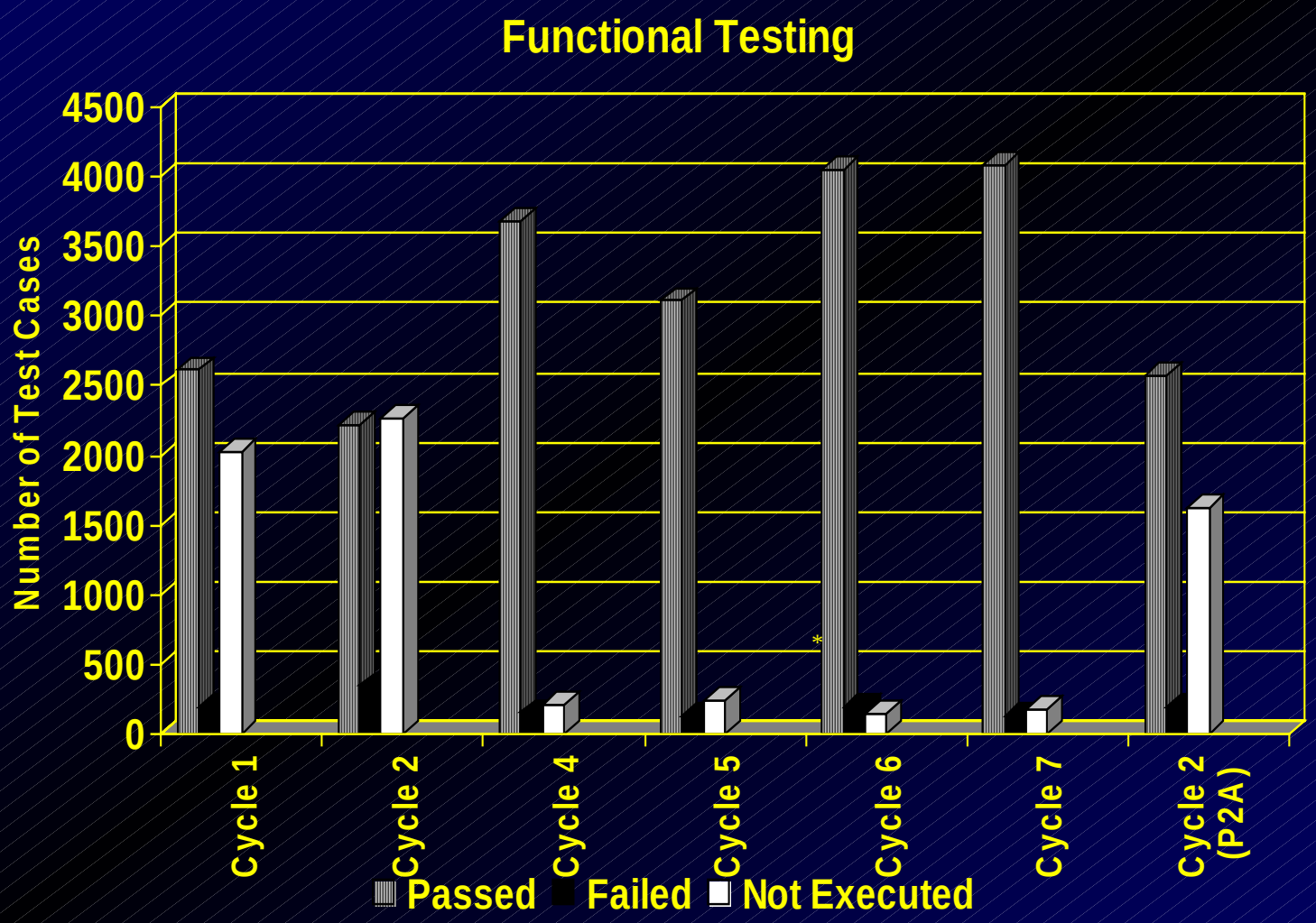
IRs Detected / day



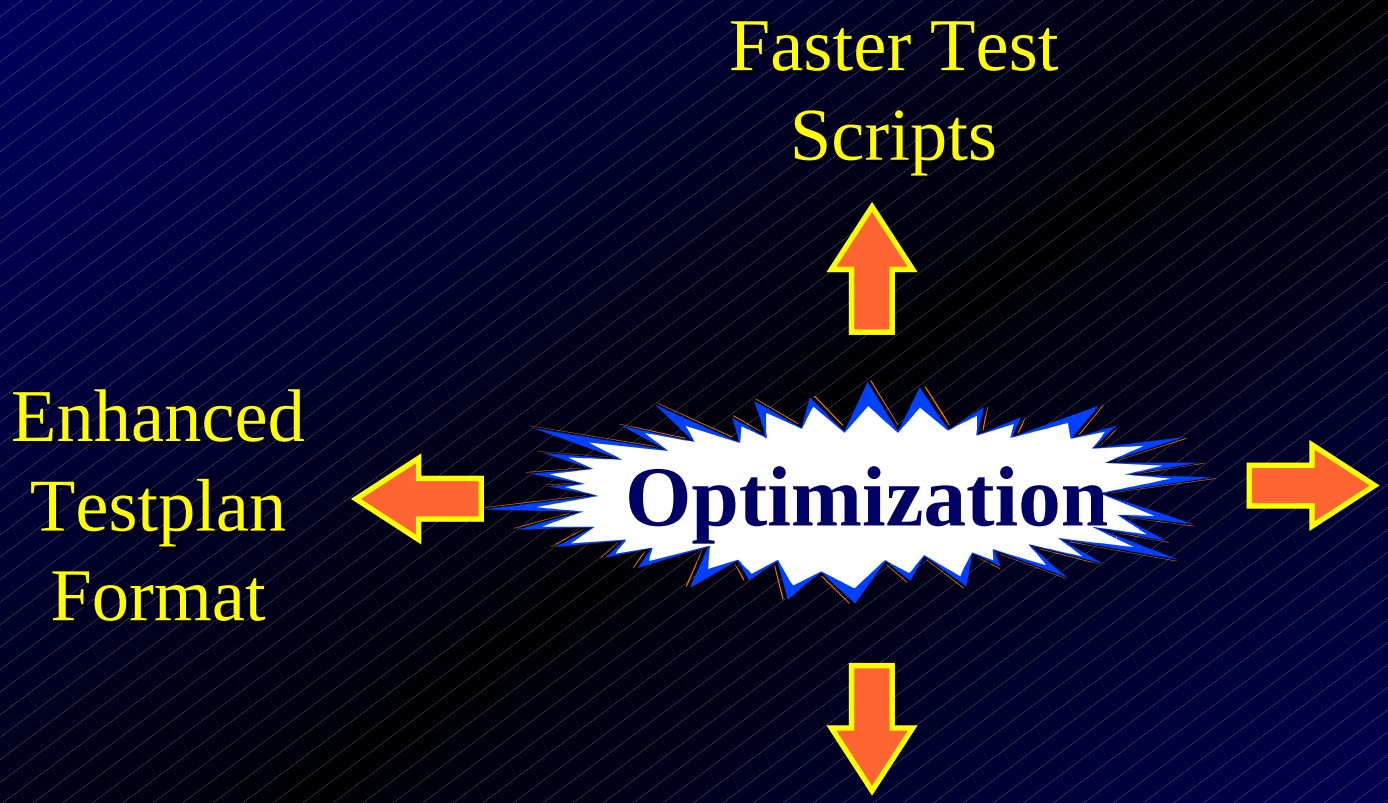
Test-case Metrics

Functional Integration Testing 8/12 to 9/4
as of 9/4, 11:15 Am

Planned: 4392



Execution Optimization



Experiences to Build Upon

- Iterative/Incremental techniques are effective
- Test cases are intellectual assets and should be maintained and reused for similar projects
- Prototype as a “look & feel” specification and also as real code
- Marketing, Development, PA, Documentation, OCS as a virtual team
- Software metrics to manage and predict project performance

Defect Management

Code Coverage

Test Process

Static Code Analysis

Automated Test Case Generation

Metrics Program

Questions?

Larry_Niech@es.adp.com

www.adp.com

Project Postmortem

Test Tools

About the Presenter

Lawrence E. Niech is the Vice President of the IT Product Assurance Department at Automatic Data Processing (ADP) - the largest dedicated software service company in the United States.

His group is responsible for the quality & test programs for all mainframe, PC and client/server products encompassing applications such as payroll, perosnnel & tax processing systems and Sales and Client support tools.

His previous positions have been at ITT Avionics, Singer-Kearfott (Guidance & Navigation Division) and Burroughs Corp. He has played major roles in implementing first time or enhanced Software Quality Assurance and Test Programs as a manager, engineer and consultant.

Mr. Niech has published papers, presented tutorials, taught at the College level and lectured extensively in the areas of software engineering management, software quality assurance, software/hardware test engineering and computer programming. His chapter “Practical Applications of Software Quality Assurance for Commercial Software” is included in the Handbook of Software Quality Assurance (Van Nostrand Rheinhold - 2nd ed).

Mr. Niech holds a MBA from the Rutgers Graduate School of Management and a B.S. in Electrical Engineering from Rutgers, The College of Engineering

About the Presenter

Fareed Z. Shaikh is the Director of PC and mainframe test tools development, measurements program, software release management, and standards and procedures in the IT Product Assurance Department of ADP. He has a strong background in the development of software applications & test tools, and design / implementation of software development / test methodologies.

His interest are in software engineering , test tools, quality metrics, object-oriented analysis & design and Microsoft Windows application development. He has development experience with C++, Microsoft Visual Basic/Office, distributed/internet technologies (COM / DCOM, ActiveX) and several other programming languages.

Mr. Shaikh has presented at international software test automation conferences and taught computer science courses at Monmouth University in New Jersey.

Mr. Shaikh holds a bachelor's degree in electronic engineering from DCET in Pakistan and a master's degree in computer science from Monmouth University.

Experience Report: Comparing an Automated Conformance Test Development Approach With a Traditional Development Approach

Alan Goldfine *goldfine@nist.gov*
Gary Fisher *gary.fisher@nist.gov*
Lynne Rosenthal *lsr@nist.gov*

Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

Abstract

This paper describes a project that investigated the effectiveness of using automated software test generation methods to help develop conformance tests for implementations of specifications of software standards. Traditionally, such conformance tests have been developed by manually coding and testing the test source code. Recently, several technologies that automate parts of the software test development process have appeared. This paper describes a case study that compared the use of a particular automated method (the Assertion Definition Language (ADL), developed by Sun Microsystems, Inc.) to develop conformance tests, compared with the use of traditional methods, when both methods were applied to the same standard software specifications.

Keywords: ADL; Assertion Definition Language; automated testing; conformance testing; software testing.

1. Background

The Information Technology Laboratory (ITL) of the National Institute of Standards and Technology (NIST) has a major responsibility to provide technical leadership for the development of conformance tests for implementations of specifications of software standards. Conformance testing is normally done through falsification testing, where an implementation that claims conformance to a specification is tested with various combinations of legal and illegal inputs, and the resulting output is compared with the "expected results." Traditionally, the test developer, after a detailed examination and analysis of the specification, manually constructs the test requirements, the test code, the inputs, and the expected results. This approach, in particular the coding of the tests and input combinations, is extremely labor-intensive and expensive. NIST has been searching for ways to improve the process.

Recently, several technologies have appeared that attempt to automate parts of the software test development process [1], [2], [3]. In particular, the Assertion Definition Language (ADL), developed by Sun Microsystems, includes a formal assertion language that can describe the behavior of program interfaces, a supporting test data description language, and a translation system that generates C language source code from specifications written in the assertion and test data languages [4]. If the assertions and data descriptions are written appropriately, the generated source code can be viewed as a suite of conformance tests for the given program interface.

2. Project Strategy and Design

Although ADL has begun to be used in several production projects, we know of no earlier studies that have investigated whether or not the use of such an automated technique really does improve the test development process. The project at NIST described in this paper is such a study. We developed, for the same software specification, equivalent sets of conformance tests using a) an automated approach and b) the traditional manual approach. We hoped that by measuring how quickly the conformance tests were developed using each approach, we would shed some light on the effectiveness of the automated approach.

The design goal was to keep the project simple by concentrating only on comparing the effectiveness (that is, speed of development) of the two approaches. Consequently, we limited the scope of the project by choosing as the application a small subset of a standard software specification to which the two approaches could be applied. Additionally, it was clear that some development tasks would be the same regardless of the approach. Since these tasks (for example, determining the requirements or assertions on which to base the test cases) would not affect the desired comparison, we factored them out.

The project can be summarized as follows:

- We investigated existing automated test generation methods, selected one (ADL version 1.1) for use, acquired it, and installed it on existing hardware. We felt that ADL was appropriate for the task, was compatible with available hardware, and was freely available.

- We selected an appropriate subset of a software standard (the C language interface to POSIX [5]) as the application. Of the various specifications that we considered, this collection of POSIX functions best satisfied the selection criteria. In particular, the POSIX standard includes an official list of assertions that, for each function, defines "conformance" [6]. (Appendix A of [7] contains our application selection criteria and the list of applications that we considered.)

- We selected the people who would perform the programming. Essentially, the first two authors of this paper split the task, with one doing ADL work and the other doing traditional coding during the first half of the study, then switching roles for the second half. Neither of us knew ADL before the start of the project; our respective learning curves are reflected in the Results. We both knew how to program in C before the project, although neither of us was a professional C programmer.

- We then developed conformance tests for four POSIX functions (**`chdir`**, **`umask`**, **`rmdir`**, and **`chmod`**, in that order). Part of this development was the validation of the tests by applying them to two candidate implementations, one of which was certifiably POSIX compliant.

- For each appropriate step we recorded the time required to complete that step.

- At the conclusion of the study we compared, according to our measure, the effectiveness of the two approaches.

2.1 Test Development

To help ensure that the strategy and design for our project was sound, we selected a typical function in the application collection, getcwd, and, using the two respective approaches, developed trial run conformance tests for that function. The results of the trial run underscored the importance of focusing on the basic objective of the study, which was to compare the use of two different approaches—essentially the use of two different languages—*to accomplish the **same** programming task*. We were quickly reminded that the specific characteristics of the individual experimenters (as opposed to the characteristics of the automated vs. traditional approaches), would be extraneous to the study and would serve only to skew the results. We therefore strove to design the study to factor out the following characteristics:

- different levels of programming skill, in both ADL and C
- different initial levels of knowledge of the application specification and possible different interpretations of the application specification details
- the potential to develop different design strategies.

In particular, the programmers deliberately worked closely together in all areas other than the actual programming. We carefully selected the sequence in which we would process the functions. We chose, for each function, an explicit subset of the official assertions to test—we excluded from our scope any assertion that corresponded to POSIX functionality that appeared to be unusually complex or tricky (the study was not supposed to be a test of an individual programmer's POSIX knowledge or design ingenuity). We agreed in advance on the precise collection of tests and test data that needed to be developed to validate the assertions for each function. (Appendix B of [7] contains the planned tests and test data for one of the POSIX functions.) Only after all these steps were complete did we begin to implement, using our respective approaches, the jointly developed design. Even during the study proper, we continued to discuss design issues, if these issues were applicable to both the automated and hand-coding approaches and were not directly related to details of the use of the respective languages. (Appendix C of [7] contains both the ADL and C versions of the conformance code written for one of the POSIX functions.)

2.2 Comparing the Two Approaches

Early in the project we developed an initial list of measurement criteria that included various approaches to comparing conformance test development in ADL with development using a traditional approach. In the end, though, the deliberately narrow goal of comparing two approaches to building the same application led us to concentrate on ensuring that the two resulting applications would, in fact, be the same. In this way we eliminated measurements of software quality from our consideration, and simply measured the time that it took each programmer to accomplish the specified work.

We interpreted the time narrowly by only counting the time of the study itself. We didn't count the time spent planning the project, selecting the application, etc. On the other hand, the timing figures included all mistakes, "false starts," etc., in the appropriate categories.

3. Results

Programmer #1 recorded 383 preliminary hours spent learning ADL; programmer #2 recorded 107 hours.

Table 1 summarizes the "comparison" results of the study. The times are given in hours.

Table 1: Time required to write, test, revise and validate the test programs

		Times for the Automated Approach				Times for the Traditional Approach		
POSIX function	Prog-rammer	ADL coding	C coding	Testing of 2nd implementation	Total time	C coding	Testing of 2nd implementation	Total time
chdir	# 1	10 1/2	25	3	38 1/2			
	# 2					31	2	33
umask	# 1	5	9	1	15			
	# 2					6	5	11
rmdir	# 1					13	1	14
	# 2	5	22	1	28			
chmod	# 1					31	1	32
	# 2	19	31	1	51			
TOTAL		39 1/2	87	6	132 1/2	81	9	90

(Appendix D of [7] contains the complete filled-in form, which provided the basis for Table 1.)

4. Observations

Our basic observation is that the use of ADL did not reduce the time needed to develop conformance tests. We can identify several possible reasons for this.

1. A somewhat complicated and non-intuitive tool such as ADL has a significant learning curve. Although we attempted to remove learning time from the direct comparison, even the times recorded for the test itself inevitably included a learning component. If we had continued the study to include additional POSIX functions, a) the ADL coding might well have gotten easier, and b) the included learning time would have been amortized over a longer total time frame.

2. On a related note, one of the advantages claimed for ADL is the ability to build re-usable libraries of symbolically specified and manipulated test data. Our study was limited to four POSIX functions, so we had little opportunity to benefit from re-usability. Had we included more functions, and taken care along the way to build an explicit, consistent library of test data, the ADL development might have been more effective.

3. However, reasons 1 and 2 may be of little overall importance. It turns out that the current ADL system automates only a relatively small part of the total job of developing conformance tests. As can be seen from Table 1, the proportion of work that we did in ADL itself (writing the assertions and the symbolic specifications of the test data

points) was small in comparison to the coding, in traditional C, of the necessary support functions. These support functions included:

- the specification of the actual test data points,
- initializations, file opens and closes, and other housekeeping details, and
- the other auxiliary and utility functions (e.g., the parsing and manipulation of filenames) that were needed to support the evaluation of the assertions.

While the equivalent of the support code would have to be developed anyway during the traditional approach, the magnitude of the C coding did tend to swamp whatever advantages ADL provided.

4. Thus, we come to the reason that we think actually overshadowed all others. We had consistent difficulty with the interplay between the C code generated by ADL and the supporting C code that we wrote ourselves. The final mixed programs crashed frequently and were notoriously difficult to debug. This isn't a criticism of ADL, since the crashes invariably turned out to be due to the user's misunderstanding of the subtleties of ADL. However, the source code for these generated programs was either unavailable to, or unreadable by, the user. While this is part of the design of ADL, and perhaps inherent in the nature of generated code, it definitely highlighted a weakness of the current version of ADL.

5. Conclusions and Future Research

The results of this study show that a reasonable question exists regarding the assumption that automated tools provide a more effective means of developing conformance tests for program interfaces than do traditional approaches. Although our first results are largely negative, a larger scale study is needed to adequately take into account the learning curve and re-usability issues. Other automated tools and techniques need to be investigated, and perhaps more sophisticated metrics developed to better measure test development effectiveness. We look forward to the development of future, more fully automated test tools.

6. References

[1] Chang, J., Richardson, D., and Sankar, S. “Structural Specification-based Testing with ADL.” *Proceedings of the 1996 International Symposium on Software Testing and Analysis*. ACM Press, 1996.

[2] Leathrum, J., and Liburdy, K. “Formal Test Specifications in IEEE POSIX.” *Computer Standards and Interfaces* 17(1995): 603-614.

[3] *IFAD VDM-SL Toolbox* web page: <http://www.ifad.dk/products/toolbox.html>.

[4] Sun Microsystems. *ADL project* web page: <http://www.sunlabs.com/research/adl>.

[5] *ISO/IEC 9945-1: 1990 (E), IEEE Std 1003.1-1990. Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language].* The Institute of Electrical and Electronics Engineers, 1990.

[6] *IEEE Std 2003.1-1992, IEEE Standard for Information Technology — Test Methods for Measuring Conformance to POSIX — Part 1: System Interfaces.* The Institute of Electrical and Electronics Engineers, 1992.

[7] Goldfine, A., Fisher, G., and Rosenthal, L. *Experience Report: Comparing an Automated Conformance Test Development Approach With a Traditional Development Approach*, NISTIR 6114. National Institute of Standards and Technology, April 1998. <http://www.nist.gov/itl/div897/ctg/nm/ir.html>.

