Using the Electronic Proceedings

Once again, PNSQC is proud to announce our electronic Proceedings on CD. On the CD, you will find most of the papers from the printed Proceedings, plus the slides from someof 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/hot/cdrom.htm to give us feedback, or send email to cdrom@pnsqc.org.

Adobe Acrobat Reader

The electronic Proceedings are in Adobe Acrobat format. If you do not currently have Acrobat Reader 4 installed, you can download it from Adobe's web site: http://www.adobe.com/acrobat.

Copyright

Proceedings Order Form

Pacific Northwest Software Quality Conference

Proceedings are available for the following years. Circle the year for the Proceedings that you would like to order.

1986, 1987, 1989, 1992, 1995, 1999, 2000, 2001

To order a copy of the Proceedings, please send a check in the amount of $35.00 each to:

PNSQC/Pacific Agenda
PO Box 10142
Portland, OR 97296-0142

Name _____

Affiliate _____

Mailing
Affiliate _____

City _____

State _____

Zip _____ Phone _____

NINETEENTH ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

OCTOBER 16 - 17, 2001

Oregon Convention Center
Portland, Oregon

TABLE OF CONTENTS

Management Track – October 16

Testing Track – October 17

Process Track – October 17

Management Track – October 17

Preface

Members,

Hello! I am delighted to welcome you to the 19th Annual Pacific Northwest Software Quality Conference. Our conference is the longest-running conference dedicated solely to the issues surrounding Software Quality.

With the uncertain economic situation at the time of this writing, I recognize the financial and temporal sacrifices you will need to make to attend our conference. I can assure you that our conference is one of the greatest values in the Software Quality conference realm.

PNSQC is a non-profit organization, staffed primarily by volunteers. My job this year has been made very simple because of the dedication of the committee chairs, and the wonderful volunteers among those committees. The excellent program being brought to you this year is because of the exceptional submissions from people like yourselves, and the selfless dedication of the Program Committee – lead this year by Howard Mercier.

We are pleased to welcome Sheri Pfleeger and Johanna Rothman as our keynote speakers this year. In addition, we have ***28*** papers in our program, have invited ***4*** special speakers, and have scheduled a panel discussion. Please make sure to visit our exhibitors this year - they work hard to bring you information, products, and services to help you and your respective companies do their jobs better.

In the past few months, PNSQC has sponsored several other events within the local Software Quality community. We have collaborated with other organizations to bring these events to you, and will continue to do so in the future. Please let us know if you have any ideas for joint events anywhere in the great Pacific Northwest!

I believe that all of us recognize Software Quality as very important – otherwise we wouldn't be here! I would like to challenge all of you to the following:

If you find that a subject of interest to you is underrepresented at the conference, please submit a paper next year! Writing a paper offers you an opportunity to learn more about the subjects that interest you, and become an expert in your topic. In addition, your decision to share your knowledge will have a positive impact on our conference next year, and going forward. If you don't want to submit a paper, invite your co-workers or someone else in the software development field to become a part of this conference. Next year is our 20th year – the end of the second decade of PNSQC! Help us make it a banner year.

In closing, I would like to thank all members of all the committees and Board of Directors of the conference for all their hard work and all the volunteer hours they have contributed to this conference. Special thanks as well to our conference manager, Terri Moore of Pacific Agenda. She always handles the organizational and administrative tasks very efficiently. We'd be lost without her.

If you find you like what you find, please let others know. If you find you are dissatisfied for any reason, please let us know. After all, how can we improve if we don't know there are problems?

Again, welcome, and please plan to attend the 2001 Pacific Northwest Software Quality Conference - the best value of it's kind!

Sandy Raddue
PNSQC President and Conference Chair

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Sandy Raddue – PNSQC President and Chair, Publicity Chair
Cypress Semiconductor

Paul Dittman – PNSQC Vice President, Program Co-Chair
Intel Corporation

Howard Mercier – PNSQC Secretary, Program Co-Chair
STEP Technology, Inc.