Nuts'n Bolts Experiences in Code Coverage Analysis

Pemmaraju S. Rao
p.s.rao@intel.com

Richard Vireday
richard.vireday@intel.com

Performance Tools Group
Intel Corporation, MS EY2-03
5350 N.E. Elam Young Parkway
Hillsboro, OR 97124-5961

Abstract

In our test and evaluation process, Code Coverage is one measure of quality and progress. This paper goes into depth describing the steps we use in Code Coverage measurement and analysis. We discuss our results and experiences with several projects and commercial products spanning over 10 years of development effort at Intel.

We take several pragmatic approaches to our normal Spiral model of development First is the Spiral model itself, which actually makes it easier to add Code Coverage analysis to our overall development process. Further, we follow a philosophy of **Test Black Box, Measure White Box.** We have found this to give the best return on our testing and coverage efforts.

Integration of Tests into the Microsoft development environment has helped make the test process somewhat more manageable, and offers further promise. Together with our philosophy of having developers create and use the Unit Tests, this helps leverage the development effort into testing, and pushes defect detection upstream into the Coding phase of a project.

We find that only Function and Branch metrics, together with simple Defect metrics give timely feedback to the development and management team.

The primary defect number we find useful to track is just the number of Open Defects (defects not resolved). Although other defect measures are tracked, Open Defects (current) has proven to be the most effective in showing trends. This is also consistent with other researchers' findings.

A higher percentage of Code Coverage, and low Open Defect count indicates that the product is tested well enough to ship. These two measures are now first checklist criteria for our products that use Code Coverage.

Key words

Code Coverage, Functional tests, Unit Tests, Branch Coverage, Function Coverage, Visual Basic, C/C++, GUI Testing

Biographies

Pemmaraju S Rao is a Software Engineer with Intel Corporation in the Performance Tools group working on quality assurance and evaluation of software products. He has an MS in Electrical engineering from Georgia Institute of Technology. Previously he worked as Diagnostic software engineer at Digital Equipment Corporation at Colorado Springs.

Richard Vireday is a Senior Software Developer and QA Test Lead with Intel. He has BS from Willamette University, and an MS in Computer Science and Engineering from the Oregon Graduate Institute. He is a member of ACM and IEEE, and has worked in the Intel Design Technology and CPLD/FPGA groups.

Nuts'n Bolts Experiences in Code Coverage Analysis

Pemmaraju S. Rao
p.s.rao@intel.com

Richard Vireday
richard.vireday@intel.com

1 Introduction

Our primary goal with Code Coverage is to insure that our software product is being tested thoroughly. Although the major output of Code Coverage is obviously finding areas of code that are not executed by our tests, we also rely on some of the effects of the Code Coverage process itself.

Code Coverage provides us with a quantitative measurement of our test effort, i.e. a number we can report! Combined with our other efforts in code reviews, coding standards and other verification and testing methods, Code Coverage fills out the details of actual program execution. But Code Coverage only tells how much of our software is tested, not how well it is being tested. So we use the current open defect metric to see how many defects are present. A high percentage of branch coverage and low number of open defects, is one indication that the product is being adequately tested.

The mindset and infrastructure required to do Code Coverage have several benefits. It helps us spread the responsibility for testing and quality on the entire development team. And, it gives the teams feedback and a feeling of reliable progress.

It is commonly held view that techniques like code coverage are academic, and are not useful and feasible in a fast development environment. For example, Watts Humphrey in his book "A Discipline for Software Engineering" **[5]** discusses Program tracing, Unit Tests, Code Reviews and other verification methods, but never discusses Code Coverage as a technique. We believe that Code Coverage should be added to the list.

Management support of Code Coverage goals is crucial for integration of Code Coverage into the development cycle. Without it, development teams who have never done it before, easily shrug off the need. In this paper we want to give a real world example of implementation of code coverage and show how we are benefited by it.

Our current project with Code Coverage has been the Intel Enhanced Debugger (EDB), which is a tool for debugging applications running on IA processors **[1]**. It consists of a collection of components that support common software debug tool functions, and was developed using OLE/COM components and a variety of third party software tools. Excluding the third party tools, EDB is about 100,000+ lines of C++ code. We used Code Coverage while executing the EDB Unit and Integration tests **[2] [3]**. The coverage data is used in enhancing present tests or to add new tests. This is a variation of a process used by the authors on other projects ranging anywhere from 50K to 700K lines of C/C++ code. **[4]**

In this paper we describe the steps involved in our current process for Code Coverage and discuss our experiences with two specific code coverage tools, C-Cover and Visual Test. We make specific recommendations and provide tips for anyone trying to use Code Coverage in their test and evaluation process.

Attempting to exercise each branch condition in every low-level routine through a system level test perspective is difficult if not impossible. In our software development cycle, software testing is carried out with Unit and Integration testing. 100% coverage is not realistically achievable, but an acceptable level of confidence is achievable at a realistic cost in terms of project schedules and quality.

Developers are involved in writing unit tests for the functionality of the code they develop, and these tests are used for Code Coverage testing. The QA/Evaluation goal is to get as much code tested and validated to the extent possible. By using coverage analysis we can actually reach our goals quicker and with better quality.

Our results indicate that Code Coverage analysis does not consume too much time for the project. It can add anywhere from 5 to 15% to the overall effort. But the code produced needs less rework and post-release bug fixing.

Manual testing supplements when automatic tests are not feasible or when they are not yet ready. But as a rule, and for several very good reasons, we do not report manual test results but only the results of the automated tests. We have reduced our manual tests over the last year from over 30 pages of steps, to almost nothing as most of the tests have been automated.

Once a process is set up for Code Coverage, other tools like BoundsChecker, Purify or other tuning tools can benefit from findings of code coverage analysis and a full running Test Suite! Opportunities open up to look at the product under test at several levels of abstraction to track down problems of all kinds.

1.1 Overview of Paper

This paper is organized into the following sections.

- A brief overview of Code Coverage
- Implementation of Code Coverage with C-Cover and then with Visual Coverage
- Differences between C-Cover and Visual Coverage
- Test Suite organization and tips
- Defect Metrics and other useful Statistics used in Conjunction with Code Coverage
- Cultural Changes and Impediments we have run into
- Mistakes, Lessons and Promises we see ahead

1.2 Terminology

We use the following terminology in this paper.

- Unit Test - A test of a component that does not require the full product. Can test an API or Library independently. Preferably runs automatically with no user intervention.
- Integration Test - A test that is done on the full product, packaged and presented, as the user would see it.
- Performance Test - A test that is primarily for measuring the speed of an activity.
- GUI Test - Tests that drive a GUI program. Usually requires mouse clicks and movements.
- SLOC - Source Lines of Code.
- Instrumented Version - a version of the program under test that has Code Coverage instrumentation added to it.

2 Code Coverage Overview

Code Coverage tool watches an application during execution. It tracks unexecuted functions in the code and identifies untested areas of applications under test.

It provides us with statistics

- Line coverage is the numbers of source lines executed.
- **Function** coverage is the number of functions that have been called.
- **Branch** or Edge coverage is the coverage of possible program branches where control is passed from one basic block to another.
- The Visual Coverage tool uses the term Code Coverage to imply coverage of Basic Blocks in the code disregarding the order in which they are executed.
- Call-pair coverage is ensuring that if a function A () is referenced in other functions, that all of the references to function A () are invoked. Hence, each Callee is *paired* with a Caller.
- Dead code is code that is not called by any code.

Function and **Branch** are the only two measures of code coverage we track during product development. Only the code we develop or modify is subjected to Code Coverage analysis. We are not interested in Code Coverage for any third Party or Commercial off-the-shelf code that we may use.

Code Coverage measurement starts with the Test Suites. When a software component is being designed, a set of Unit Tests must be written with a view to exercise all the functionality of library or API. In addition, any Integration Tests are also developed first and foremost to test the important functionality. Later they are enhanced with the results of Code Coverage.

The difference we maintain between Integration and Unit Tests is quite simple. Our Unit Tests are designed from a Black Box perspective to fully exercise interfaces. Our Integration Tests are designed to test the user's view of the product. Since our software components may go into multiple products, the Unit Tests are quite important to provide component level testing.

To measure code coverage, the software product components must be instrumented. The two common techniques are either during the build (pre-processor) or the binary instrumentation (post-process) using the executable (a technique finally reliable on Windows™). Binary Instrumentation requires the product to be built with the debug option. For Code Coverage analysis, initially we used "C-Cover", a pre-process tool. Currently we are using "Visual Coverage", a post-process tool. We discuss our experiences with both of these products.

The block diagram in **Figure 1** illustrates the overall sequence of tasks for code coverage analysis, except for the operational details of using an individual Code Coverage tool.

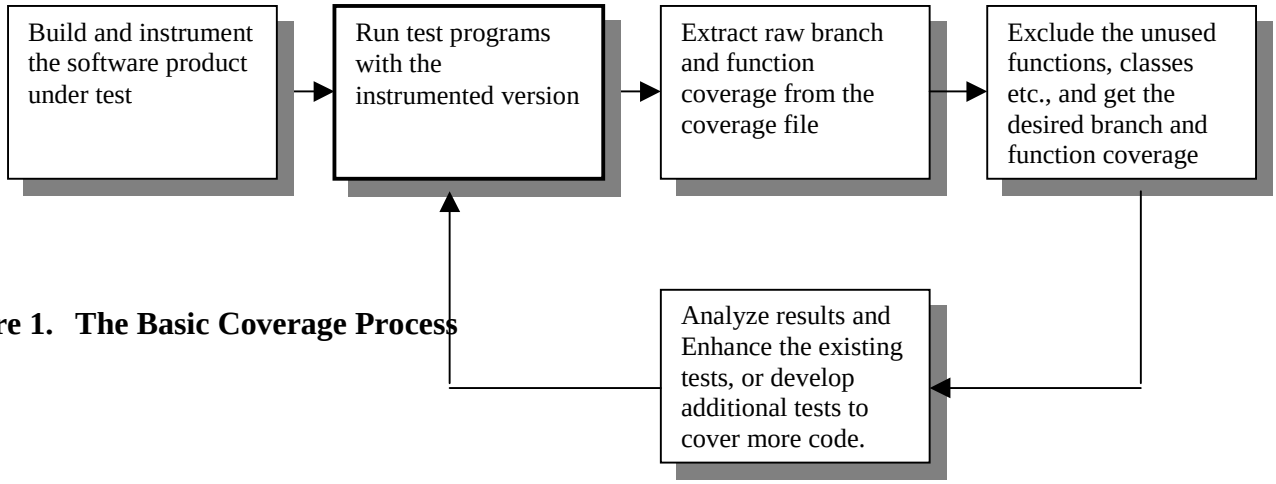


Figure 1. The Basic Coverage Process

The total time for coverage analysis is GUI test time + Unit Test time + Code Coverage time + time to filter data. Once we were into the Coding phases of our projects, we typically did Integration Code Coverage once a week. It took a day to build, test and filter the data. This included time looking for areas to target next or where bugs in the normal development process interfered with Coverage collection.

1.1 Code Coverage Graphs

In Figure 2, we show the normal statistics that we gathered and presented weekly to our staff. (A few weeks are missing while coverage was suspended to work on another project. Notice that the developers did not stop, and coverage decreased during that time!)

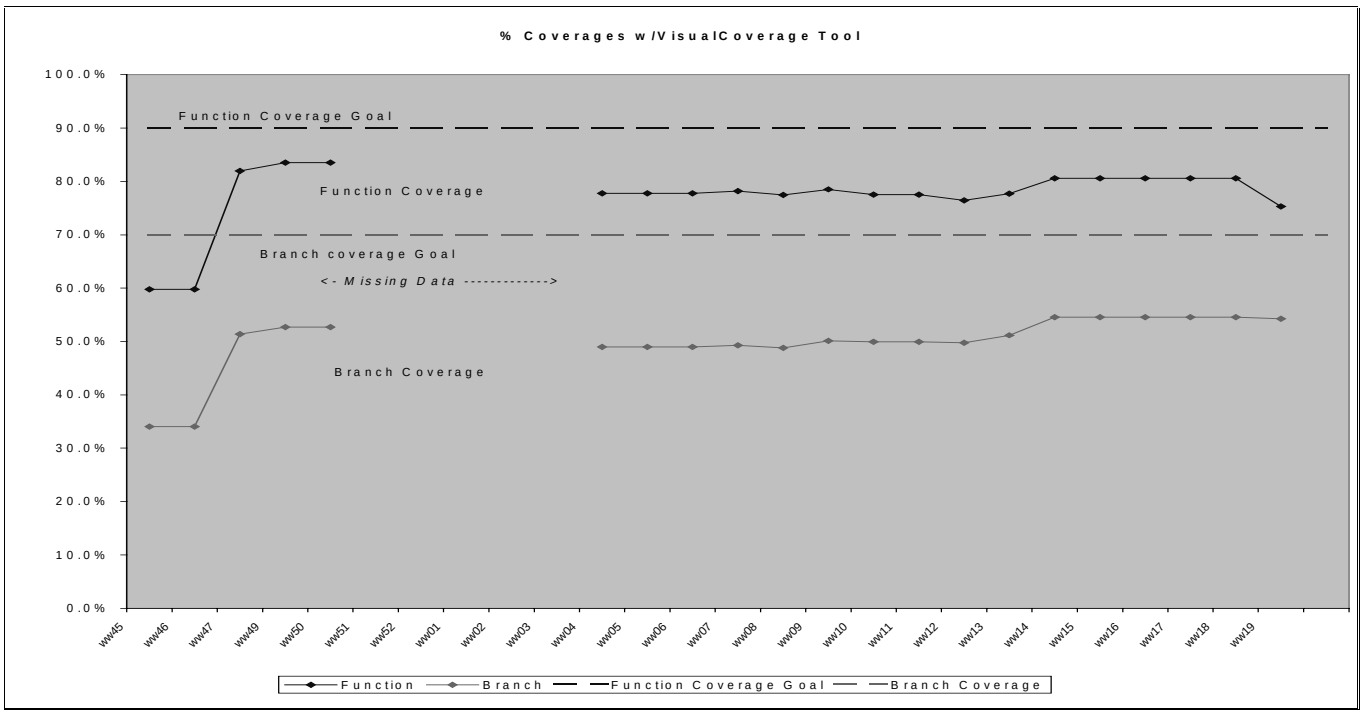


Figure 2. Code Coverage using Visual coverage

While the graph in **Figure 2** is fairly self-explanatory, our team as a whole felt it was unsatisfying. Therefore, we developed the following graph in **Figure 3**. This was much more useful on a number of levels. It not only showed the Code Coverage progress, but also the progress of the entire product.

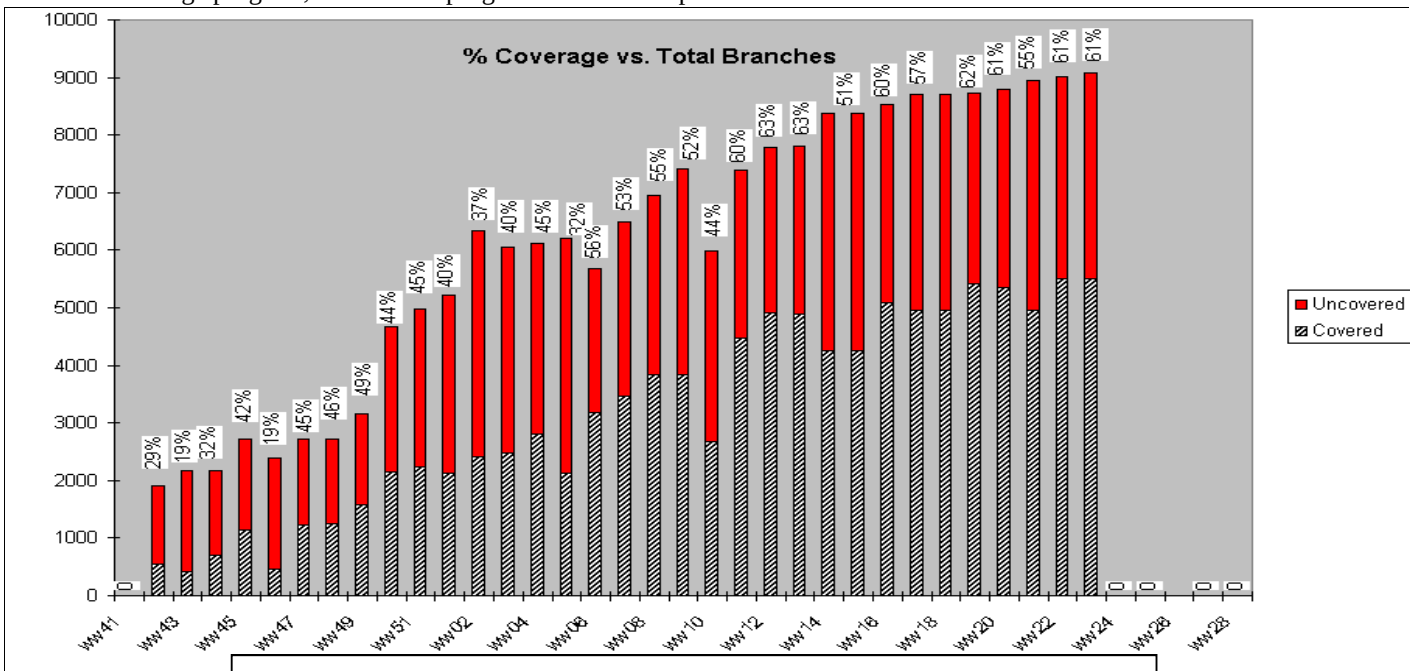


Figure 3: Branch Coverage by workweek (using C-Cover)

Please note that the coverage dips in some weeks were actually the result of our failure to fully instrument the product. In a few instances, the product was reworked and code was removed. Other instances, changes to the build

process or the code required changes in our instrumentation procedures. In one case, we had to get an update to C-Cover to deal with new functionality in MS VC++.

Although the details can be misleading (and sometimes embarrassing), honestly reporting the numbers yielded much more trust by the developers in the use of Code Coverage. The feedback and roundtable discussions of the data enlivened several staff meetings as discussions of cause and effect were explored.

Early on it became clear that the rest of the team did not trust manual testing results. The data from the early efforts varied widely from week to week. So, in order to minimize the problems we decided to never report manual tests. Only the Automated test results would be recorded and reported.

Next we will discuss the two specific tools that we used for Code Coverage.

3 Code Coverage with C-Cover

C-Cover is a code coverage analyzer for C and C++. C-Cover measures function, branch and class coverage. C-Cover is intrusive, automatically adds probes to C, C++ code. Execution time is increased by 3-6X and code and data size is increased 1.5 – 2X. Several reports like branch coverage, function coverage, source file coverage, C++ class coverage, and directory coverage can be generated. We primarily tested Code Coverage using WinNT, Microsoft Visual C++ and MFC, but other operating systems and languages are also supported as well by this tool.

3.1 Code Instrumentation

C-Cover option is activated before Microsoft's Visual Studio (MSDEV) is invoked, and then EDB debugger is built with the normal debug build process. The resulting build is an instrumented version of the EDB product, which is ready for testing. The process of using C-Cover is very straightforward, and we would refer the reader to the C-Cover manual **[7]** for the necessary details.

3.2 Running tests

To generate the Code Coverage numbers, the Automated Unit and Integration tests are run against the instrumented version. Code Coverage data is saved cumulatively to a coverage database for report generation.

Manual tests can also be performed, when required. But, we tend to avoid reporting code coverage due to manual efforts. The reasons are simple: we don't want to have to manually test each time, repeatability becomes a major headache, and the trust of repeating the same steps is obviously lacking. So, we stick with reporting code coverage only with the Automated tests

3.3 Filtering and Report Generation

The coverage report has lots of data to be analyzed and it needs to be presented in a summary form. The need to exclude unused functions and classes is important in the EDB source. This is both to get more accurate results and with over 200 different C++ classes (averaging 8 methods each), the sheer amount of data to wade through is daunting. Here is an example of our C-Cover exclusion list. Note that we exclude not only whole classes, but also sometimes whole files. This is necessary for the reasons listed next to each.

```
-w130                          // 130 column width for output.  Most useful to get actual function
names
C:\reno\dev\+                  // only the development tree, not our 3rd party tools
!IRN*_Proxy                    // Generated by MIDL code.  Part of infrastructure, so coverage not
needed by developers
!IRN*_Stub
!DebugReader::IsFB09
!RNBreakpointTree::            // Unused Functions by the developers. Some are placeholders for
future work
!RNBreakptCombo::
!RNBreakptEdit::
!RNBreakptToolbar::
!RNBreakptWnd::
```

```
…
!RNStateWnd::
!RNTraceWnd::
!operator>>                    // Generated by Roguewave software for EVERY exportable class!
!RN*::Copy                     // Next * items also generated from Roguewave derived C++ classes.
!RN*::classIsA
!RN*::copy
!RN*::newSpecies
!RN*::operator<<
!RN*::operator>>
!rwCreate*
!C:\reno\dev\src\EXPREVAL\cplusplus.cpp   // 3rd party software parser for C++ expressions.
!C:\reno\dev\src\EXPREVAL\CPPParser.cpp      // Not our code, so we don't count it!
!C:\reno\dev\src\EXPREVAL\CPPParser.h
!C:\reno\dev\src\EXPREVAL\DLGLexer.cpp
!C:\reno\dev\src\EXPREVAL\DLGLexer.h
!C:\RENO\dev\src\Internals\MARSHAL\administrator.c   // Marshalling code for DLL's.
Infrastructure again.
!C:\RENO\dev\src\Internals\MARSHAL\breakpoint.c
!C:\RENO\dev\src\Internals\MARSHAL\mediator.c
!C:\RENO\dev\src\Internals\MARSHAL\memory.c
…
```

Figure 3. C-Cover Exclusion list

Extra Perl scripts are used to filter the unused parts of the code using an exclusion list and the results are summarized by class name. These scripts summarize by Class::, which is easier for the developers to scan and look for the pieces they are interested in during large review sessions.

3.4 Tips for effective use of C-Cover

1. Try to define the Code Coverage environment to be the same on all test machines. The same general spot for the Coverage database and location for the test program. This saves tester time by knowing where things should be.
2. Define the search paths so that the Instrumented versions are found first, if they exist. This way mixed Instrumented/non-Instrumented versions can be mixed easily. This is most useful when some DLLs or OCXs can be instrumented and others cannot.
3. Instrument all, then apply the exclusion when generating the reports. This is much easier than trying to specify exclusions during the instrumentation phase. Also, that way you can turn off or on watching various pieces by just modifying the Exclusion list.
4. C-Cover has a floating license, so 2-3 copies in a small group are all that is necessary.

4 Code Coverage with Visual Coverage

Visual Coverage was a tool developed by Tracepoint Technology (now defunct), a spin off from Digital Equipment Corporation. Visual Coverage has a good Graphical User Interface (GUI) to instrument and generate reports for code coverage. Results can be immediately viewed on Browser and html report file can be generated for later use.

An advantage to the Tracepoint Visual Coverage tool over compilation tools such as C-Cover has been 1) no need for a separate build for code coverage. It can just use our normal Debug build. And 2) it can work regardless of the language. This was important to us for our other projects which have VB components.

4.1 Code Instrumentation

Code Instrumentation with Visual Coverage is a post-build event, and the whole process follows these steps.

1. Build EDB (Debug) as normal.
2. Open the MSDEV **Viscov** workspace and instrument the EDB code
3. Run the Test Suites as normal to generate results. Unit and Integration tests are run against the Instrumented version just as was done with C-Cover versions.

4. Open the MSDEV **Viscov** workspace to generate a report with exclusions, or the Tracepoint Visual Coverage window to drill down to specific functions.

4.2 Filtering and Report Generation

Selecting the "update run data" from Visual Coverage Applications menu updates the coverage file with the coverage data from the tests run against the instrumented EDB. Selecting "generate report" from tools menu brings up a dialog box that provides the options to preview the coverage summary, and to generate a report in html or text format.

Visual coverage does not provide an option to filter out any selected functions or classes. Although we may instrument many other components we did not develop, we exclude them later from the report generation. We find it much easier to deal with the report generation after we complete our testing on a completely instrumented product. Trying to keep track of what to instrument or not instrument is difficult especially when the product components and build rules can change drastically.

4.2.1 Visual Cover Exclusion list

Visual Cover did not come with an easily usable Exclusion list like we were used to with C-Cover. The one built into the tool proved cumbersome for the size of the projects we create. So instead we created our own, using C-Cover's as a starting reference point. The exclusion list is reviewed with the developers, so there is no misunderstanding between the tester and developer as to what part of the code is being considered for coverage analysis.

In addition, Visual Coverage instrumented too much of the "hidden" functions in the code, and the Exclusion List was an excellent way to filter out the unneeded code. By "hidden" functions, we mean the compiler infrastructure for things like Stack checking, Type conversions, implicit C++ Constructors and other infrastructure and cruft of modern compilers.

Here is a sample of our exclusion file named "edb.cfg" that is used in filtering the coverage data.

```
# Exclusion List for the Intel Enhanced Debugger
# Name: edb.cfg
# Exclusions for Viscov artifacts

$E*

# Exclusions for Third Party tools and MSDEV functions
*VC*
*Tools*
*DevStudio*

RN*::saveGuts*                  / Exclusion for unused  functions
RN*::restoreGuts*
RN*::binaryStoreSize*

RNDisassemblyList::OnDisCopy   / Disassembly: Copy, Find, Annotate no longer supported
RNDisassemblyList::OnDisFind#

LocalsList.*                    / Exclusion for those not yet implemented
RNClippedFrm::OnViewLocals

RNProcessTree::*                / Not used in current build

LexicalScope::Print             / Exclusions for  Functions not accessible to users
Procedure::Print

Scope::AddModule                 / Exclusions for virtual functions in expreval.cpp
Scope::AddLine

RNBreakpointDlg::Create          / The following were identified as dead code by Visual Coverage
RNBreakpointDlg::GetContext
```

Figure 4. Visual Coverage Exclusion list

4.3 Java Filter for data from Visual Coverage

Vhf.Htmfilter is our Java filter for the raw coverage file "edb.html". The example below shows the command line options for the filter.

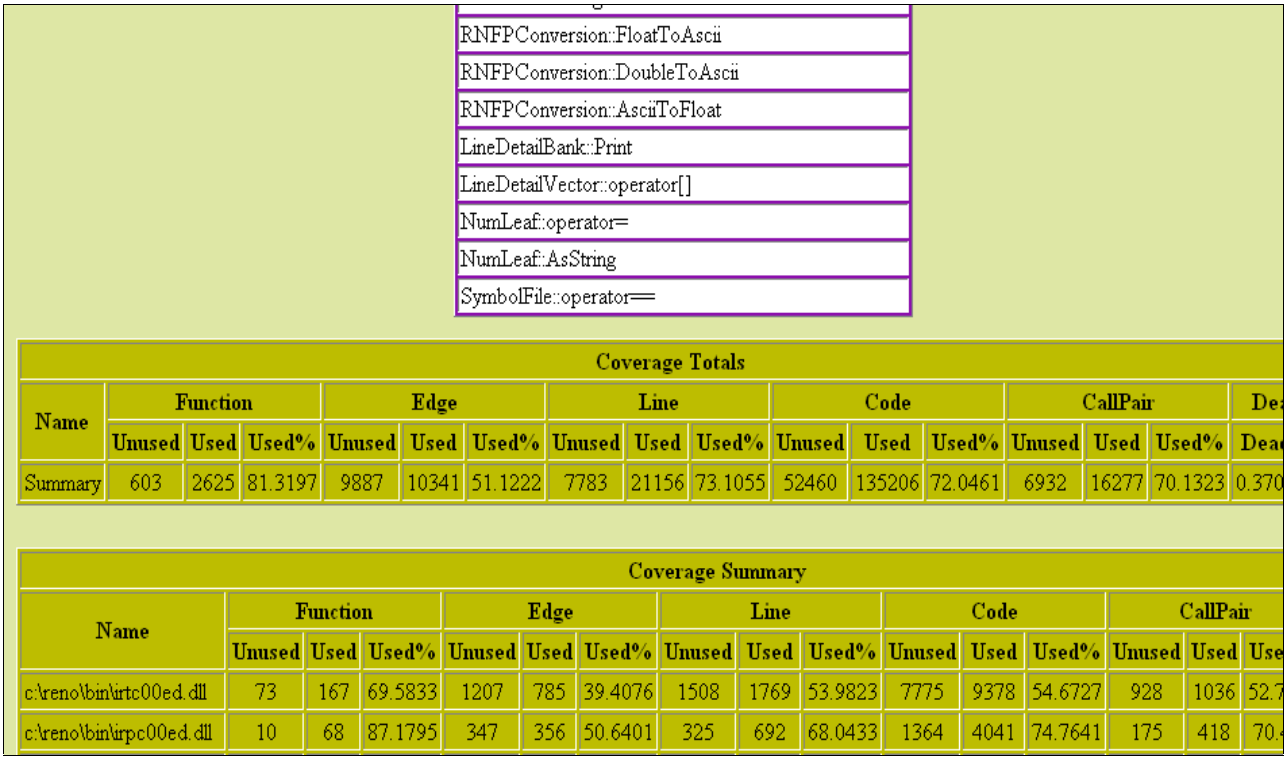
```
jview <filter program> <raw coverage file> <exclusion file> <filtered output>

jview  vhf.HtmlFilter  ..\viscov\edb.html  edb.cfg   edb_0506.html
```

"edb.html" is the raw Visual Coverage file. Applying an exclusion list "edb.cfg" described above, the filtered file is "edb_0506.html" which can be viewed with any HTML Browser, as shown in Figure 5 (although at > 1MB of size, the file does tend to stress current browsers' ability to display the file easily). The untitled portion on the top is the exclusion list. The last column is the dead code coverage.

The Java filter also created the **Summary** section. This was critical as Visual Coverage only provided totals on a DLL or EXE level, and we wanted a rollup for the entire product.

4.4 Metrics Collection Notes

1. Microsoft Excel is used to record the code coverage data and generate the charts. The Excel workbook was updated and checked into our version control database on a weekly basis.
2. The goal set for EDB Alpha phase using Visual Coverage was 80% function coverage and 80% branch coverage. We were able to reach 81% function coverage and only 52 % branch coverage. The reasons for the lower branch coverage are known, and are being addressed.



RNFPConversion::FloatToAscii
RNFPConversion::DoubleToAscii
RNFPConversion::AsciiToFloat
LineDetailBank::Print
LineDetailVector::operator[]
NumLeaf::operator=
NumLeaf::AsString
SymbolFile::operator==

Coverage Totals

Name	Function			Edge			Line			Code			CallPair			De
	Unused	Used	Used%	Unused	Used	Used%	Unused	Used	Used%	Unused	Used	Used%	Unused	Used	Used%	Dea
Summary	603	2625	81.3197	9887	10341	51.1222	7783	21156	73.1055	52460	135206	72.0461	6932	16277	70.1323	0.370

Coverage Summary

Name	Function			Edge			Line			Code			CallPair		
	Unused	Used	Used%	Unused	Used	Used%	Unused	Used	Used%	Unused	Used	Used%	Unused	Used	Use
c:\reno\bin\irtc00ed.dll	73	167	69.5833	1207	785	39.4076	1508	1769	53.9823	7775	9378	54.6727	928	1036	52.7
c:\reno\bin\irpc00ed.dll	10	68	87.1795	347	356	50.6401	325	692	68.0433	1364	4041	74.7641	175	418	70.4

Figure 5. Visual Coverage summary of **edb_0506.html**

5 Differences between C-Cover and Visual Coverage

We must mention here that when we switched tools, the Code Coverage numbers **dropped drastically**! This caused no end of consternation and explanation. The technical reasons were obvious, but it required a little time to explain and recalibrate. It also provided us impetus for getting our Exclusion list for Visual Coverage working sooner than we planned.

As explained earlier, Visual Coverage instruments EVERYTHING in the executable file, so the numbers for Branches and Functions changed simply because the two tools were measuring things differently.

Below is a basic summary of what we feel are the important differences between the two tools.

C-Cover	**Visual Coverage**
There is an exclusion file to exclude a list of functions etc. that are dead or should not be counted.	Feature was difficult to use for our large product. We had to develop our own program to do the filtering.
Post processing is required to get meaningful data from the information generated in the coverage file.	Output is available in graphical form. It also provided a graphical display of code coverage that is helpful in readily identifying the areas of source code that are not being exercised.
All the functionality is command line driven, requiring several steps to get the output in desired form. This was automated with several scripts.	Relatively Easy to use because of it's GUI interface. The output in HTML format enabled us to view the results. The Drill-down features were quite nice. However, it lacked the ability summarize the output across multiple EXE's and DLL's.
Even though EDB could be built, some times the instrumented version could not be built with C-Cover option. Does not work with languages other than C/C++.	Binary Instrumentation was generally successful. A few pieces such as COM Proxy DLLs we were never able to get successfully instrumented. Works (generally) with any compiled Windows executable.

On the whole, we have found C-Cover to be an extremely reliable technology across multiple generations of compilers and our own products. It has continued to stay current, and we will be switching back to it for future Code Coverage work, but maintain our existing Tracepoint tools for those languages that C-Cover does not support.

6 Test Suites

While seemingly off-topic, Test Suites are the keys to accurate and repeatable Code Coverage. A good suite offers a base to build on, and offers other quality benefits besides Code Coverage data.

Test Black Box, **Measure** White Box - this is the philosophy we follow in building all of our tests. We have heard any number of comments over the years from people who think that if you test using Black Box methods, that is only how you should measure as well! Whatever.

With EDB, we kept the tests in a parallel tree structure, so that the tests could be shipped separate from the source base. The implicit justification was that the developers did not want to have to maintain these tests. But, then this also isolated changes and made maintenance more difficult. As developers charged ahead with changes, they never ran the Unit Tests. QA had to run the tests, tell them there were failures, then have them get fixed.

Another model used by other Intel groups was to always run the Unit Tests before checking in changes and enhancements. Tests were in the same tree as the code! This works better, as changes are local and easily accessible to the developers. Tests can be in a sub-directory, but should be close by the source. If they are not, then it is easy to lose track of the tests, and they will not be run regularly. We will be moving back to this model, and put the Unit Tests into the same project Workspaces as the code development in the future. That way, the developers can easily run the tests during their normal development phases.

6.1 Developers must help write Unit Tests, …

Developers must also do Code Coverage, …

and will resist doing both. (Resist might be too weak a word in this case.)

Given the small size of most Evaluation groups, having developers help create Unit Tests is an important requirement. Our ratio is 6:1, in some groups it is 20:1 or more. The Evaluation team may also be good programmers, but there is no way one person can keep up with writing Unit tests for 5-6 others. Besides, who else know the programs better, than the engineers who developed them do? Other reasons include:

1. Pride of ownership
2. Responsibility for changes
3. Higher Quality code

One argument against allowing developers to create test cases goes like this:

Often the developer is trained to develop software to perform a function, but rarely is the developer trained to develop good test cases. As a result, there can be a build up of many tests in a test suite that essentially covers the same code.

However, if the developer is also required perform Code Coverage analysis, *the game changes*! In our organizations that have required the *developers* to perform code coverage **[4]**, the quality and productivity drastically rose once the developers realized that they could actually find problems and keep them the quality up. In fact, average productivity rose 4x. At the same time, the number of bugs decreased by the same amount.

And, with practice, PEOPLE IMPROVE. If they were trained to create programs in the first place, they can be trained in the same fashion to learn to develop good test cases. If you never start, you never learn.

6.2 GUI Testing

Subtitle: But we can't test that! (Say it in an annoying tone of voice for the full effect)

One of the mistaken beliefs of many current GUI developers is that they can't Unit Test Windows and Controls and all those nifty GUI things. You need an expensive GUI test tool.

HOGWASH!

Every mouse click or action is a function call, so why not call the functions directly?

```
# Call your functions in C/C++ or VB!  What else could be simpler?
Thatform_click()
Ret = Thatform_dblclick()
If Ret != 0 then
      Print "We got a problem big guy!"
Endif
```

Challenge the developers to programmatically call their own functions! In reality, they may not have to call them all. Just the ones that actually do something important or have more than 1 line of code in them. This is a case to be pragmatic and use the Code Coverage to determine which functions really need coverage or not. If the resize () functions are just hooks to the underlying system functions, then obviously these functions can be skipped.

If a developer can't programmatically call their own functions, then how does their application even work to begin with? How do they expect an Evaluation team to even test their code?

The point to get across is that Unit Tests provide FUNCTIONAL TESTING. Which, if you don't have it, is a heck of an improvement over what you already have today!

Over 50% of the EDB code base is in the GUI area, which uses MFC C++ and other classes, derived from 3rd party tools. To achieve better results on all the little methods used by MFC requires better exercising of all the keyboard and mouse paths. Most of our unexercised code in EDB is the result of these thousands of little functions, which are difficult to test with Integration Tests.

On other projects, we are starting more use of these GUI Unit Tests. The results have been very encouraging so far, as the developers feel ownership, and have a basic reference to check their changes against.

But, there are still many excuses against Unit Tests. Some of the more common are…

6.2.1 The Too Many Changes Excuse…

Okay. GUI's change. Probably too fast at first to truly write useful Unit Tests. So just provide a simple harness that calls the most important function(s) in that part of the GUI, make sure some end values are correct, and move on. As the GUI stabilizes, then have the developers go back and add more tests.

You can sell this to developers with the pragmatic argument that only the most important features will be called. Realize however, that this is a false economy! As with other Unit Tests, doing as much as you can sooner is cheaper over the development lifetime.

6.2.2 The Cost Excuse…

"But it will take too much time for me to change the tests!"

Pushing the costs of maintenance and quality up front? Making the developers aware of the ramifications of their changes? This is good! Maybe they will prototype better in the future if they are aware of the back-end costs!

6.2.3 The Performance Excuse…

Too often the argument will get turned around to performance (speed!). Remind them politely, THAT AIN'T THE GOAL! THE GOAL IS TO SEE THAT YOU DIDN'T BREAK THE FUNCTIONALITY LIKE YOU DID LAST TIME!

On the whole, diplomatically conveying this message is better. But sometimes, you cannot be subtle!

6.2.4 Unit Test Spiral Phenomena…

Once developers actually have a unit test that they compile and run after they make a change, then the change tends to get added to the unit test. Once that change is checked in, then they add the next change and update the unit test to test their new change, and so on and on.

After about 6 months of this, all of a sudden developers realize that all those little changes they have been making have changed their little unit test a Unit Test. (note the capital letters). And that they actually can rely on their own test systems for results.

We have seen this and done this. It is an impressive phenomenon. And unfortunately, rare.

6.2.5 GUI Verification Requires Work…

Okay. This is a valid argument for using high-end tools. They do have more verification and validation tools. You obviously, except for a very few cases, DO NOT want to capture pixels. But again, testing the results is *easier* in a small Unit Test app that has access to the underlying data structures and can do a better job of comparing results. In these cases we focus on getting the underlying data to be correct, and generally let the pixels take care of themselves. Cross-checking the pixels at key intervals in the development cycle is built into our test plans, so the problems will eventually be seen.

And if GUI Verification requires **Work**, why isn't the whole team helping the Evaluators? Or, do they just want to push off doing the Evaluation portion of the job. We run into this cultural impediment and prejudice against doing the work so often!

6.3 Automation under MSDEV workspace

The process of instrumenting , running tests, updating the test data and generating code coverage reports is automated using makefiles . Each project is created under a separate MSDEV workspace.

`Workspace` **`ED`**

- Build EDB Debugger. Debug and Release modes.

`Workspace` **`Viscov`**

- Instrumentation
- Generate Reports (Raw coverage data file is generated in html and text format.)

`Workspace` **`Unit Tests`**

- Compile all Unit tests
- Run all Unit Tests (runs the tests, compares with gold files, generates a report file)

`Workspace` **`Integration tests`**

- Run all Integration Tests (define instrumented executable path, load the Test Suite driver, and run the tests)

Thus our process is fairly modular. A regular development cycle is **Ed, Unit Tests.** A regular test cycle is **Ed, Unit Tests, and Integration Tests.** A code coverage cycle invokes **Ed, Viscov, Unit Tests, and Integration Tests.**

We plan to move to a new model where we can do the tasks of building, instrumenting, testing, generating coverage reports all in one Work space under different projects.

An example of running Unit tests under MSDEV workspace **All Unit** tests is shown in Figure 6. All a developer or tester has to do is Right click on a project to run the individual tests. Right clicking and selecting the *Build* rule on the Unit Tests or the Test Drivers will run the Unit Tests or rebuild the Drivers.