Doug Vorwaller – PNSQC Treasurer
Willamette Industries

Sue Bartlett – Keynote 2001, 2002 Chair
STEP Technology, Inc.

Rick Clements – Strategic Planning
Cypress Semiconductor

Dennis Ganoe – Web Master
Professional Data Exchange

Shauna Gonzales – Birds of a Feather
ImageBuilder Software

Bhushan Gupta – Workshop Chair
Hewlett Packard

Merle Knisely – Software Excellence Award Chair

Cindy Oubre – Volunteer Coordinator

Don White – Exhibits Chair
BidTek

CONFERENCE PLANNING COMMITTEE

Hilly Alexander
Comsys

Robert Bennett
SRC Software

Kit Bradley
PSC Scanning, Inc.

Pam Clark
Host Corporation

Kathi Harris
Intel Corporation

Connie Ishida
Tektronix, Inc.

Mark Johnson
Cadence Design Systems

Charles Knutson
Brigham Young University

Ganesh Prabhala
Intel Corporation

Eric Schnellman

Erik Simmons
Intel Corporation

Piet Van Weel

Stephanie Verzasconie
Hewlett Packard

Richard Vireday
Intel Architecture Labs

Release Criteria: Defining the Rules of the Product Release Game

Introduction

How do you know when you're done testing? How do you know when the product is ready to release? Sometimes the decision to stop testing and release seems as if someone's making deals in a smoke-filled room, or that there are rules of the game we don't know. Sometimes the release decision even seems completely arbitrary. Instead of arbitrary decisions, it's possible to come to an agreement about when the product is ready to release, and maybe when it's time to stop testing. We'll discuss how to define release criteria, and then use those criteria to decide when to release the product, and what to do under pressure when we don't meet the release criteria.

The Problem: Is the Software Ready to Release?

For any project, the big question is "is the software ready to release yet?" When is the development and testing part of the project complete? You can't know if you're ready to release, unless you know what "done" means.

Sometimes, not knowing what done means, you release before you are done. In the mid-1980's, I was a test lead working on a major operating system release, including a brand new window-based GUI. At the time, most operating systems had command line interfaces, so our customers couldn't tell us what they wanted in a GUI. We worked on the release for months, and finally released the system over many internal objections (from developers and testers), because our management decided it was time to release.

The release was a disaster. Our customers hated the way the windows worked, and wanted to get rid of the GUI altogether. We hastily planned an incremental release, fixing the obvious defects. Our customers liked the initial point release, and were thrilled about the next point release, when everything in the system finally worked in a way they could use.

We had no release criteria for the major release. Senior management mandated the release date, primarily to make revenue, and we didn't have an easy way to know if the software was good enough for out customers by the desired release date.

Other times, when you don't know what done means, you release after you've been done for a while. A few years ago, I facilitated a project retrospective for a company. The company had already released a successful client/server product, and needed to transition to the web quickly. While doing the web transition, they added a few more features to the product, and fixed a number of defects that the new customers would not tolerate. The project was on an aggressive schedule, and the VP was under serious pressure from the rest of the management team to release the product. For the last four weeks of the project, the VP interrogated everyone daily on the project team, asking if they were done yet. During the retrospective, the VP recapped what he'd thought was necessary for the

These ideas will be published later in an article in STQE magazine, scheduled for 2002.

release to be done. The developer most on the project's critical path said, "If I'd known that was all I had to do, I could have been done a month ago." The VP looked furious.

This company didn't have release criteria; they only had a release date. Because they had no way to assess how good each product component had to be, the relevant developers and testers determined each component's done-ness by themselves.

Another release problem occurs when someone who doesn't actually know what's going on decides to release the software. Recently, I consulted with a QA manager, Deb, whose test team was having trouble deciding in how much depth to test the pieces of the release. They'd received the software the previous week, had done some planning and exploratory testing, and were now having trouble allocating their time to the different pieces—they felt as if they had too little time and too much to do. I was going to help the technical leads and Deb assess risk, and figure out where to spend their scarce resources.

During our meeting, one of the testers ran in, saying, "Don't even bother. They shipped the release at noon time."

Deb asked her VP why they'd already shipped the software. The VP said, "You weren't going to be able to tell me anything about the software in less than a month, and the VP of Sales convinced our CEO that we had to ship. I was vetoed." Deb saw this as a potential opportunity, and said, "Ok, let me hire more testers." The VP said, "If I give you more people, can we release the software faster?" Deb said, "I don't know if we'll be able to release it faster, but we'll know earlier if it's good enough to release. We can make release decisions without you being vetoed."

Maybe you've run into problems like these, or other problems when deciding if the software is good enough to release:

1. You take on the role of gatekeeper, the one personally responsible for the content and quality of the software release.
2. You have a date to release, but you don't know precisely how good the software has to be.
3. You can't easily explain how much progress you've made in testing, but you know you're not done testing the software.
4. You're not involved in the decision to release the software, and you find out when your VP tells you to stop testing, because you don't need to test anymore—the product is ready to release by *fiat*.
5. Your organization makes decisions based on gut feel about the software, without sufficient data to back up your guts.
6. Some number of people can veto the current release decision, whether that decision is to release or not release.

If you're finding it difficult to make rational decisions about when to release the software, developing and using release criteria can help.

Develop Release Criteria

When you develop release criteria, first decide what's critically important to *this* project: why the company is doing the project, why the customers would buy this product. If you don't know what success for this project means, start by defining it. Then quantify how you would recognize success during the project, and gain agreement by the project team and senior management that the criteria you generated are how you will measure the "done-ness" of the release.

Define Success through Interviews

Not every project has a clear definition of success. If you're on a project in the fuzzy beginnings, you may find it easier to interview the PM and other stakeholders first, before trying to define what's important to the project, or drafting release criteria.

I define what success means by understanding:

- The problem (or problems) this project is trying to solve
- The project's requirements (what are your managers paying you to deliver? How good does it have to be? When do they want it? Are there other constraints?)
- The product's requirements (what do the customers want out of the project?)
- Planning and executing the testing portion of the project, to meet what your company is trying to sell.

You'll notice that success here has nothing to do with the number of defects found, or the number of tests defined or run. Success is about what the customers will be able to do with the product when you're done with it.

If your project manager doesn't initiate projects this way, you can learn what success means by interviewing the project manager and other managers in the organization. If you find out at the beginning that not everyone agrees on success, you won't be able to agree on what makes the software ready to release. Better to discover that disconnect at the project's beginning, before you plan your approach to testing.

If you do find a disconnect about what success means, make sure the PM knows. If you're the PM, make sure your project sponsor knows. Then, get help reconnecting everyone's definition of success. Get a professional facilitator to help with this reconnection.

Use Context Free Questions

I use context free questions [Gause and Weinberg] as my primary interviewing technique for discovering what success means. These are some context free questions:

1. What does success look like?
2. Why are these results desirable?
3. What is the solution worth to you?
4. What problems does this system solve?
5. What problems could this system create?

These questions all ask why we're doing the project, without actually asking why. Sometimes, when we use the word why, we sound as if we're interrogating others, or asking them to defend their positions. Interrogations or putting people on the defensive are rarely a good way to gather information about the project, and retain a good working relationship with other people.

You'll notice that the questions also don't ask "how". Sometimes, when we ask how, we start designing things. You don't want to design the project or the product, you want information about the product, to know how to evaluate the product's state of completion throughout the project.

Deb recently tried these questions on a project. She said, "Not everyone was happy that I wanted to pin down what success means. The VP Sales was unhappy, because it was clear that if we all knew what success was, he couldn't make the decision to release or not by himself, and he wouldn't be able to blame Engineering for his salespeople's inability to sell the product." Release criteria is not about the ability to move blame from one group to another, but to have some objective measurement about when the product is ready to release.