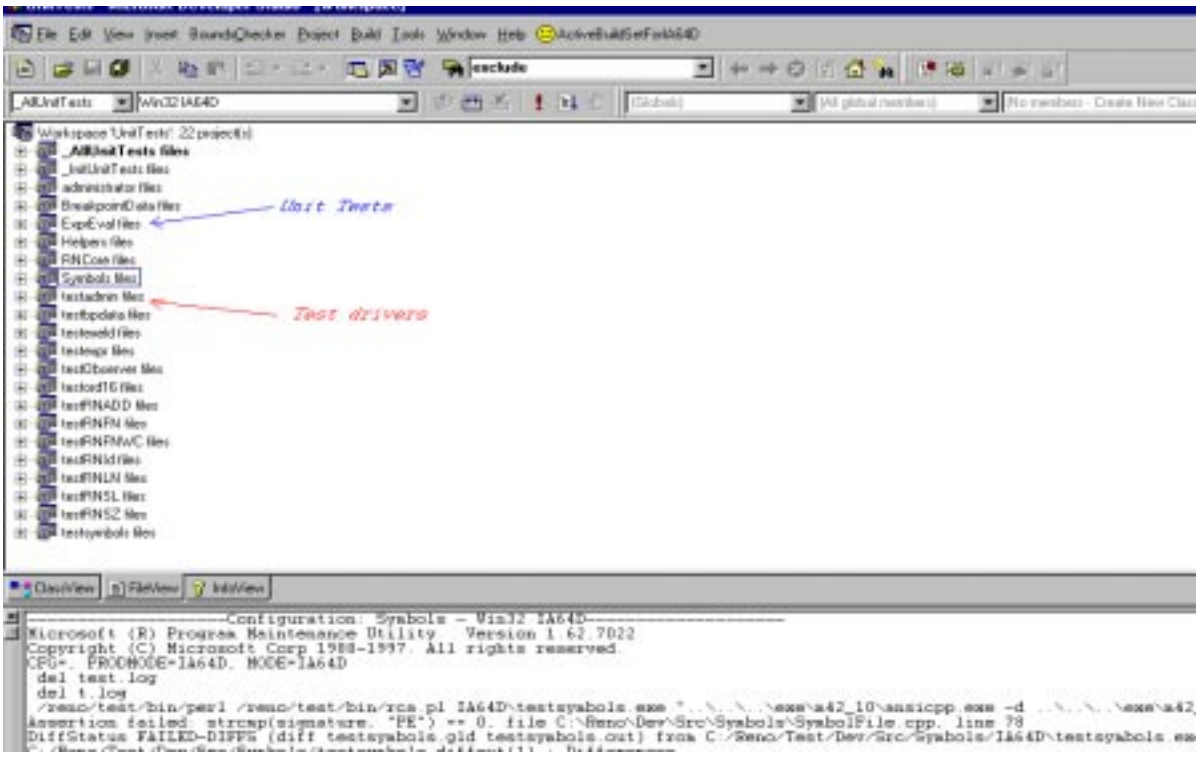


Figure 6. Running Unit Tests from MSDEV

As a practical matter, the UnitTest.dsw file references a separate .DSP files in the rules. And for the actual rules we put them into different ". mak" files in each of the sub-directories. This makes possible running the tests without the overhead of MSDEV for automatic tests on test machines.

Integration tests are similar, but go to a single directory and makefile.

The Integration rules are "NMAKE /F _ALLGUITESTS.MAK <RULE>", where the <RULE> is a specific test suite.

6.3.1 MSDEV Macros

Although the macro of language of MSDEV is painful to learn and use, it has yielded some useful items. We have created several project macros for overnight builds and for switching from Debug to Release modes. The *ActiveBuildSetforIA64D* on the Toolbar in Figure 5 sets the all the sub-projects in MSDEV to a specific build mode.

7 Selling the Benefits

Management support of Code Coverage goals is crucial for integration of Code Coverage into the development cycle. Without it, development teams who have never done it before, easily shrug off the need. Yet how to sell it will remain a continual challenge.

Reasons we have given in the past are:

- More reliable estimates on schedules and quality
- Way to measure progress and fend off upper management
- Feedback to the development team and improvement of the development process.

Nonetheless, there many cultural impediments to such wholesale changes. It is never easy to sell Code Coverage to groups that are dead set against being measured or having the results used against them. And there are other items to think about too.

7.1 Resources

We have seen that adding Code Coverage can take anywhere from 5-15% of the total effort for a project. The primary bulk of this effort comes from the test suite work, and bug fixes that the Unit Test cases exposed.

Selling a 15% hit on effort is always tough. Even though the management team may want the quality that this imposes, it means **GIVING UP FEATURES!** Always a popular option to bring up. Nevertheless, be up front with the cost estimates.

7.2 Defect Metrics

Tom DeMarco makes the point in "Why Does Software Cost So Much?" that if he had only one metric worth collecting it would be ***defect count.***

By the time of the PNSQC conference, we hope to have defects rates for our older products without Code Coverage versus those with.

7.3 The (False) Promise of Eliminating Tests

Using code coverage provides a way to determine the effectiveness of a test case. One can remove a "useless" or redundant test from a test suite once the effectiveness of a single test case can be determined.

Unfortunately, that takes time to analyze the runs and merge the tests together. Although we have promised this ourselves and sold it to management in the past, it is easier to just add more tests than do a thorough analysis of them and see which is more effective overall.

The tools might help, if they could do an analysis of each run and find those runs where the intersections or unions of each run could be analyzed separately. Sadly, this is still a largely manual process at this point except for very high-end tools.

To sell it, we recommend not promoting this aspect of Code Coverage too highly, except as a possible optimization later in the future.

8 Conclusions, Results and Mistakes

Some of the conclusions we have reached from our use of Code Coverage.

- C-Cover only measures coverage of branches, not lines. Some of our C++ destructors and other cleanup code with " IF-ELSE" guards never took the nonexistent ELSE path because the data was always present. These functions show 100% line coverage, but only 50% branch for such cases. Extra processing of the reports eliminated these wrong counts.
- To get better code coverage we had to add Unit Tests for some of the core areas, and enhance the existing unit tests to be more effective. Some sections of the code require more Unit Testing than we have currently provided for. But, we have the tests in place for when resources are available to fully flesh them out.
- Using Rational's (earlier Microsoft's) Visual Test was a mistake in the long term, but not a fatal one. Although the cost was good, the overhead and development costs have been very high.
- Initially we performed Code Coverage once a week. Experience has shown us that this does not allow enough time to identify and add additional tests or improve the current tests. "Testing often" does not get us any additional benefit, unless the tests are improved or new tests are added. The interval of code coverage measurement is changed to once in two weeks as the product as matured.
- Automated builds helped the optimal utilization of QA resources, as we could focus testing and code coverage analysis.
- As the screen layout changed often during the initial development cycle, Integration tests were written only after EDB has stabilized **or** we simply captured the screens but did not compare them. Since we did not have many automated tests in the beginning the code coverage did not increase during initial stages.
- Even with the best automated test tools, manual intervention is sometimes required for failing tests as they pop up unexpected windows requiring unanticipated user action. Complete automation of code coverage testing may not always be possible. **Semi-Automation** of tests can sometimes help for these cases, particularly for Video or Audio tests where a human is best able to check the results. However, be sure semi-automated tests also have a default mode of running without human intervention.
- Tracking code coverage and open defect metrics on a regular basis helped us in directing our test activity by using the metrics as guiding parameters.
- It is easier for the developers to write some of the tests, as they better understand the features in their specific area that need testing.
- Having both Unit and GUI tests runnable from within the Microsoft Visual C++ environment, makes it possible for the developers to write these tests and run on their own systems. Code Coverage can be improved with developer participation in writing tests Automation of the build process enabled the QA/Evaluation team to focus on testing, measuring code coverage and enhancing the tests to increase the code coverage. Everything started from a central Makefile, for ease of use and tracking.

8.1 Test Suite Mistakes

We made a mistake with EDB, by keeping the tests in a parallel tree structure. This was an explicit decision was so that the tests could be shipped separate from the source base. The unspoken justification was that the developers did not want to have to maintain the darn things. So, a conscious decision was made to separate the tests from code base.

But, then this also isolated changes and made maintenance more difficult. As developers charged ahead with changes, they never ran the Unit Tests regularly. QA had to run the tests, track down the exact failures, then report as new defects to get them fixed.

The model used in other Intel groups was to always run the Unit Tests before checking in changes and enhancements. Tests were in the same tree as the code. This was more effective, and the productivity of the group was over 1500 DSI per man month **[4]**, as opposed to our current 600-700 DSI.

9 Acknowledgments

The authors wish to thank Dale Fugate and Larry McGlinchy for their support and valuable suggestions in implementing the Code Coverage measurement and analysis as part of the test and evaluation of EDB. The Java-filter program to exclude user-defined list of functions/DLLs/classes in Visual Coverage was developed by Ben Cichy during his Intel Internship. The reviewers' comments on early versions of this paper, particularly Linda Westfall, were greatly appreciated by the authors.

10 References

[1] Foye, Sharon : Reno Debug Tool Architecture & Reno Application Debugger Functional specification. Debugger Technology Group 2 / 16 / 96

[2] Vireday, R. and Rao, P.S. Reno Debugger Test plan for Alpha and Beta Releases Debugger Technology Group 8 / 30 / 96

[3] McGlinchy Larry, Rao Pemmaraju.S. Vireday, Richard An Implementation of QA & Evaluation Process for the Intel Enhanced Debugger Project, Intel Software developers conference Oct 27-29, 1997 Proceedings - Practices & Experiences Track 3 pages 220-226

[4] Vireday, Richard and Roger Auble and Dave Regis. A Working Parallel Software Development Methodology. Intel Software Developers Conference. October 1994.

Product: 600K Code Base, 150K DSI. Reached a rate of 1500 SLOC per MM. Development methodology most important for our current efforts.

[5] Humphreys, Watts S. **A Discipline for Software Engineering**. Addison-Wesley, May 1995.

[6] Marick, Brian. "Classic Testing Mistakes". http://www.stlabs.com/marick/Classic/MISTAKES.html 1997. As Brian Marick says, "I ♥ code coverage." We agree Brian!

[7] C-Cover User Manual, Version 3.1, September 1996.
http://www.bullseye.com has more information. The updated on-line help for 4.0 is also an excellent reference point.

11 Tool References

Bullseye C-Cover (www.bullseye.com)

TracePoint VisualCoverage (www.tracepoint.com). A dead URL.

Beyond Coverage Analysis - Time Optimization of Regression Testing

Roy D. Trammell

Performance Tools Group
13575 SW Fircrest
Beaverton, OR 97008
royt@teleport.com

Abstract

A large portion of the cost of some development projects is in regression testing, repeated passes of the product through a set of validation tests. This paper concerns a method whereby the time required for a pass through the set may be predicted and reduced. The method involves measuring coverage patterns of the set and using this information to generate an ordering of tests optimized for the incremental coverage/time ratio. A point in the sequence is selected for either the desired coverage or the maximum attainable coverage, then tests on the tail of the sequence are discarded. The benefits are to:

- *Predict the number of tests required for a given level of coverage.*
- *Determine which tests can be discarded without loss of coverage.*
- *Determine the test subset that runs in least time without loss of coverage*
- *Order the subset such that most bugs are detected early in the test sequence.*

Beyond Coverage Analysis - Time Optimization of Regression Testing

Roy D. Trammell

Performance Tools Group
13575 SW Fircrest
Beaverton, OR 97008
royt@teleport.com

Abstract

A large portion of the cost of some development projects is in regression testing, repeated passes of the product through a set of validation tests. This paper concerns a method whereby the time required for a pass through the set may be predicted and reduced. The method involves measuring coverage patterns of the set and using this information to generate an ordering of tests optimized for the incremental coverage/time ratio. A point in the sequence is selected for either the desired coverage or the maximum attainable coverage, then tests on the tail of the sequence are discarded. The benefits are to:

- *Predict the number of tests required for a given level of coverage.*
- *Determine which tests can be discarded without loss of coverage.*
- *Determine the test subset that runs in least time without loss of coverage*
- *Order the subset such that most bugs are detected early in the test sequence.*

1 Introduction

A method and tools developed to expedite compiler regression testing appear applicable to regression testing in general. This is a description of the method in both theoretical and practical modes. The first sections look at the costs of regression testing in two examples. Then we take a formal approach to coverage analysis by deriving a mathematical model of overlapping test coverage, laying the groundwork for an algorithm to sort and cull test sets based on fine-grained coverage data. The method is described in sufficient detail to be reproduced. Then we show its application to several real-world examples. Last, we look at some shortfalls of this method, and of coverage measurement in general.

2 Example 1: The Expanding Compiler Test

This first example is drawn from a small business, having less than 50 employees, with 2 engineers dedicated to developing a parallel C compiler. Over ten thousand tests were generated by permuting '#ifdefs' within a hundred or so basic tests. Whenever a new category of bug was discovered, another variable was added to the permutation. The developers eventually found themselves waiting six weeks to validate a new release containing three month's work. The costs were two engineers, six workstations, six weeks, four times a year - comes to about $88K/year, and a six week delay on each release. Since they couldn't give the specific reason for adding a particular test, neither could they safely remove any tests when the set grew too large.

3 Example 2: A CPU development project is mostly software

VLSI chip developers rely heavily on gate level simulation before a chip is actually fabricated. These simulations comprise large bodies of code. Some parts are written in a logic simulation language to describe the chip design. Other parts may be written in C or another general purpose high level language to represent system components in

the simulation environment such as buses, peripherals, and memory. Though the eventual result will be a piece of hardware, the task at this point is software development and testing.

Tens of thousands of tests taking up to several hours each may be involved. More resources may be burned in regression testing the simulation than in the actual design process. Some of these costs are: simulation hosts and associated network and administrative support, coding and maintaining tests, and engineering time spent managing the testing process and waiting for test results.

One CPU chip development project burned capital at an estimated $230K/week over two years. A pass through the entire regression suite could take 6 weeks, and be performed 5 or more times. By these assumptions, the simulation costs are 7. 5 months and $7. 5M over the course of the project[1]. The value of the dollars is concrete. The value of the months, in terms of time-to-market, is harder to estimate, but is perhaps greater than the dollars.

The complexity of chip designs is growing rapidly. The simulation time required to validate chip designs is growing even faster than their complexity. For example, a processor support chip design that ran at better than 10 simulation cycles per wall-clock second in the previous generation, now runs at 1.5 cycles/s. The regression suite now contains more tests, and the individual tests now require more cycles. If these trends continue, regression testing for design validation will soon become the dominant time and cost factor. Chip design *is* software development, and its cost is exploding.

4 Constructing a Formal Model

Our objective is to find some way of optimizing regression testing by maximizing the coverage per unit time. To understand the process, we must derive a formula for coverage as a function of test set size.

We will use geometric and statistical techniques to develop a model of overlapping test coverage, expressing cumulative coverage as a function of the number-of-tests. Starting with a simplified model, and making the (erroneous) assumption that coverage is smoothly and randomly distributed over the test set, the model will be extended to cover the general case.

Consider a target under a set of N tests. It has a total coverage area of A_t measured in *atoms* or the smallest individually testable functional elements. Some of these may be elements of the design such as linear blocks of code. Others may be states or components of state such as the contents of a variable. For simplicity, we will treat them as identical in type and value. C_1 is the mean coverage of a single test in the set. The coverage of the entire set is C_N. The mean coverage overlap between a pair of tests is P_{ov}.

4.1 Coverage Overlap

This concept is key to predicting and optimizing coverage. Usually, most of the atoms covered by a test are also covered by other tests. Modeling coverage as a function of test set size is mostly a matter of correctly handling this overlap. The probability of a particular atom being covered by a single test is:

1. Assuming 100 engineers at a loaded salary of $90K, a pool of 500 workstations for simulation at $10K amortized over 5 years and a $2M/year CAD tool cost.

$$P_1 = \frac{C_1}{A_t}$$

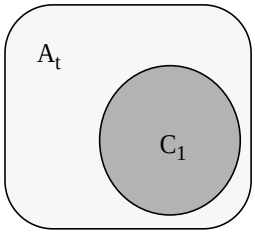
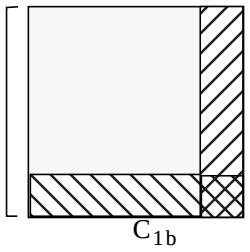


Figure 1: Single test coverage.

So the probability of a given atom to be covered by both of any test pair is:

$$P_{a \cap b} = \left(\frac{C_1}{A_t}\right)^2$$



Eg:
$A_t = 1$
$C_1 = 1/4$
$P = 1/16$

Figure 2: Test pair coverage.

The number of atoms common to both of a test pair is:

$$C_{a \cap b} = \left(\frac{C_1}{A_t}\right)^2 A_t = \frac{C_1^2}{A_t} = P_{ov}$$

We will call the mean of this number the *pair-overlap*

4.2 A Model of Random Coverage

The total area covered by a test pair is:

$$C_2 = C_1 + C_1 - \frac{C_1^2}{A_t} = 2C_1 - \frac{C_1^2}{A_t}$$

Now consider cases of N > 2. The probability of an atom *not* to be covered by a particular test is:

$$P = 1 - \frac{C_1}{A_t}$$

So the probability of an atom *not* to be covered by *N* tests is:

$$P_N = \left(1 - \frac{C_1}{A_t}\right)^N$$

The total area covered by *N* partially overlapping tests is:

$$C_N = C_1 P_0 + C_1 P_1 \ldots C_1 P_{(N-1)}$$

where P_i is the probability of an atom not to be covered by *i* tests. Combining these two gives:

$$C_N = C_1\left(1-\frac{C_1}{A_t}\right)^0 + C_1\left(1-\frac{C_1}{A_t}\right)^1 \ldots C_1\left(1-\frac{C_1}{A_t}\right)^{N-1}$$

Or

$$C_N = C_1\sum_{i=0}^{N-1}\left(1-\frac{C_1}{A_t}\right)^i = C_1\left(1-\left(1-\frac{C_1}{A_t}\right)^N\right)$$

As *N* becomes larger, the coverage asymptotically approaches the total area of the object. When an artificial test set is generated with randomly distributed coverage, its coverage as a function of *N* agrees with the model to within .06%.

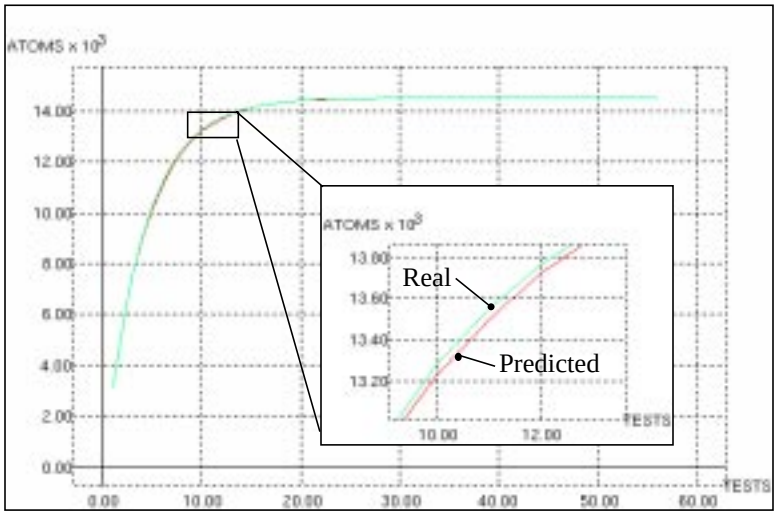


Figure 3: Real and predicted random coverage.

4.3 Modeling Non-Random Coverage Distributions

Real tests do not have randomly distributed coverage. For example in most cases, many atoms will be referenced by all tests. Consider the distribution of overlap as how many atoms are referenced by 0,1,2...*N* tests. This property can be measured and divided by A_t to get a probability distribution *G(i)*, the probability that an atom will be referenced by exactly *i* tests. Figure 4 shows the distributions for an artificial test set with random coverage (a 'normal' distribution), a typically skewed and multimodal one from a real test set, and an artificial one generated to have the same P_{ov}, C_1, and A_t as the real set but otherwise random.