I use this objective measurement for accountability. Can the salespeople sell the product that meets these criteria? Can the support staff support the product? Can the trainers develop and deliver training? When I ask context free questions, other people realize they are accountable for their part, as well as pointing me towards project success.

Once you know what success means, you can define what's most important for this project.

Learn What's Important For This Project

Customers buy products because the products solve some problem the customer needs solved, not on the basis of the number of defects. When you define what's most important, don't stop at the number or type of defects.

If you're a non-profit or an in-house organization, this still applies to you. Non-profits still need revenue; they just don't distribute the excess profit back to shareholders. In-house organizations create products because your customers think you can create the product in question more cheaply than they can buy the product outside the organization. As an in-house service, you exist more to save money and time for the rest of the company, not to specifically create revenue. Whether you sell products or do work for in-house organizations, we're all solving problems for our customers.

What's critically important to this project is a combination of what the company needs and what the customers need.

Sometimes the date is most important. Sometimes, it's a particular feature or set of functionality. More often, it's a combination of schedule, features, and low-enough defects. It all depends on your customers, and their expectations.

Geoffrey Moore[1], discusses a model of high tech marketing, what your customers expect and when. If you combine the notion of what customers want and when, with possible goals for a software project [3], you have the table in Table 1.

Technology Enthusiasts	Visionaries	Pragmatists	Conservatives	Skeptics
1. Time to Market	1. Time to Market	1. Low Defects	1. Low Defects	1. Low Defects
2. Feature Set	2. Feature Set	2. Time to Market	2. Feature Set	2. Feature Set
3. Low Defects	3. Low Defects	3. Feature Set	3. Time to Market	3. Time to Market

Table 1: Project priorities based on where you are in the product's lifecycle

You'll need to define where your product is in its lifecycle, to decide what's most important. Even when you define what's most important for your customers, your company will still have corporate goals it needs to accomplish with a particular release.

Many project managers (PMs) talk about the trade offs between cost, schedule, and quality, but that's too simplistic for what PMs actually do. In reality, PMs trade off among the things the customers value: time to market, the feature set, and the defect levels, *and* the things the company values: cost to market, the number of people working on the project, and the working environment of the project.

As testers, we're more frequently aware of potential problems than other people. You can't change the corporate needs. After all, if you need revenue this quarter, you still need revenue. However, you can help illuminate the tradeoff decisions by discussing potential risks with the project manager. You can use the tradeoffs between the six project attributes (time to market, feature set, defect levels, cost to market, people, working environment) to best employ the test people and use time to the project's advantage.

A colleague, Rita, worked at a startup company that was driven by cash flow. They were not fully funded, so they had a tremendous push to ship product early, to get enough revenue to continue the company's existence. For their first, second, and third releases, their only release criterion was the date the release had to go to customers, so the company could legally recognize revenue. Once they made it past the first couple of years, and were adequately funded, they started developing other criteria, including defect counts, test progress, and the states of "frozen-ness" of the code. Rita described it this way:

"When we were a startup, we just needed to keep our heads above water. Our initial customers wanted pretty much what we gave them, but were willing to work with us. We were all blown away though, with last year's release. All of a sudden, the customers cared about defects, more than they ever had before. My management demanded to know what kind of a QA group I was running, and I felt completely besieged. The only thing that saved my sanity was knowing that I'd checked with the entire project team and senior management in advance, to know that the release criteria we chose were what we needed. Unfortunately, we didn't realize how much demand we would have for this product, and we now realize we can't use only the date or some defect or test numbers to assess the state of the release. We need a much bigger picture to know when we're ready to release the software."

Even when you discuss release criteria, you may not know everything perfectly. If your product transitions to the mainstream (see Figure 1) when you thought you were still in the Early Adopter phase, as Rita's product did, you'll be surprised.