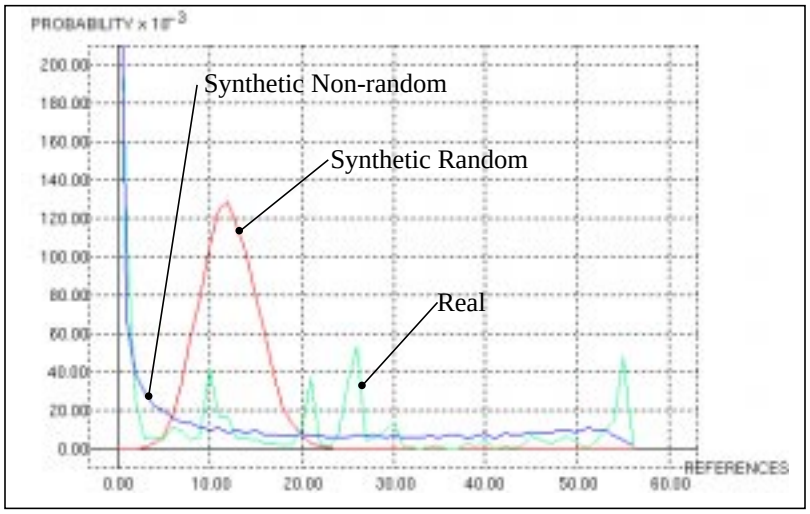


Figure 4: Overlap distributions

To apply the distribution measured from a test set, we need to partition coverage into components referenced by *i* tests, for *i* from 1 to *N*. For each component, we calculate the probability that an atom with *i* references will have at least one reference in *n* of *N* tests. The probabilities are summed and multiplied by A_t to give the total coverage in atoms.

If N is the total number of tests, and *n* is the size of a subset, the probability that at least one of *i* references will fall within the subset is:

$$P = 1 - \left(\frac{N-n}{N}\right)^i$$

The number of atoms covered by the subset *and* having *i* references is:

$$C_n = A_t\left(1 - \left(\frac{N-n}{N}\right)^i\right)G(i)$$

Summing over all possible reference counts:

$$C_n = A_t \sum_{i=1}^{N} \left(1 - \left(\frac{N-n}{N}\right)^i\right)G(i)$$

This is a model of test set coverage with arbitrarily distributed overlap. We can use this model to determine what coverage will result from a subset of the tests we have in hand. Figure 5 compares the coverage curve of a test set with the curves predicted by the two models for that set.

4.4 Predicting Coverage from a Sample Test Subset

Unfortunately, we can't use this model to predict what coverage will result if we write more tests. Even if we assume the new tests will have similar coverage properties, the distribution will vary with test set size, albeit in a predictable way. Given that assumption, however, we can predict the new distribution (of a superset) by scaling the one in hand upward in the x-axis (atoms), and downward in the y-axis(probability). We do this such that the range of the

result is the superset size, the ‘shape’ is preserved, and the probabilities are normalized to retain the property of summing to 1.

$$G'_i = \left(\frac{N}{N'}\right) G_{i(\frac{N}{N'})}$$

Adding tests tends to a smoother distribution and more accuracy may be obtained if the distribution is smoothed before scaling. The method used here is to generate a synthetic test set with the same C_1, A_t, and P_{ov} as the original, but otherwise random, and measuring its distribution. The result will fit the coverage curve of the original, but will be a better predictor of supersets. It would be interesting to experiment with other, more direct, smoothing algorithms.

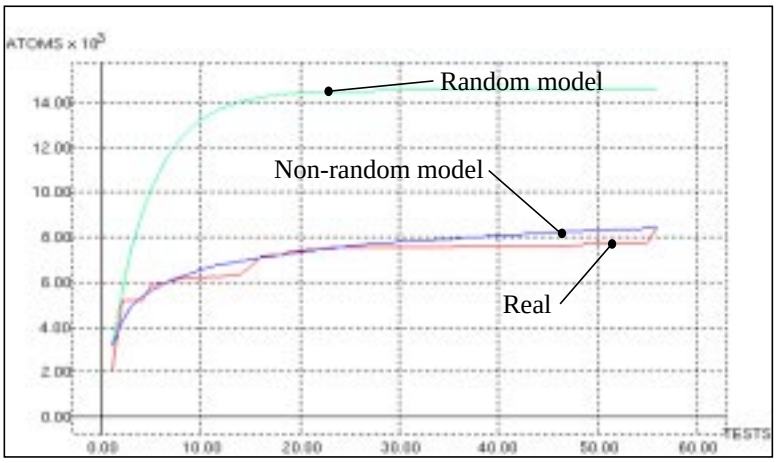


Figure 5: Real data and the models.

Figure 6 is the result of applying this model to a real dataset of 56 tests, to predict the coverage curve of 200. The model’s prediction is compared with the 56 tests of real data, and with a montecarlo simulation of 200 tests having the same A_t,C_1, and P_{ov} as the actual tests, but otherwise random.

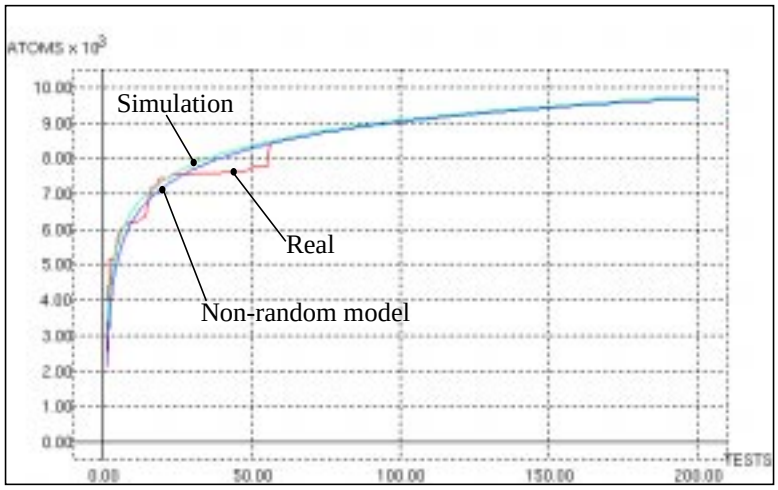


Figure 6: Predicting 200 tests from 56

Shuffling the test order yields a family of real coverage curves illustrating more clearly how the model fits the data.

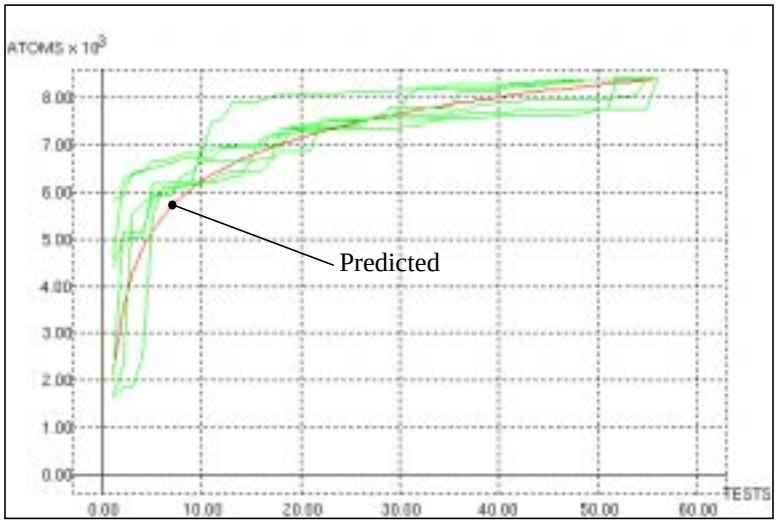


Figure 7: Central tendency of the model

This model can predict the coverage of supersets tens of times larger than the sample, if there are sufficient samples to measure. Figure 8 shows the degradation in accuracy from trying to predict progressively larger sample-to-superset ratios.

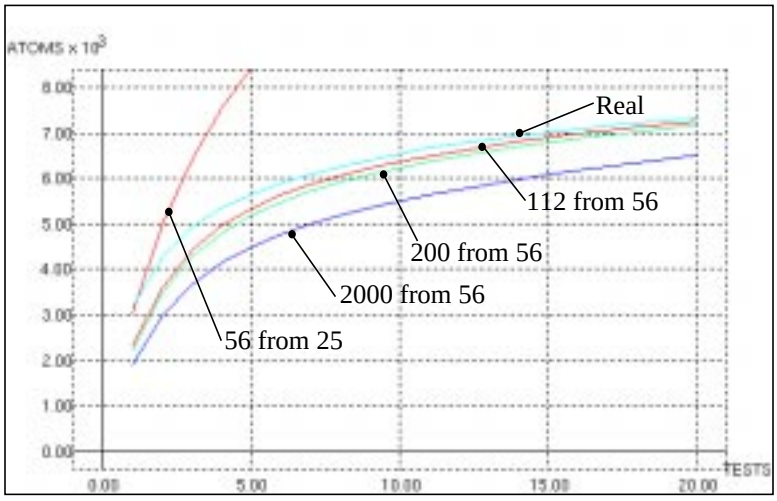


Figure 8: Effect of sample size on error.

5 Assumptions and Sources of Error

*All atoms are equivalent in coverage type and value***:** All bugs are not equally bad and all coverage atoms are not equally good. This must be considered when interpreting coverage statistics. A weighting scheme could be introduced but might not be worth the effort.

Sample test sets have coverage properties that are representative of their supersets: As with any statistical technique, sample size and the method of sample selection will affect the accuracy of predictions. If you start out writing large, generalized tests, then later switch to small tests focused on specific components, early samples will be poor predictors of later coverage.

The G(i), A_t, and C_1 are sufficient to determine coverage: We have seen that error is small, given sufficient samples. Still, other properties not modeled may account for some of the observed error. Some possible candidates are distribution of coverage per test and distribution of overlap per test. It may be possible to improve the model by treating them as we treat the distribution of overlap per atom, reducing the sample size required for a given degree of accuracy.

An artificial test set generated with the same G(i), C_1, and A_t, as the original, but otherwise random, will have a smoother distribution with the same coverage curve as the original, and will be a better predictor of larger sets than the distribution of the original: This makes intuitive sense. It holds true for the limited data available and in simulations. The three relatively small test sets are encouraging, but fall short of a proof. More and larger data sets are needed to establish this with confidence.

6 A Sequence With Some Useful Properties

A key attribute of regression tests, whatever the medium, is a high degree of overlap in coverage between tests. There tends to be a large common portion of the test target required for almost any test to succeed. So the time vs. coverage graph of a random test sequence is an exponential curve approaching the total area of the target. Figure 8 shows these curves for several unordered test sets, normalized in y.

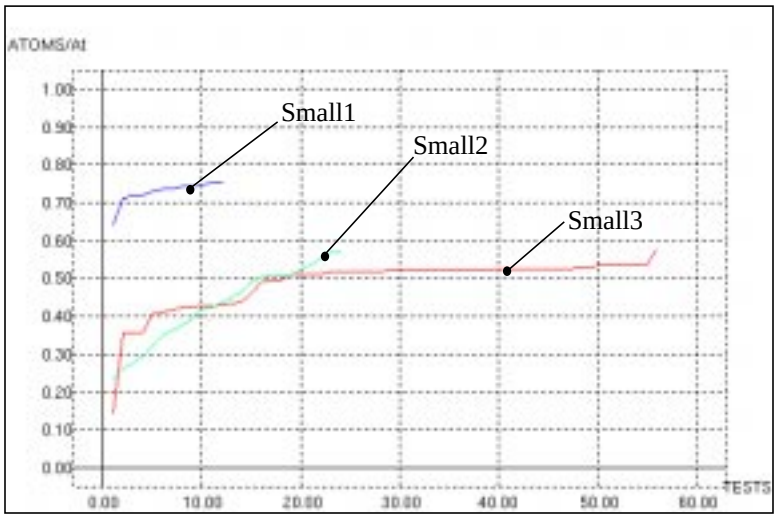


Figure 9: Variation in coverage curves.

There is likely to be more than one test combination that covers a specified number of atoms. These combinations will have different run times, and one will be optimal. These run times may differ considerably. In large sets, they may vary by 1 or 2 orders of magnitude.

There is one sequence of tests in a given set which optimizes the incremental coverage/run time ratio. Put another way, all initial portions of the sequence, regardless of length, contain the combination of tests which finds the most bugs in the least time, for that number of tests. This sequence will have a time-coverage curve with a larger exponent

than the original random sequence, It yields, for all lengths, least time for a given degree of coverage. The remainder of this paper is primarily concerned with a practical means of generating that optimal sequence.

At the tail of this sequence are usually some tests which contribute no additional coverage. Assuming that a test which contributes no additional coverage will detect no additional bugs, all these tests may be culled. Discarding them leaves a subset which provides the same coverage, but with reduced run time.

Larger test sets yield better reductions due to the larger combinatorial space from which an optimal sequence can be selected, and their tendency to have higher pair-overlap. This may range from 1.5X with a few tens of tests, to an order of magnitude with hundreds of tests. In one real example, involving over ten thousand tests, run time was reduced by two orders of magnitude.

Even without any culling, there is an advantage to performing the tests in this optimal sequence. Most bugs will be detected early in the testing process because the ‘unique’ or incremental coverage contributed per-test falls off exponentially, and the probability of detecting a bug is proportional to the unique area of coverage. This is also true, to a lesser degree, of any randomly ordered test sequence.

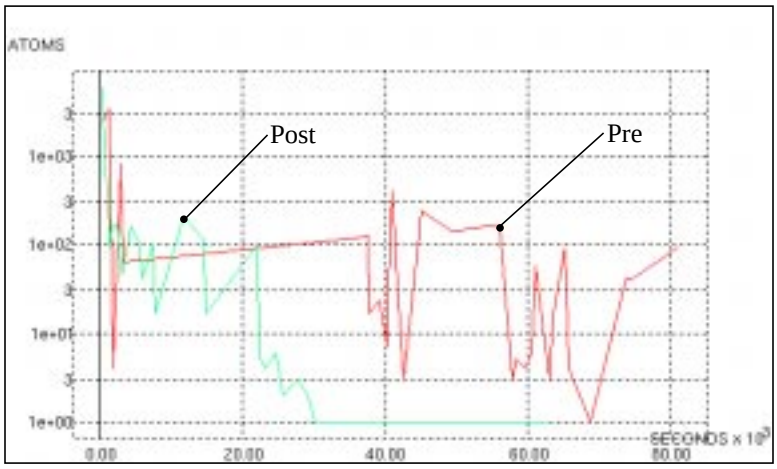


Figure 10: Pre vs. post-optimized incremental coverage (Log Y scale)

Another implication of the exponential curve is that, once the optimal sequence has been generated, further large reductions in run time may be achieved if small reductions in coverage are acceptable. In one example (Fig. 11) the initial reduction was from six weeks to 11.4 hours with no loss of coverage, and to 3.5 hours with a 1% coverage loss.

7 Taming the Combinatorial Dragon

A set of tools which perform this prediction, ordering and culling process for software regression tests has been developed. They operate on instruction execution profile data from PURECOVERAGE (a product of Pure Software Inc.), or from the GNU gcc compiler tool GCOV (Free Software Foundation). The optimizing algorithm seeks to assemble a sequence, one element at a time. At each step, the unsorted residue of tests is searched for the one with the greatest ‘merit’. Merit is determined by comparing a candidate test’s covered atoms with the atoms already covered by the partially assembled sequence. Those not already covered are the ‘incremental coverage’ of the candidate test in one specific place in the sequence. The merit is this number of atoms divided by the test run time. Note that merit depends on position in the sequence. Pseudocode for this algorithm is given below.

```
sort(SEQUENCE *random, SEQUENCE *ordered)
{
    set_empty(ordered);

    while(nonempty(random)) {
        best = 0;
        test = cand = get_first(random);
        while(cand){
            merit = get_merit(ordered,cand);
            if(merit > best) {
                best = merit;
                test = cand;
            }
            cand = get_next(random);
        }
        if(best == 0)
            break;
        remove(random, test);
        append(ordered, test);
    }
}
```

The comparison function *get_merit()* computes the additional coverage a candidate would add to the sequence, divided by the run time added.

```
get_merit(SEQUENCE *set, TEST *cand)
{
    unique = 0;
    for(atom = get_first(cand); atom; atom = get_next(cand)) {
        found = FALSE;
        for(test=get_first(set); test; test=get_next(set)) {
            for(set_atom = get_first(test); set_atom;
              set_atom=get_next(test)) {
                if(atom == set_atom) {
                    found = TRUE;
                    break;
                }
            }
            if(found)
                break;
        }
        if(!found)
            ++unique;
    }
    return( unique/get_runtime(cand) );
}
```

A brute-force sort utilizing this algorithm takes an impractical amount of compute time for non-trivial design complexities and test set sizes. Comparison of candidates for a given position in the sequence is computationally expensive, being proportional to $tests^3$ X $atoms\_per\_test^2$. The first version required several hours to process a set of fifty compiler tests. The algorithm has been refined in several ways to reduce this time by several orders of magnitude to a manageable level:

- Keeping bookmarks for directory, file, function, and block context to exploit coherence of the input data in these dimensions between tests.
- Dropping covered atoms from later passes. As coverage of the ordered set rises, atoms_per_test in the unordered test pool shrinks, making merit evaluation cheaper.
- Dropping uncovered atoms from the database before the sort.
- Revising the algorithm to operate incrementally on test subsets. This allows dividing the task into many runs, with small numbers of tests each. The reduced results are merged and become the input for another tier of pro-

cessing, and so on. This avoids the combinatorial explosion of large sets. Tests are eliminated earlier and more cheaply, and the task can be performed in parallel on multiple hosts.

The hierarchic divide-and-conquer strategy was also desirable for other reasons. Operating on more than a few hundred tests at a time would require an improbable amount of memory. Also, practical aspects of iterative development make it desirable to revise some of the tests without resorting the entire set.

These optimizations have made the ordering and culling process nearly linear with respect to test set size.

8 Two Trial Runs

The original application for this optimizing technique was a compiler test suite with thousands of tests, requiring 6 weeks on 4 workstations to execute. The run time improvement with no loss of coverage was better than 100X (from 6 weeks to 11 hours). These were random tests varying combinatorial parameters, so there was a high pair-overlap factor. This partially accounts for the large reduction in run-time. Many regression suites are more carefully focused, and unlikely to achieve this degree of improvement.

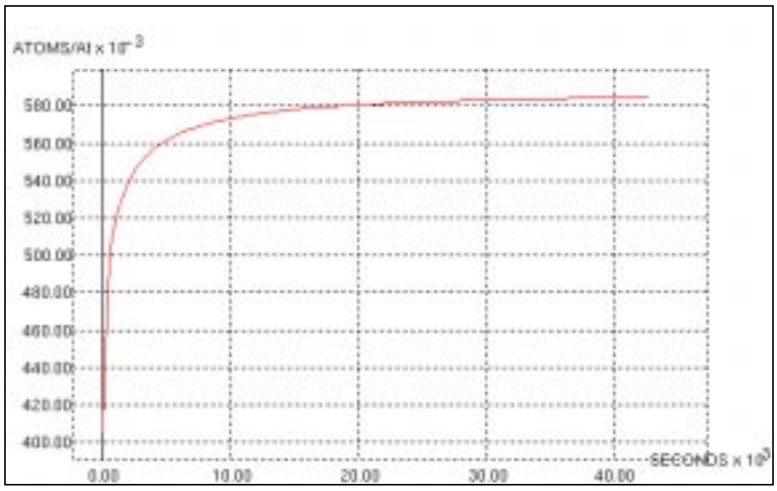


Figure 11: Big1 Optimized coverage (Un-optimized data not available)

Gathering the coverage information for this large test set generated tens of gigabytes of data, which created some challenging problems in spooling, storage, network bandwidth usage, and staged data reduction. Ways of dealing with this were developed, involving application-specific data compression, shell scripts to manage the process, and the algorithmic improvements mentioned previously. Unfortunately, these raw data were not retained.

The second trial involved three separate regression suites targeting different libraries of software functions. The following figures (12a,b,c) show the reduction of run time in seconds vs. coverage.

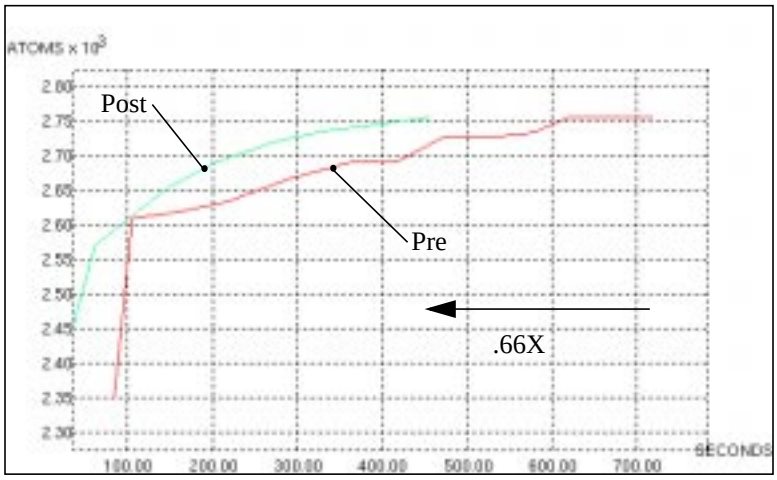


Figure 12: (a) Small1 Pre and post optimized coverage.