For the next release, Rita and the rest of the project team came up with these release criteria:

1. All code must compile and build for all platforms.
2. Zero high priority bugs.
3. For all open bugs, documentation in release notes with workarounds.
4. All planned QA tests run, at least 90 percent pass.
5. Number of open defects decreasing for last three weeks.
6. Feature *x* unit tested by developers, system tested by QA, verified with customers A, B before release
7. Ready to release by June 1.

These criteria don't address everything for Rita's release, but they cover what's critically important for this release: the ship date, good-enough software, and a specific feature tested and working according to two specific customers. Rita was disappointed that only 90% of the planned QA tests had to pass—she thought there was too much risk with such a low passing number. But, after hearing what everyone else said, she was willing to go along with the rest of the criteria, because the release date was so critical for the release.

Draft Release Criteria

Rita first drafted a set of release criteria, to have a starting place to discuss the release with the PM and the rest of the project team. Originally, she had more criteria, and they were much more focused on low defects as a critical aspect of the release.

However, once she started talking to the PM, Rita realized that her initial criteria were not what the customers wanted. Yes, the customers were concerned with defects, but not to the same extent that Rita was concerned with defects. In fact, if a couple of the major customers were satisfied with the release, then chances were good that the release was good enough for the rest of the customers.

Initially, Rita's PM was surprised that Rita had tried to look at the whole release from the customers' perspective, and come up with a balanced idea of what would make a complete release. He'd expected that Rita would be much more concerned with a traditional test managers's perspective on quality—low defects. But, Rita knew from her previous projects at the company that low defects were just part of the story when making a release decision.

Make Release Criteria SMART

When you draft the release criteria, make sure they can be answered by anyone on the project team in the same way. I use an acronym, SMART, Specific, Measurable, Attainable, Relevant, Trackable to test that I have reasonable release criteria. Each criterion should be specific for this product at this point in it's lifecycle. When you make a criterion measurable, you're ensuring that you can evaluate the software against the criteria. Release criteria are not the place for stretch goals, so make each criterion attainable. Relevant criteria make sure that you're evaluating this product against what the

customer and management want. When you make criteria trackable, you can evaluate the state of the criteria during the project, not just during the last week.

“No high priority open defects” is an objective and measurable criterion. You can look at the open defects and verify that none of them deserve to be high priority.

“All open defects reviewed by cross-functional team” is another example of an objective and measurable criterion. Either the cross-functional team reviewed the open defects or they didn’t.

“Fast performance” is an ambiguous criterion. To change this into an objective and measurable criterion, make it something like this: “Performance scenario A (corresponding to use case A) to complete in under 10 seconds”. Here, you name the specific scenario, so people can refer to it, and give the performance criterion to verify that you met that performance.

Rita applied the SMART acronym to her release criteria, and was able to have the PM use the release criteria during the release and for evaluating the product's release readiness.

Gain Consensus on Release Criteria

Now that you have reasonable release criteria, it’s time to gain consensus on what you’ve developed. If people react negatively to your draft criteria, (“No, we couldn’t possibly do that”), then learn why they are concerned. Generating release criteria surfaces assumptions and fears about the project and the product. I've found these questions helpful when generating or gaining consensus on release criteria:

1. Must we meet this criterion before the release date?
2. What happens if we do not meet this criterion before the release date?

These questions generally help the entire project team stay focused on what's needed for *this* release.

As the test lead, I’ve gained consensus on release criteria in two ways: either draft the release criteria and discuss them and come to agreement at a project team meeting, or draft them with the PM and discuss them at a project team meeting. I prefer generating release criteria with the entire project team, because then the team owns the release criteria, not just the PM or me. However, if you’re working on a large project, or have never used release criteria before, and want people to understand what they’re working towards, you may be better off generating release criteria with the PM in advance.

No matter how you gain consensus, make sure you evaluate the release criteria, the product's readiness for release, not how the developers and tester achieved the release criteria.

Using Release Criteria

I use release criteria to evaluate the state of the project throughout the project. I ask my PMs to use the release criteria as part of the project team meeting, so that the entire team pays attention to the criteria, throughout the project. This way, we get to assess the project's state of "done-ness" during the entire project. When we get to the formal system

test part of the project, I use the criteria as part of the testing status report (the where we are in testing report).