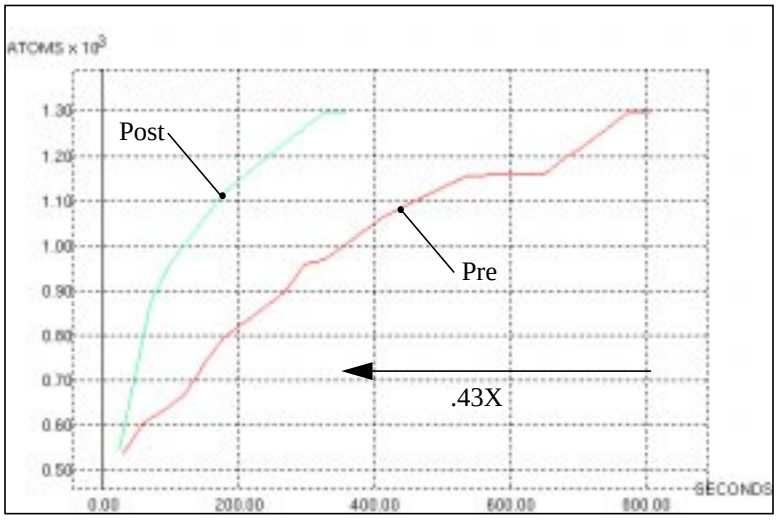


Figure 12 (b) : Small2 pre and post optimized coverage.

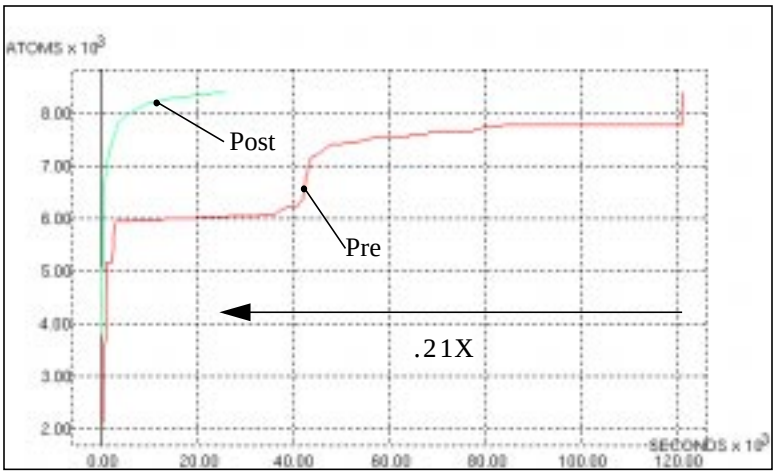


Figure 12 (c): Small3 pre and post optimized coverage.

Trial	Original Tests	Total Area	Mean Overlap	Total Coverage
Small1	12	3657	2333	2756
Small2	24	2282	450	1299
Small3	56	14532	2032	8413
Big1	>10K			

Trial	Original Run-time	Tests Culled	Optimized Run-time	Time Reduction
Small1	.20 hrs	4	.13 hrs	.66 X
Small2	.23 hrs	15	.10 hrs	.43 X
Small3	33.5 hrs	23	7.2 hrs	.21 X
Big1	> 1K hrs	> 9K	11.4 hrs	< .01 X

Figure 13: Summary of trial results.

9 Application to Chip Validation

Optimizing silicon design validation is equivalent to optimizing conventional software testing. Some issues involved in adapting the sequencing method to this domain are programmatic, some are procedural. Work is in progress to apply this technique to VLSI regression test sets. This section explores several issues encountered in that adaptation. Note that much of this applies equally to other forms of software testing.

The tools we have operate on tests instrumented for instruction execution profiles. Some simulation engines have built-in coverage metering options. This is the best metric for optimization, superior to the execution profile data employed in the trials. Adapting the tools to use the native simulator coverage data is a format conversion issue.

Other simulators compile the silicon design into executable ‘C’ code such that functional blocks of silicon are represented by blocks of code. There, the existing optimizer can be applied without modification.

When neither kind of data are available, we are out of luck, unless some other coverage metric can be devised. A usable metric must identify which atomic design elements are exercised (in a binary sense) by each test in the set.

A huge bulk of raw coverage data will be generated (e.g. 25GB for 5,000 tests). In the trial using the compiler tests, this created problems in storage, staging, and network bandwidth. These problems were further complicated by the coarse-grained parallelism of the divide-and-conquer algorithm. The process of data acquisition and management will need to be scaled up and customized for the design validation problem.

The generation of coverage data requires one full pass through the test set, a significant outlay of computing resources and time. Addition of coverage metering increases this time by 1.5X to 2X.

The consequences of overlooking a chip bug are so severe, there is some reluctance to trust an unproven technique which discards many tests. One way to address this is to initially apply the reduced test set to the penultimate regression passes, running the full set once, before the tapeout date, and perhaps one other time in the middle of the process. This will still provide substantial gains if 4 out of 5 passes take 1/4 the time (not an unreasonable expectation). Eventually, this safety net could be omitted.

As the design process proceeds, changes will degrade the relevance of previously acquired coverage data. As long as the functional requirements of the design do not change much, a test which covers a given functional block, will *tend* to cover a functionally equivalent block. Up to a point, the same tool that generates the optimal test sequence can be applied incrementally to a design as it changes, to test if coverage is still maintained, and select additional tests to repair coverage holes.

10 Other Areas of Application

If an application meets certain criteria, it is a candidate for test reduction by this method. Those criteria are:

- More than two passes through the test set will be made.
- The cost per pass (e.g. in time, money, or scarce materiels) is high enough to warrant the extra effort, or the reliability requirements are high enough to demand maximum return on testing resources.
- Fine-grained coverage data per-test can be acquired.

Some applications which seem to meet these criteria are: military and avionics software, compilers, operating systems, banking and financial software, and software-controlled medical equipment. There are also mechanical and electronics applications.

11 What Coverage Tools Can’t Measure

It is tempting to believe if a coverage instrument reports 100% coverage (which almost never happens), all faults will be detected by the test set. Such a report would actually refer to 100% of the *measurable* coverage. Some tools measure factors which only correlate roughly to coverage. Others use application-specific knowledge to measure the coverage in a more direct way. RTL simulator coverage tools are an example of this later class. They make the coverage determination in a more functional way, detecting which wires are toggled, which states of a state machine are visited and from where, and which branches of logical decision-trees are taken.

They measure the states of each explicit ‘state machine’, but not the metastate of the combination. They measure the toggled signals but the information regarding toggle time and direction is usually lost. The coverage of states implied by the combination of signals is not measured. Sequential coverage information is also lost. If all these were considered, then all the metastates of the chip would be measured, but that would be an impossibly large number.

Similar caveats apply to tools which measure coverage of software targets by instruction execution profiles. In addition to most of the weaknesses cited above, they also have a major blind spot in the area of data dependencies. Data variations which produce erroneous output, but do not result in a unique flow of control may not be covered, even if all instructions have been visited.

So it seems that coverage tools can't be relied upon to conclusively measure coverage. As a consequence, automated test set ordering and culling algorithms which use such data cannot be completely trusted to retain all useful tests. We must consider their blind spots and manually insure that the set contains tests which cover them.

12 Conclusion

We have derived a mathematical model of overlapping test coverage capable of predicting the coverage produced, and time required by a subset or superset of sample tests. This model is based on measuring the probability distribution of the coverage overlap. We have presented a method of reducing development costs and time-to-market of products which have a large investment in regression testing. This is done by using specific (as opposed to statistical) and comprehensive fine-grained data on per-test coverage patterns to generate an ordered subset of the regression suite having the optimal sequence that guarantees the highest rate of coverage over time, for any initial ordered subset of that sequence.

Tools have been built to implement this method, and the results of several trials are included. In one case the run time was reduced by two orders of magnitude without loss of coverage.

Even the best coverage analysis tools are inherently incapable of measuring certain types of coverage, and these weaknesses propagate to tools which use their data. It is possible to guard against these blind spots once they are understood. Then the application of tools which order and cull test sets based on coverage data can dramatically reduce the time and resources consumed by tests which must be repeated.

Third-Party Registrars' Audits - for Better or for Worse?

Robert C. Bamford and William J. Deibler II
Software Systems Quality Consulting
2269 Sunny Vista Drive, San Jose CA 95128
Internet: ssqc@concentric.net
World-Wide-Web: http://www.ssqc.com
Voice 408-985-4476
FAX 408-248-7772
July 1998

Abstract

This paper reviews the potential impact of third-party, standards audits on an software organization's ability to respond to changing requirements. The presenters focus on the responsibility of the auditee's management to ensure that the registrar's audit presents a balanced view of the auditee's position with respect to a standard. While an ineffective audit and an undeserved certification may seem at first like a gift, they waste the time and money invested in the audit. An ineffective audit undermines carefully-nurtured cultural attitudes about quality, standards, procedures, and audits. At worst, an ineffective audit can obscure problems and divert a software organization from addressing real problems that adversely affect customer satisfaction, the quality of products and services, the efficiency of processes, and the business viability of the organization.

Drawing on their extensive experience with ISO 9000 in software engineering environments, the presenters discuss actual problems they have encountered and suggest techniques for ensuring that third-party audits are effective - for ensuring that major nonconformities identify serious problems.

The presentation includes the current state of international efforts, like ISO/IEC Guides 61 and 62, QSAR, IAF, and the IIOC, which will have significant impact on registrar selection and on the future acceptance of ISO registration.

While the authors' examples are drawn from ISO 9001 registration audits, the principles, problems, and remedies they recommend translate readily into any standards environment (e.g. SEI Capability Maturity Model, Trillium, Bootstrap).

Keywords

ISO 9001, ISO 9000-3, Assessment, Audit, Compliance, Registrar, Standards, TickIT, Software Certification, Software Quality Assurance Models

Objective

The authors' immediate objective in writing this paper is to identify and promote discussion of institutional problems that surround current registration practices, which grew unexpectedly rapidly following the release of ISO 9001, 9002, and 9003. At best, these problems reduce the value of obtaining and maintaining ISO registration. Software organizations that develop and implement ISO-compliant business systems are often disappointed in the depth of the registrar's audit. At worst, these problems result in the registration of non-compliant - incomplete or ineffective - business systems. At either extreme, the people in the affected organizations come away with diminished confidence in the registrar, in the registration process, and, by extension, in the standard, itself.

For Example

The most striking example we have encountered is of a division in a company that set an aggressive target for registration - against the advice of others in the company who had already achieved registration. The division in question achieved registration, but only because the registrar's auditors overlooked a number of major nonconformities. Much of the work done in the division to achieve registration was wasted window dressing - which everyone knew. Individual contributors lost faith in management as

they tried to reconcile the obvious waste with company objectives to cut costs - and as they tried to maintain project schedules and the extra work "for ISO". Achieving registration and receiving the amazed congratulations of their peers only increased management's fear of the outcome of the semi-annual surveillance audits. Not to mention the effects on the division's customers, who may have made decisions based on the registration.

While the conclusions addressed in this paper raise concerns about the value of on-going ISO registration as it is currently practiced, two fundamental conclusions continue to underlie our work:

> ***We endorse ISO 9001 - independent of third-party registration - as a foundation for developing a quality culture and organization***
>
> ***Auditors who work for registrars are competent and capable. As in all populations, some auditors are excellent, some are better than others; everyone has good days - and bad days.***

Our conclusions are based on our personal experience in teaching, consulting, and in auditing. The examples presented are anecdotal and, of necessity, anonymous. However, concerns about the process of obtaining and retaining ISO registration are being expressed with growing frequency in the press, on the Internet, in a variety of international efforts to harmonize registrar accreditation, in the world-wide automotive industry's perceived need for Big-Three-initiated QS9000 standards, and in the proposal led by Hewlett Packard and Motorola[1] to redefine the registrars' involvement in registration maintenance.

We have grouped the institutional pressures that affect registrars into two categories:

- The business environment
- Clients

The Business Environment

The principal institutional pressure on registrars is competition. With some 70 registrars doing business in the United States, it is no surprise that a 1995 survey "found that the average price for a first-time ISO 9000 registration is $10,920, down 9% from last year's average. The range of responses from the 34 participating registrars was $3,500 to $20,400."[2] Because the international standards associated with ISO registration[3] do not prescribe minimum duration or sample sizes for audits, as QS9000 does, minimizing both on-site and preparation time are obvious methods for cost reduction. It is not unusual for a member of the audit team to receive his or her first exposure to the client's business practices in a briefing the evening before the audit. The Lead Assessor typically sets up the audit program based on the input from the application, the client's quality manual, and a business manager who actually met with the client to set up the contract. Internal auditors are taught that preparation is at least 80% of the work and that adequate preparation is critical to the success of the audit. While the amount of time diminishes as auditors gain experience, confidence, and familiarity with the organizations policies, procedures, and standards, the lack of time for preparation forces registrars' auditors to either stay on the surface or to look for and follow familiar paths based on their experience with similar organizations. Warehouses, printing plants, and computer manufacturing lines, and similar organizations, with relatively standardized practices receive rapid, thorough audits that are typically worth the investment.

Given the time pressures, it is not surprising that document control and quality records can become the focus of an audit. They are relatively easy to identify and easy to audit. Whether or not a major nonconformity is found, it is very likely that a nonconformity is lurking somewhere, which proves that the auditor was doing his or her job. It takes substantial time and skill to audit the more subtle parts of a client's business systems which tend to be unique to each organization and which tend to cross organizational boundaries - design control, configuration management, and process control. More individual contributors need to be interviewed and more documentation reviewed to ensure that these larger, cross-functional processes are effectively implemented.

It also takes substantial time and skill to conduct complete process audits and to follow the threads that tie the ISO clauses together. It is easy to determine whether management review meetings occur by examining the recorded results. But are inputs for those meetings effectively generated at the lower levels of the organization? Are all the results of preventive action reported?

For Example

One of the most common symptoms of inadequate time for planning and conduct of audits is incomplete coverage of the functions in the scope of an ISO registration. In one organization, a certificate was issued for hardware and software design and development without any interviews being conducted in the software engineering organization. In another organization, the auditors spent their time interviewing the MIS department, responsible for internal software systems support and development. Once again, the auditors did not interview any individuals involved in the creation of software product delivered to customers. In a third, already-registered organization, the auditors missed the fact that the internal audit process had come to exist only on paper. Since the last surveillance audit, no schedule had been published and no internal audits had been conducted.

Pressure to reduce time, as well as the desire to avoid internal disagreement in front of the client, apparently impact the review of audit team recommendations by the authority responsible for issuing certificates. Even when there are relatively obvious procedural and content errors, audit team decisions to recommend or not to recommend for registration and the accompanying reports are rarely, if ever, modified.

For Example

In addition to not uncovering the fairly obvious errors in scope identified above, nonconformities unrelated to the requirements of ISO 9001 pass successfully through the mandatory "home office" review process[4]. Some of these findings include statements such as:

> *"Internal audits must be carried out by employees; consultants cannot be used."*
>
> *"Documented procedures do not adequately describe the use of test tools." in an organization using standard test tools, on which the engineers have been trained.*

Because clients are unwilling to report problems, especially when the overall conclusion of an assessment is favorable, the mandatory "home office" review is critical to ensuring the integrity of the audits - especially those audits conducted by a single auditor.

A second institutional pressure on the registrars is the nature of the job. An auditor spends a substantial amount of time on the road and, in part, because of the price competition, the salary does not appear to be commensurate with a high degree of risk or sustained pressure. Faced with booming demand for ISO registration, registrars must draw from an available labor pool that includes a large number of individuals who have come out of the Quality Control world and who are now called on to assess the effectiveness of increasingly complex and esoteric Quality-Assurance-based practices. While the auditor should not bring preconceived notions to an assessment, the auditor must bring sufficient experience, knowledge, and time to the audit to compare the documented policies, procedures, and standards with the observed practices and the requirements of ISO 9001.

For Example

In assessing on-line documentation, auditors haven taken one of two positions.

In the first case, the auditors trust the system implicitly since it is automated and overlook such things as the new complexities of notification of change. Ensuring that everyone has access to the on-line system and that everyone know that printed copies are for convenience only, is of far less significance than figuring out how to make all affected individuals aware of a change in a document, for example, a procedure, that they have not referred to since they were trained.

In the second case, the auditors mistrust the on-line system since the documents are electronic and intangible. The result is found in systems where one printed, signed copy is retained in a file drawer, where copies printed from the system say "uncontrolled copy" in the footer on every page, and where suggestions to hook up card readers to control access to desk-top computer systems are taken seriously.

Similar problems occur in determining how an organization's Configuration Management practices apply to the requirements associated with software testing - from recording results, to automatically reporting problems for corrective action, to controlling the case data associated with the tests.

Another institutional problem facing registrars is associated with conflict of interest. There is no doubt that accreditation bodies and registrars must adhere to the requirements expressed in ISO/IEC Guides 61 and 62[5] to prevent the unscrupulous registrar from tying consulting services and accreditation into a single, profitable package. This opportunity for conflict of interest extends even to the ubiquitous preassessment. Unfortunately, because ISO has not taken charge of the registration process and because the implementation guidance is inadequate, the registrars have become the interpreters of the standard and the best source of guidance for implementation techniques. In fact, a great source of frustration for registrars' auditors is that they must remain essentially silent and neutral when they see their clients implementing outrageous, unnecessary, illogical, wasteful practices in the name of ISO registration. It is difficult, if not impossible, to draw a line between training and consulting. One of the folklore guarantees for obtaining registration on the first try is to find a registrar who also offers training and send a large number of people to a number of the courses. You'll find out all the "inside" information and understand the auditors' perspective.

While Guide 62 implicitly allows an organization to offer both training and registration services[6], QS9000 explicitly does not.

For Example

An example of the vacuum that registrars must fill in providing guidance is demonstrated by the lack of any standard definitions for "design" and "production". While "production" is typically interpreted as ISO 9001's code for Manufacturing, in software, "production" encompasses replication, packaging, and distribution. Inventory (e.g., software product) is actually built by Engineering and transferred intact to production, which pulls finished goods from an endless "electronic" shelf.

In ISO 9001, "design" refers to all of the activities between the signed contract and the delivery of the product information to Manufacturing for production in quantity. "Design" in software development environments typically refers to specific activities that occur between the receipt of requirements and the development of a detailed specification or the beginning of coding.

With software prototyping and Computer-Aided Software Engineering (CASE) tools it is not clear where one phase stops and another begins. This is recognized as part of the distinct nature of software in the guidance offered by ISO in ISO/IEC 9000-1:1994[7] Paragraph , 7.4 which states that:

> *The process of development, supply, and maintenance of software is different from that of most other types of industrial products in that there is no distinct manufacturing phase. Software does not "wear out" and, consequently, quality activities during the design phase are of paramount importance to the final quality of the product."*

The result of a lack of standard guidance is the proliferation of programs like the UK's TickIT[8] program that add local guidance to the guidance in ISO 9000-3[9].

Another area of confusion in defining limits on conflict of interest is found on the Internet. There are numerous postings offering "advice" signed by an individual who might be identified as the Regional Manager, Lead Assessor, Registrar X. There is no way of distinguishing whether the advice offered is "official". Compounding this, there are many times exhortations to buy my book for more information. For informed individuals reading such messages, the inherent potential for conflict of interest may raise questions about the integrity of the registration process.

Clients

While clients contribute to the pressure to reduce costs, several inherent aspects of human nature and client attitudes conspire to create additional challenges for registrars.

A certain number of organizations work to please the registrar. Since the standard does not prescribe solutions and since the standard is difficult to interpret outside Manufacturing environments, these organizations turn to the large body of folklore that surrounds ISO registrars and registration. In anticipation of what they think the registrar's auditors will look for, these organizations put business systems in place that have no impact on - or that impede - the ability to deliver quality product. Convoluted, arcane procedures in the ubiquitous "ISO format" describe processes that are relatively straight forward and well understood. Purchasing contracts are written to require that all subcontractors be ISO registered.

Another manifestation of pleasing the registrar is found in organizations that take the registrar's auditors findings as absolute. The management representative or the audit guides don't object or

question - especially if the finding is classified as a minor nonconformity and it does not affect the overall compliance with the requirements of ISO 9001[10]. Out of a mistaken belief that filing a complaint would have negative repercussions on the next audit, organizations have considered not reporting sexual harassment and other inappropriate behavior to the registrar.

In some cases, it is difficult for the auditor to avoid some type of response behind closed doors when the client management representative asks, pleads, or begs for answers and consultation - especially when he or she wants to do a good job and obviously received bad advice in the past.

A small number of organizations cheat and are not caught - by the registrar. In one case, an organization has had the laboratory technicians take home all uncalibrated test equipment for the duration of the audit. Fortunately, such organizations tend to take care of themselves. They will lose their certificate because they are not committed to the investment necessary to maintain it. Once they realize the cost and stress of cheating, they will find out how to do it right. Or they will go out of business, because of the wasted effort or, more likely, because ISO registration is not the only place they cut corners.

Conclusions and recommendations

The primary solution for many of the problems associated with ISO registration lies with the client organizations. Each organization seeking registration must develop a thorough understanding of the standard and how it applies to the organization's business practices. A focus of our educational work is in promoting the concept that ISO 9001 is a minimum, common sense framework. If an activity doesn't make sense for your business, if an activity doesn't deliver a bottom-line benefit - don't do it until you can make sense of it. With this level of understanding, a management representative or guide can work with the registrar to eliminate misinterpretation, confusion, and inefficiency from the audit process.

Of equal importance, the International Organization for Standardization must ensure that the standards, themselves, don't contribute to the confusion, requiring the use of consultants who apply yet another level of interpretation. Guidance documents must be written in such a way that the implementor can assemble all of the relevant information (guidance and related standards) for a clause in the model standards - ISO 9001, ISO 9002, and ISO 9003.

Relationships with other international standards, like ISO 15504 **Standard for Software Process Assessment** (known as SPICE) and ISO 12207, **Information Technology - Software Life Cycle Processes**, should be coordinated and defined - even though they may have been developed by different subcommittees. This is not a trivial exercise when standards are written by committees of volunteers that meet on a quarterly basis, with requirements defined by other committees that meet annually. The value of and need for this coordination becomes readily apparent when members of the software engineering community start to refer to ISO 12207 and ISO 15504 as alternatives to ISO 9001.

For Example

From 1991 to 1997, it has been virtually impossible for the reader to trace the complete relationship between ISO 9001 and ISO 9000-3. This traceability problem was exacerbated by the four-year lag in updating ISO 9000-3:1991 to conform to ISO 9001:1994. From 1994 to 1998, ISO 9000-3 provided guidance on an obsolete version of ISO 9001.

While the revision of ISO 9000-3 issued in 1998 appears to address this problem by adopting the outline structure of ISO 9001, key information, like how "design" and "production" relate to software engineering, continues to be overlooked in the complete set of ISO standards and vocabularies. In addition, although ISO 9000-3:1998 was to introduce no content changes, changes were introduced that contradict the body of accepted usage and other guidance documents. In particular, while Paragraph 4.9 in ISO 9001 effectively applies to software development (e.g., Paragraph 4.9 (d) calls for project management and (f) calls for applicable coding standards), ISO 9000-3:1998 suggests that Paragraph 4.9 applies only to software replication.

An additional responsibility, which is just now being addressed, is for the International Organization for Standardization to take a more definitive role in the registration process by issuing standards for accreditation bodies and for registrars.

Guide 62, produced by the ISO Council Committee on Conformity Assessment (CASCO)[11], defines requirements for registrars that are nearly equivalent to those contained in ISO 9002, which is the most appropriate standard to apply to an

organization offering a service. Guide 61 defines requirements for accreditation bodies that are also nearly equivalent to those contained in ISO 9002 with the registrars defined as the suppliers.

In both Guides, the most notable deficiency is the lack of explicit requirements for project/process management and the application of statistical techniques. Both guides require reporting on performance, which may be construed as implying some measurement.

After an extended period of evaluation, the International Organization for Standardization has shut down its own committee, Quality System Assessment Recognition (QSAR)[12], and delegated responsibility for ensuring adherence to Guides 61 and 62 to the International Accreditation Forum (IAF), an association of accreditation bodies. Systematic assessment of registrars' activities by accreditation bodies, systematic peer assessment among accreditation bodies, and defined avenues of communication of "customer complaints" among suppliers, registrars, and accreditation bodies will provide necessary support for registrars who remain responsible for the quality of their services.

The auditors' responsibilities? To continue to do the best possible job.

Bibliography

[BSI1] British Standards Institution, **A Guide to Software Quality System Construction and Certification using ISO 9001:1994**, DISC TickIT Office, 389 Chiswick High Road, London W4 4AL, United Kingdom, 1995

[ISO1] International Organization for Standardization, **ISO/IEC Guide 61, General Requirements for Assessment and Accreditation of Certification/ Registration Bodies**, First edition, 1996, Geneva

[ISO2] International Organization for Standardization, **ISO/IEC Guide 62, General Requirements for Bodies Operating Assessment and Certification/Registration of Quality Systems**, First Edition, 1996, Geneva

[ISO3] American Society for Quality (Control), **ANSI/ASQC Q9000-1:1994, Quality Management and Quality Assurance Standards - Guidelines for Selection and Use**, August 1, 1994, Milwaukee

[ISO4] American Society for Quality (Control), **ANSI/ISO/ ASQC Q9000-3:1991, Quality Management and Quality Assurance Standards - Guidelines for the Application of ANSI/ISO/ASQC Q9001 to the Development, Supply, and Maintenance of Software**, November 15, 1995, Milwaukee

[PEA1] Robert Peach, ed., **The ISO 9000 Handbook**, 2nd edition, 1995, Irwin Publishing,

About the Authors

William J. Deibler II has an MS in Computer Science and 20 years experience in the computer industry, primarily in the areas of software and systems development, quality assurance, and testing. Bill has extensive experience in managing and implementing CMM- and ISO 9001-based process improvement in software and hardware engineering environments.

Robert Bamford has an MA in mathematics, and has managed training development, technical publications, professional services, and third-party software development. His over 30 years of experience include the implementation of a Crosby-based Total Quality Management System, facilitating quality courses, managing education teams, and serving on a corporate quality council. Bob and Bill are the principals of SSQC.

Since 1990, SSQC has specialized in supporting organizations in Software Engineering Life Cycle Definition, Software Quality Assurance and Testing, Business Process Reengineering, ISO 9000 Registration and CMM implementation. Bob and Bill have developed and published numerous courses, auditing and assessment tools, research papers, and articles on interpreting and applying the ISO 9000 standards and guidelines and the SEI Capability Maturity Model for Software. They are active United States TAG members in the ISO/IEC JTC1 SC7 - Software Engineering Standards subcommittee which is responsible for the development and maintenance of ISO 12207 and ISO 15504 (SPICE).

Their software development clients have successfully achieved ISO registration and advanced maturity levels.

They can be contacted at:

Software Systems Quality Consulting
2269 Sunny Vista Drive, San Jose CA 95128
Internet: ssqc@concentric.net, http://www.ssqc.com
Voice 408-985-4476, FAX 408-248-7772

Footnotes

1 "The SAC Route", memorandum from Hewlett Packard to "Companies Affected by ISO 9000", dated August 9, 1995, including extensive lists of registrars (ABS, BVQI, KEMA, Lloyds, etc.) and other companies (Motorola, Matsushita, Cisco, Novell, Digital, Varian, Sun, Bausch and Lomb, National Semiconductor, etc.) who are described as supporting the Supplier Audit Confirmation (SAC) proposal

2 "ISO 9000 registration costs falling", **Quality**, November 1995, Volume 34, page 10

[3] In this context, the principle international documents governing ISO registration are ISO/IEC Guide 61 ([ISO1]) and ISO/IEC Guide 62 ([ISO2]). It should be noted that the UK national program for ISO 9001 registration for software, TickIT, does prescribe contact hours. For information on TickIT, see [BSI1].

[4] [ISO2] Paragraph 3.5.1 states that "Those who make the certification/registration decision shall not have participated in the audit."

[5] [ISO1] and [ISO2]

[6] [ISO2] Paragraph 2.2.2 (o) lists three specific categories of activities in which an accredited registrar cannot engage: (1) services that compete with the clients it registers; (2) consulting services to obtain or maintain registration; (3) services to design, implement, or maintain quality systems. Other services are allowed as long as they do not "compromise confidentiality or the objectivity or impartiality of the certification/registration process and decisions."
In addition, [ISO2] Paragraph 2.2.3.2 (f) requires that audit team members "be free of any interest that might cause team members to act in other than an impartial or nondiscriminatory manner," which includes, as an example, having provided consulting services to the auditee in areas "which compromise the certification/registration process and decision."

[7] [ISO3]

[8] [BSI1]

[9] [ISO4]

[10] This desire to avoid complaining is echoed in a letter identified as 081-943-6168, 10 September 1993 from NAMAS to all registrars. This letter reveals that NAMAS, which certifies laboratories that calibrate inspection measuring and test devices, reports that it has been receiving a "growing number of complaints ... concerning the way in which assessors implement the requirements for traceability of measurement in BS5750 [ISO 9001]. It may be that these complaints do not apply to any of your assessors but because the complainants normally refuse to divulge which certification body [registrar] is involved ..."

[11] [PEA1], page 392.

[12] "Current and Potential Role of QSAR", **Quality Systems Update**, Volume 5, Number 6, June 1995, page 5; "ISO Decides Fate of QSAR", **Compliance Engineering Newswatch, July/August 1997** (http://www.ce-mag.com/isojul.html)

Abstract:
Hybrid Multi-Model Assessment (HM^2) - When the CMM Meets ISO 9001

Robert C. Bamford and William J. Deibler II
Software Systems Quality Consulting
2269 Sunny Vista Drive, San Jose CA 95128
Internet ssqc@concentric.net World-Wide-Web http://www.ssqc.com
Voice 408-985-4476 FAX 408-248-7772

Faced with governments' transitioning to commercial standards and with business pressures to expand into new lines of business, many software engineering organizations are faced with adapting CMM® - and ISO 9001-based systems for compliance with the other model. At the same time, many small and medium-sized software engineering organizations are exploring methods to exploit these models - ISO 9001 and the CMM - for process definition and improvement. While significant work has been done in defining methods for implementing each model, there is a lack of cost-effective tools and methods to evaluate and compare the models for selecting the most appropriate model or for planning the transition between the models.

This paper outlines a strategy and methods for employing formal appraisals to determine which model - or which elements from either model - offer the most value for a particular software engineering organization. The paper is illustrated with examples from the authors' experiences in guiding software engineering organizations in examining and selecting the most appropriate model for software development.

Keywords

ISO 9001, CMM, Assessment, Process Assessment, Process Improvement, Comparison, Process, Standards, Software, Software Development

Hybrid Multi-Model Assessment (HM^2) - When the CMM Meets ISO 9001

Faced with governments transitioning to commercial standards and responding to business pressures to expand into new lines of business, many software engineering organizations are faced with adapting their CMM-based systems for ISO 9001 compliance. At the same time, many small and medium-sized software engineering organizations are exploring methods to exploit these models - ISO 9001 and the CMM - for process definition and improvement. In the US, CMM-to-ISO is emerging as the prevalent transition for government organizations and their suppliers. In Europe, because of the early adoption of ISO 9001 and ISO 9000-3,

® CMM is registered in the U.S. Patent and Trademark Office.
℠ Capability Maturity Model is a registered service mark of Carnegie Mellon University.

there is greater interest by commercial software development organizations in the transition from ISO 9001 to the CMM[1].

In response to this industry need, a number of articles and conference presentations[2], published since 1992, have laid a foundation for comparing the requirements of the two models. These articles and presentations provide background information useful to individuals preparing to:

- Plan a transition between models.
- Identify the most appropriate model.
- Implement the most effective and efficient combination of elements from both models.

The balance of this article presents steps such people can take to translate this generic background information into a detailed, specific action plan for their organization.

Where to Start

At the beginning of the selection or transition process, there are typically champions for both models and a small group which is responsible for selecting a model. Whether the champions participate in the group is determined by the company culture and the willingness of the champions to participate in a process that has an uncertain outcome.

The **first step** in the selection or transition process is to explore and assimilate enough of the large body of knowledge to overcome initial resistance to the language and structure of ISO 9001 and ISO 9000-3[3] or to the sheer size of the CMM - even when the focus is only on Levels 2 and 3. For organizations that persevere, the outcome of this phase is typically an understanding that both models:

- Can be the basis for effective process improvement in any software engineering organization.
- Are flexible and in principle and in practice support any software development lifecycle[4].
- Are extremely susceptible to problems introduced in the implementation (excessive bureaucracy, inflexible life cycle definitions, over documentation, lack of management and/or engineer buy-in, etc.).
- Require executive management commitment.
- Require continuing organization-wide support.
- Include regular appraisal (audits or assessments) to ensure the effective implementation and continuing relevance of the defined processes.
- Are not equivalent to each other, although ISO 9001 compliance and Level 3 CMM maturity are similar – though not equivalent[5].
- Include requirements that can be selectively implemented to satisfy the requirements of the other model. For example:
  - ISO 9001 clause 4.18, Training, maps to the requirements outlined in the Level 3 KPA, Training Program.
  - The key practices described in the Level 3 KPA, Peer Reviews, can satisfy the requirements of ISO 9001 clause 4.4.6, Design reviews[6].

Formal Appraisal

The **second step** in the selection or transition process is a formal appraisal, which creates a bridge between the available information and the impact of a model on the practices in a particular software engineering organization. The formal appraisal is a continuing opportunity to educate the organization about the content of the models, to allay concerns about bureaucracy and change, and to reinforce the credibility of the models and the appraisal process. Experience with the appraisal process also serves as a valuable input to the model selection.

An appraisal is also a necessary first step in planning the introduction and implementation of any new system or process.

A review of the literature reveals a number of well-defined, formal appraisal methods associated with ISO 9001 and the CMM: the ISO 9001 registration audit or pre-assessment, the Software Process Assessment (SPA), and the family of CMM-based appraisals (CBAs), which includes the Software Capability Evaluation (SCE), the CBA for Internal Process Improvement (CBA-IPI), and Interim Profiles.

ISO Registration Audits

While the ISO 9001 audit process originally had little associated documentation, it has been well understood and consistently practiced by registrars:

1. A scope is selected
2. The quality manual and supporting policy and procedural documents are reviewed
3. A plan and schedule are prepared
4. People from all levels of the organization within the selected scope are interviewed, typically at their work stations
5. The results of the appraisal or audit are consolidated into a presentation and report, which includes a recommendation for or against registration and which is presented to management of the audited organization
6. The report, including a recommendation for or against registration, is reviewed and approved by registrar personnel who were not involved in the audit

The registrars' initial and surveillance audit methods have shaped the internal audits (ISO 9001, 4.17), which are periodic, mandatory self-assessments to ensure on-going compliance *with planned arrangements and to determine the effectiveness of the quality system.*[7]

The internal audits are a critical input to the management review process and to the management representative, who is responsible for *ensuring that a quality system is established, implemented, and maintained in accordance with this International Standard.*[8]

With the publication of **ISO/IEC Guide 61**[9] and **ISO/IEC Guide 62**[10] in 1996, the accreditation and the registration processes have been completely defined in standards virtually equivalent to ISO 9002. With the formalization of the role of the International Accreditation Forum (IAF), an audit structure has been implemented to monitor the on-

going compliance of the accreditation bodies and registrars[11] and to ensure a baseline consistency across registrations.

Because of the experience of the assigned assessors, advance preparation tends to be minimized. Typically only the Lead Assessor reviews the quality manual, the top-level document, and selected policies and procedures. In the course of the on-site interviews, auditors examine process and product documentation in detail to ensure compliance with planned arrangements. ISO 9001's unique requirement that the quality policy be *understood, implemented, and maintained at all levels of the organization*[12] has led to the practice of interviewing a sample of some 15% to 20% of the organization. Although national programs, like TickIT, include guidance (i.e., suggestions) regarding contact hours[13], there are no international standards defining sample size and contact hours for ISO registration and surveillance audits. This has, in fact, become a problem as competition forces registrars to reduce costs.

For implementation guidance, a pre-assessment by a registrar falls short. While the registrar will be as thorough as the implementor requires, the results can only define observed nonconformity. Recommendations for corrective action would be a form of consulting, which is forbidden by accreditation bodies and **ISO/IEC Guide 62** as a conflict of interest that would contravene the impartiality[14] of the assessors, who would have a vested interest in their recommended solutions.[15] As a result, a large, independent consulting community exists to provide detailed, collaborative audits, similar to the CBA-IPI (pre-assessments, gap analyses, diagnostic audits, etc.) to support implementation and action planning.

CMM Appraisal Methods

It is only since 1993 that methods like the SCE have been published. Before 1993, *detailed information about the SCE Method was available only through SCE team training, which was available only to government teams.*[16] The Software Engineering Institute has also published a standard, the **CMM Appraisal Framework** (CAF), against which any CMM-based appraisal method can be evaluated[17] and with which it has been stated the SEI-provided methods will comply. From comments in [Se2][18] and [Se3][19], and from a wealth of anecdotal data provided by numerous people, it appears the CMM appraisal process has changed dramatically. In its initial form, the appraisal process did not rely on objective evidence or on systematic techniques for obtaining evidence; the process incorporated interviews with project managers and free-form discussions with functional area representatives. The CMM appraisal process has evolved to its present form, engaging individuals from all levels of a development organization, including middle management, and incorporating more traditional auditing methods and systematic techniques for corroboration.

This change in method, in terms of the CBA-IPI, is presented in the SEI lead assessor training[20]. Key improvements are identified as:

- *Documentation review*. The CBA-IPI incorporates more extensive documentation review.
- *Interviews.* The CBA-IPI includes individual interviews of project leaders, structured group interviews of middle managers, and structured group interviews of functional area representatives (FARs, e.g., individual contributors). Previously:

  - Middle managers were not necessarily interviewed.
  - Functional area representative interviews were free-form discussions.
- *Data consolidation*. The CBA-IPI incorporates more systematic analysis and consolidation of the data from the interviews and documentation review.

The CBA-IPI requires an extensive commitment from the software engineering organization undergoing the assessment. An integral part of the CBA-IPI, as a CAF-compliant method, is training for a team of members from the sponsoring organization, who participate in the assessment and who are positioned to participate in the follow-up process improvement activities. Although this commitment of resources and money has a high potential rate of return, it is typically more than can be justified by an organization investigating whether the CMM is applicable.

Key Differences between the Appraisal Methods

Both the ISO audit and the CBA-IPI produce objective results to support process improvement. Although the two models differ in content and scope[21], both appraisal methods require that the organization have completed the implementation activities necessary to define and institutionalize - implement across the organization - effective processes. The appraisal confirms the success (or continuing success) of the implementation by gathering and analyzing objective evidence. Both methods require that management invest whatever resources are required to address issues identified in the assessment or audit. There are, however, a number of key differences between the methods.

Difference 1: Level of Involvement

The most prominent difference between the ISO audit and the CBA-IPI is the level of involvement required of the organization. To position the organization to understand and take action on the findings, the CBA-IPI requires *more extensive* team participation than the comparable ISO audit. In a CBA-IPI, one or more members of the organization serve on the assessment team. The comparable ISO audit relies on a representative of the audited organization (the guide), who accompanies the ISO auditor and depends on detailed, written findings reviewed with the audited organization's management as the last step in the on-site portion of the audit.[22]

Difference 2: Interview Methods

A second difference is in the way in which input is gathered. In an ISO audit, interviews are typically conducted in or near the interviewee's workplace and are attended by the auditor, interviewee, and the guide. In the CBA-IPI, while project managers are interviewed individually, small groups of middle managers and small groups of functional-area representatives (FARs) meet with panels of assessors, chaired by the Lead Assessor or another member of the assessment team. The group approach is based on the principle that individual contributors will be more willing to speak frankly when they are part of a small group.

Difference 3: Reporting Results

The third difference is in the way in which findings are reported at the conclusion of the assessment or audit. As described, above, in conjunction with *Difference 1*, detailed,

written findings are provided to and reviewed with the organization's management as the last step in the on-site portion of the ISO audit, typically on the day following the last interviews. Findings are expressed in terms of a specific clause of ISO 9001 and include detailed information about the nonconformity and are classified as major or minor. The only global finding is binary. Based on the identified nonconformities, the audit team states whether it recommends that the audited organization be registered or retain its registration.

The CMM assessment concludes with presentations of draft and final findings, defining at a minimum the organization's strengths and weaknesses at the Key Process Area level. Detailed information is transferred through the members of the organization who participate on the assessment team. In addition, a written report may be purchased as an option, but typically it is not available for an additional 4 to 6 weeks and it does not necessarily include any significant information beyond that which was published in the final findings presentation.

Difference 4: Role of the Lead Assessor and Assessment Team Make-up

In the CBA-IPI, preparation spans at least two weeks. Team members may not have had any prior assessment experience or direct experience with the CMM. In the first week, the Lead Assessor trains the team. In the second week, documentation is reviewed and the assessment checklists and schedules are prepared. The Lead Assessor conducts key interviews, leads team interviews, and facilitates team discussions that reach consensus on findings. The Lead Assessor is also the CMM expert, guiding the team in their interpretation of whether observed practices satisfy the requirements of the CMM. This latter function becomes increasingly critical as the CMM is applied to commercial organizations, providing products to customers outside the Department of Defense community. The problem of interpretation is exacerbated as commercial organizations go beyond the experience provided by many Department of Defense software providers who have been able to rely on CMM-compatible Military Standards (e.g., MIL-STD 2167A, MIL-STD 498) to completely define the implementation.

The CMM recognizes this problem, and states that

> *Organizations using the key practices should be aware of these conventions* [in expressing the key practices] *and map them appropriately to their own organization, project, and business environment.*[23]

To mitigate the potential problem of auditor preconception and to enhance the maintainability of the implemented quality system, ISO 9001 requires that the organization create a quality manual, which documents how the requirements of the standard are addressed.

Difference 5: Who Appraises the Appraiser? Ensuring the Quality of the Assessments

The ISO registration process includes four levels of quality assurance:

- ISO lead auditors must complete an approved course.
- Performance of ISO lead auditors is monitored by the registrars.
- Performance of registrars is monitored by accreditation bodies.
- Performance of accreditation bodies is monitored by the IAF.

The CBA-IPI infrastructure is less extensive; the SEI is just beginning to define standards, guidelines, and a mechanism for proactive monitoring of the quality of assessments. People with extensive experience in software development, including participation in two CBA-IPIs, can become registered Lead Assessors by completing the SEI curriculum and completing an assessment observed by a Lead Assessor. Only assessments conducted by SEI-authorized Lead Assessors can be recorded at the SEI as "SEI-recognized" assessments. The registry of CBA-IPI Lead Assessors is maintained by the SEI.

Selecting an Appraisal Method

For the *CMM Level 3-compliant organization* or the *ISO 9001-compliant organization*, one or two individuals experienced in both models and familiar with the organization[24] can convert two available sources of information into an accurate benchmark of the position of the organization with respect to the other model. The first source of information is the organization's library of presentations and reports from recent appraisals (ISO 9001: internal and registrar's audits; CMM: CBA-IPI, SCE, and SQA audits and reviews). The results of these assessments record the degree of "institutionalization" (CMM) or "effective implementation" (ISO 9001) of the required practices. The second source of information is the organization's set of documented policies, procedures, and standards, which describe in detail how the organization should operate.

Omissions identified in this conversion are addressed by updating the existing set of process documents. A by-product of the conversion is a mapping of the organization's policies, procedures, and standards to the requirements of the other model.

For *CMM Level 2-compliant organizations*, the most effective strategy depends on whether the organization is committed to the CMM and how well it is positioned for Level 3 (i. e., how much ground work has been done, exceeding the requirements of Level 2). If the CMM is well-established and the organization is well positioned for Level 3, the recommended strategy is to continue to Level 3 and to follow the conversion strategy outlined above for a CMM Level 3-compliant organization. An alternative for a CMM Level 2-compliant organization is to commission an ISO pre-assessment and proceed with an exclusive ISO focus. To the extent that the Level 2 organization has adopted standard processes across projects (a Level 3 requirement), work products and processes will carry forward to ISO.

Organizations that are not committed to either model face the greatest challenge – and opportunity – in designing an appraisal strategy. The most straight-forward approach, to undertake separate ISO and CMM appraisals, is typically too expensive, too time consuming, and too confusing. The *hybrid model* and its associated assessment method offer a viable alternative for the currently uncommitted organization.

Hybrid Models

A number of hybrid models exist, including Bootstrap[25], and Trillium. Although these models incorporate ISO 9001, ISO 9000-3, the Malcolm Baldrige National Quality Award (MBNQA) criteria, the CMM, and various other standards, they do not answer the needs of the organization investigating ISO 9001 and the CMM. The outputs of the appraisals associated with these hybrid models are unique to the model and do not facilitate translation

among the various source models, with which these proprietary hybrid models are intended to compete.

Hybrid Multi-Model Assessment (HM^2)

The emergence of a general agreement on the relationship between the requirements of the CMM and ISO 9001 implicitly defines another hybrid model, the union of the two models, and forms the basis for a set of assessment tools and report templates that address both the goals of the CMM's KPAs and the requirements of ISO 9001.

The tools associated with the hybrid assessment address both the requirements the two models share, like documented procedures for planning, and those requirements that lie outside the intersection of the two models. For example, ISO requirements for record retention, technical support, packaging and distribution, and software maintenance, are not addressed in the CMM and the CMM's detailed requirements for planning (size and cost) are not addressed in ISO 9001.

In the method adopted by the authors of this paper, a hybrid multi-model assessment is conducted by a small team of experienced, independent assessors following standard audit practices reflected both in ISO 9001 registration audits and in the SEI CAF:

- Scope selection
- Off-site document review
- On-site interviews
- Report preparation and delivery of findings

By building on the well-defined relationship between the clauses of ISO 9001 and the Level 2 and Level 3 CMM KPAs, as described in Figure 1, the results of a single set of comprehensive interviews can be presented from both an ISO 9001 and a CMM perspective. To achieve the maximum impact from the report, the findings are presented twice in separate sections. Each section of the report is organized around one of the models. Each time the finding is presented, it includes detailed recommendations for action planning and points to the sections of the other model that address similar or identical requirements. To support overall action planning, the report concludes with a single set of priorities for implementation that define the assessors view of a logical path through the most important of the detailed findings.

Level 2 KPAs	ISO 9001
Requirements management	4.3, 4.4
Software Project Planning	4.1, 4.3, 4.4, 4.9.
Software Project Tracking and Oversight	4.4, 4.9
Software Subcontract Management	4.6
Software Quality Assurance	4.4, 4.17
Software Configuration Management	4.4, 4.5, 4.7, 4.8, 4.9, 4.12, 4.13, 4.14, 4.15

Level 3 KPAs	ISO 9001
Organization Process Focus	4.9, 4.14
Organization Process Definition	4.2
Training Program	4.18
Integrated Software Management	4.4, 4.9
Software Product Engineering	4.4, 4.10
Intergroup Coordination	4.4
Peer Reviews	4.4

Figure 1 Relationship between the Level 2 and 3 KPAs and ISO 9001 clauses

Considerations for Planning Hybrid Multi-Model Assessments

A hybrid multi-model assessment can be completed by a small, independent team in approximately 140% of the time required for an ISO 9001 registration assessment. Based on anecdotal information and on the authors' experience in a number of CMM-based process assessments[26], a hybrid multi-model assessment can be completed with 20% - 30% of the time and resource for an initial CBA-IPI[27].

Although the proposed hybrid multi-model assessment does not include the team training that prepares the organization to act on the results of the assessment, there is implicit training in the interview process, especially when a value-added approach is used[28]. Employing collaborative assessment techniques, the hybrid multi-model assessment method is sponsored by the assessed organization and allows time in the interviews, in the entry meetings, and in general communications to solicit and deal with specific issues and concerns that might otherwise influence responses. The communications and interviews can be structured to reinforce three key principles:

- The purpose of the appraisal is to gather information to measure the completeness of the organization's actual practices (what you do, not what you are supposed to do) with respect to the models.
- The purpose of measuring is to improve the organization's ability to develop and deliver software to its customers.
- Any changes or new processes introduced as a result of the appraisal must support the way the organization does or wants to do business. This principle leads to at least three corollaries.
  - Members of the organization will be required to adopt the defined processes – and alert the appropriate people if there are problems with the documented procedures.
  - The models – ISO 9001 and the CMM - will be used to determine only what has to be done – not how it will be done.
  - Achieving compliance with the chosen model will be a by-product of implementing effective and efficient processes that support the organization's business goals and objectives and that meet the needs of its employees.

The report derived from the assessment reinforces the similarity between the sets of requirements both models place on a software engineering organization. The consolidated

report adopts the value-added audit technique of including recommended actions, allows the champions of each particular model, who are frequently well-respected and influential engineers, to consider the other model and to leverage their experience. The recommended actions also facilitate post-assessment action planning.

After the Assessment

The assessments described in this paper provide the information required to select a model – or combination of models – and to prepare a plan for effective process definition, implementation, and improvement. Achieving compliance with the chosen model is a by-product of acting on the assessment findings.

Whether the organization's management chooses to go beyond compliance to become ISO registered or to complete a formal CBA-IPI or SCE, it is critical that management acknowledge and respond systematically to the assessment findings. In the context of ISO 9001 and CMM Levels 2 and 3, the assessment findings define problems that are adversely affecting the organization's ability to perform – that are costing the organization time and money and creating unnecessary stress. If management fails to respond, the members of the organization will inevitably draw the obvious conclusions about management's commitment to its customers and to its employees. In fact, it would have been preferable not to have conducted the appraisal.

If management responds positively, there is no guarantee of success, but that positive response may launch the organization on a path to software process improvement that will lead to increased efficiency, capability, and competitive strength.

Bibliography

[Ba1] R. C. Bamford and W. J. Deibler, **A Detailed Comparison of the SEI Software Maturity Levels and Technology Stages to the Requirements for ISO 9001 Registration**, Software Systems Quality Consulting, San Jose, Calif., 1992.

[Ba2] R.C. Bamford and W. J. Deibler, *Exploring the Relationship between ISO 9001 and the SEI Capability Maturity Model for Software Engineering Organizations*, **Conference Proceedings, 1993 International Conference on Software Quality**, (held at Lake Tahoe, Oct. 4 - 6, 1993), The Software Division of the American Society for Quality Control, page 199

[Ba3] R.C. Bamford and W. J. Deibler, *Comparing, contrasting ISO 9001 and the SEI Capability Maturity Model*, **COMPUTER**, October 1993, Vol. 26, No. 10, IEEE Computer Society, page 68

[Gi1] Gilbert Associates (Europe) Limited, **TickIT Auditor' Training Course**, Issue 1, April 1992

[Ha1] Volkmar Hasse, Richard Messnarz, Robert M. Cachia, **Software Process Improvement by Measurement, BOOTSTRAP/ESPRIT Project 5441**

[In1] International Organization for Standardization, **ISO 9001, Quality Systems - Model for quality assurance in design/development, production, installation, and servicing, ISO 9001**, International Organization for Standardization, Geneva Switzerland, 1987, revised 1994

[In2] International Organization for Standardization, **ISO 9000-3, Guidelines for the Application of ISO 9001 to the development, supply and maintenance of software**, 1991-06-01, International Organization for Standardization, Geneva Switzerland, 1991

[In3] International Organization for Standardization, **ISO/IEC Guide 61, General Requirements for Assessment and Accreditation of Certification/ Registration Bodies**, First edition, 1996, Geneva

[In4] International Organization for Standardization, **ISO/IEC Guide 62, General Requirements for Bodies Operating Assessment and Certification/ Registration of Quality Systems**, First Edition, 1996, Geneva

[Ka1] Kasse, Tim, Wihalm Josef, *The Long Way to CMM Level 4*, **Proceedings of the First World Congress for Software Quality**, June 1995, ASQC

[Ku1] Pasi Kuvaja et al., **Software Process Assessment & Improvement - The Bootstrap Approach**, Blackwell Publishers, Oxford, UK, ISBN 0-631-19663-3, 1994

[Pa1] Paulk, Mark C., *A Detailed Comparison of ISO 9001 and the Capability Maturity Model for Software*, **Software**, January 1994

[Pa2] Mark C. Paulk et al., **Key Practices of the Capability Maturity Model, Version 1.1, CMU/SEI-93-TR-25**, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA 15213, March 1993

[Sa1] Sayle, Allan J., **Management Audits - the assessment of quality management systems**, (ISBN 0-9511739-1-X), 1989. Available from the ASQC Press

[Se1] CMM-Based Appraisal Project, **Software Capability Evaluation Version 2.0 Method Description, CMU/SEI-94-TR-6**, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1994

[Se2] Masters, Steven, Bothwell, Carol, **CMM Appraisal Framework, Version 1.0**, CMU/SEI-95-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1995

[Se3] **CBA Lead Assessor Training, Participant's Guide, Version 1.1a,** Software Engineering Institute, Carnegie Mellon University, Pittsburgh, April 1997

About the Authors

William J. Deibler II has an MS in Computer Science and 20 years experience in the computer industry, primarily in the areas of software and systems development, quality assurance, and testing. Bill has extensive experience in managing and implementing CMM- and ISO 9001-based process improvement in software and hardware engineering environments.

Robert Bamford has an MA in mathematics, and has managed training development, technical publications, professional services, and third-party software development. His over 30 years of experience include the implementation of a Crosby-based Total Quality Management System, facilitating quality courses, managing education teams, and serving on a corporate quality council. Bob and Bill are the principals of SSQC.

Since 1990, SSQC has specialized in supporting organizations in Software Engineering Life Cycle Definition, Software Quality Assurance and Testing, Business Process Reengineering, ISO 9000 Registration and CMM implementation. Bob and Bill have developed and published numerous courses, auditing and assessment tools, research papers, and articles on interpreting and applying the ISO 9000 standards and guidelines and the SEI Capability Maturity Model for Software. They are active members of the United States Technical Advisory Group (TAG) for the ISO/IEC JTC1 SC7 - Software Engineering Standards subcommittee, which is responsible for the development and maintenance of ISO 12207 and ISO 15504 (SPICE).

Their software development clients have successfully achieved ISO registration and advanced maturity levels.

They can be contacted at:

Software Systems Quality Consulting
2269 Sunny Vista Drive, San Jose CA 95128
Internet: ssqc@concentric.net, World-Wide-Web: http://www.ssqc.com
Voice 408-985-4476, FAX 408-248-7772

Footnotes

[1] This European interest is evidenced by steadily increasing attendance numbers for the annual European Software Engineering Process Group (ESEPG) conferences sponsored by the SEI.

[2] [Ba1], [Ba2], [Ba3], [Pa1]

[3] [In2]

[4] Documents associated with both models contain explicit statements of principle regarding lifecycle independence. For ISO, [In2] Paragraph 5.0 states that the standard is "intended for application irrespective of the life-cycle model

used." For the CMM, [Pa2] Paragraph 4.3.5 states that "the key practices are not meant to limit the choice of a software life cycle. ... there is no intent to encourage or preclude the use of any particular software life cycle."

[5] [Pa1] and [Ba3]

[6] The requirements for verification assigned to the SQA function by the CMM can be satisfied by ISO 9001's internal audits.

[7] [In1] 4.17

[8] [In1] 4.1.2.3 and 4.1.3

[9] [In3]

[10] [In4]

[11] "Current and Potential Role of QSAR", **Quality Systems Update**, Volume 5, Number 6, June 1995, page 5; "ISO Decides Fate of QSAR", **Compliance Engineering Newswatch, July/August 1997** (http://www.ce-mag.com/isojul.html). At the time this paper is being written, the IAF is seeking incorporation.

[12] [In1] 4.1.1

[13] [Gi1] section 11, page 18

[14] [In4] 2.2.2 (o)

[15] It is interesting to note that, while ISO 9000 accreditation bodies do not consider training as a consulting activity, QS9000 expressly forbids training as well as consulting.

[16] [Se1] page 28

[17] [Se2]

[18] [Se2] page 25

[19] [Se3], Section LA.B, page 13

[20] [Se3], Section LA.B, page 13

[21] [Ba1], [Ba2], and [Ba3] contain the background for the authors' conclusion that the two models produce comparable results when adopted by a software engineering organization. One of the key differences is in the ability to extend ISO 9001 to all parts of an organization (e.g., Marketing, Sales, Program Management, Systems Engineering, System Test).

[22] The results presented by the audit team are submitted to the registrar's "home office" for review before they become official.

[23] [Pa2], Paragraph 4.1. It is of interest to note that there is an analogous statement in ISO 9001. In the Introduction, [In1] states that " ... it is not the purpose of these [standards] to enforce uniformity of quality systems. ... The design and implementation of a quality system will be influenced by the varying needs of an organization, its particular objectives, the products and services supplied, and the processes and specific practices employed."

[24] The individuals ideally were or have access to internal auditors or members of the CBA-IPI team.

[25] [Ku1] and [Ha1]

[26] The authors have participated in CBA-IPIs and conducted HM^2 assessments in organizations ranging in size from 100 to more than 800 employees. This size estimate is also consistent with [Ka1] page 15.

[27] Note that the initial CBA-IPI considers all of the Level 2 KPAs and a number of level 3 KPAs.

[28] [Sa1]

Using the Electronic Proceedings

Once again, PNSQC and ICSQ are proud to announce our electronic Proceedings on CD-ROM. On the CD-ROM, you will find most of the papers from the printed Proceedings, plus the slides from some of the presentations. We made every effort to see that the electronic and printed Proceedings match, but we were not able to get electronic copies of all the presentations.

We hope that you will enjoy this addition to the Conference. If you have any suggestions for improving the electronic Proceedings, please visit http://www.pnsqc.org/cdrom.html to give us feedback, or send email to cdrom@pnsqc.org.

Installing Acrobat Reader

The electronic Proceedings are in Adobe Acrobat format. The CD-ROM includes the free Acrobat Reader software for Macintosh, Microsoft Windows, and several of the most popular UNIX workstations.

If you do not currently have Acrobat Reader 3.01 installed, you can install it from the CD-ROM. The CD-ROM uses the industry standard ISO-9660 format. In the ACROBAT folder, find the appropriate folder for your system: Win16 (Windows 3.1), Win32 (Windows 95, 98, or NT), or UNIX. Follow the directions on the next page for your particular system.

Next Page 

UNIX

Change directories to the `unix` directory on the CD-ROM, and run the shell script `./install`. The script will automatically detect your hardware and operating system. Answer the questions to install Acrobat Reader. If the script reports that your configuration is not supported, visit http://www.adobe.com/acrobat to see if you can download a version of Acrobat Reader for your system. The CD-ROM includes versions for the following systems: SunOS, Solaris, HP-UX, IBM AIX and SGI IRIX. Acrobat Reader without the Search tool is available for DEC Alpha and Linux. Read `instguid.txt` for information about installing in a network.

Macintosh

You can download Acrobat Reader for the Macintosh from Adobe's web site: http://www.adobe.com/acrobat.

Microsoft Windows 3.1, Windows 95, Windows 98, Windows NT

Run SETUP.EXE from the CD-ROM's root directory. The setup program creates Program Manager icons to access the Proceedings. If you want, you can copy the Proceedings to your hard drive. If you don't copy the files to your hard drive, the setup program will create links to the Proceedings on CD-ROM. If you do not already have a program that can read PDF files, the setup program asks if you want to install Acrobat Reader. If you decide not to, you can install Acrobat Reader at a later date by running ACROBAT\WIN16\SETUP.EXE for Windows 3.1 or ACROBAT\WIN32\SETUP.EXE for later versions of Windows.

Next Page 

Using Acrobat Reader

After you have installed Acrobat Reader, you can use it to read and search the electronic Proceedings. On Windows, just select the 16th PNSQC-8th ICSQ icon to read the Proceedings. For the Macintosh or UNIX, start Acrobat Reader and open the file, PNSQC.PDF in the root directory of the CD-ROM. From there, you can read the electronic Proceedings, examine the table of contents to find a particular article, jump to the index to look up a topic, or use the search tool to find all the articles that include a keyword. The electronic Proceedings include a full-text index so the Acrobat Search tool is a fast and convenient way to find information on the CD-ROM.

You can print articles or slides for your own private use, but remember they are copyright to the original author. You cannot redistribute the Acrobat files or their printed copies. If you need extra copies of the printed or electronic Proceedings, please contact Pacific Agenda. An order form appears on the next page.

1998 Proceedings Order Form

Pacific Northwest Software Quality Conference
joint with
ASQ Software Division Eighth International Conference on Software Quality

To order a copy of the 1998 Proceedings, please send a check in the amount of $35.00 to:

PNSQC/Pacific Agenda
PO Box 10142
Portland, OR 97296-0142

Name_____

Affiliate_____

Mailing
Address_____

City_____

State_____

Zip_____

Daytime
phone_____