I also find that release criteria are an early warning sign that you're not going to make the release. I was recently worked on a six-month project that had been underway for three months. The PM was concerned that the team was not going to make the ship date, so he asked me to help assess the state of the project. When I came in, there were no release criteria, so I first facilitated generating release criteria. The team was small and worked well together, so they didn't have any trouble generating release criteria.

The PM then used the release criteria each week in the project team meeting, to make sure the project was making progress. That worked for a couple of weeks. One week during the release criteria evaluation, one of the engineers said, "I'm not going to make it. I've tried and tried, and I'm just not going to make that criterion for our ship date." The PM said, "OK, I need to go back to management and see what we need to do. Before I do that, does anyone else think they're going to have problems meeting any of the release criteria?" Another engineer said, "I can't get performance that good between now and the time we have to release. When we discussed the release criteria, I thought it was possible, but now I realize it's not going to happen."

For me, release criteria are either met or not met. We aren't partway to meeting a criterion—we haven't met it. I find that this binary approach helps me more when I'm discussing the state of the software with senior management. If I say we're partway there, they hear that we're done. If I say we haven't met the criterion, they hear we're still working.

For this team, realizing they weren't going to meet the release criteria two thirds of the way through the project instead of at the end was a relief. The PM knew what the project reality was, and could work with management to see which tradeoffs made the most sense. In this case, the PM was able to renegotiate the release date, so that the product could meet the release criteria.

When You Don't Meet the Release Criteria

If all goes well, you'll evaluate the release criteria as you go, and you'll meet the criteria when the project is supposed to end. However, projects don't always go well, and you won't be able to meet the release criteria sometimes. When that happens, make sure you're honest about what's happening.

I generally don't change the release criteria. Sometimes, we learn more about what "done" means as the project goes on, and if it's necessary to change the criteria based on that new knowledge, then the entire project team agrees to the new criteria. Generally, I find that we're not able to meet some of the release criteria by the requested release date. Since I've already gone through the interviewing at the beginning of the release, and we've been measuring our progress towards the release criteria all along, this is not usually a big surprise.

If you can't meet the release criteria, say so. Have your management say something like this, "We thought these other criteria were important, but we realize now that the date is even more important than we thought. We're going to release the product, even though

we haven't met all the release criteria." If this is true, you can have them add, "We're going to determine what prevented us from meeting our criteria this time, and create the next project so that we don't miss our release criteria."

Summary

It's possible to create a software product your customers will want to use, and to know when you're done creating it. You don't have to play the "when can we release the product game". When you use release criteria to know when a project is done, you have taken potentially hidden decisions and made them public and clear. Make your release criteria objective and measurable, so everyone on the project knows what they're working towards, and then use the criteria as you progress through the project, through to final release. Then, you can say "release it" with pride.

Acknowledgements

I thank the following people who reviewed earlier drafts of this paper: Esther Derby, Dwayne Philips.

References

1. Moore, Geoffrey, Crossing the Chasm, Harperbusiness, New York, 1995.
2. Gause, Don and Gerald M. Weinberg, Exploring Requirements, Quality Before Design, Dorset House Publishing, New York, 1988.
3. Grady, Robert, Practical Software Metrics for Project Management and Process Improvement, Prentice Hall, New York, 1992.

Solid Software:
Is It Rocket Science?

Shari Lawrence Pfleeger
Systems/Software, Inc.
4519 Davenport St. NW
Washington, D.C.

ABSTRACT

The toughest kind of system to build involves safety-critical software, where the reliability requirements are extremely strict. We can never guarantee that our software will never fail, but we can take several serious steps to reduce the risk of failure: inspection, static analysis, hazard analysis, configuration management, and more. Can careful application of these techniques enable us to build software whose failure puts many lives at risk? We look at what "solid software" means in the context of the proposed National Missile Defense System.

BIOGRAPHY

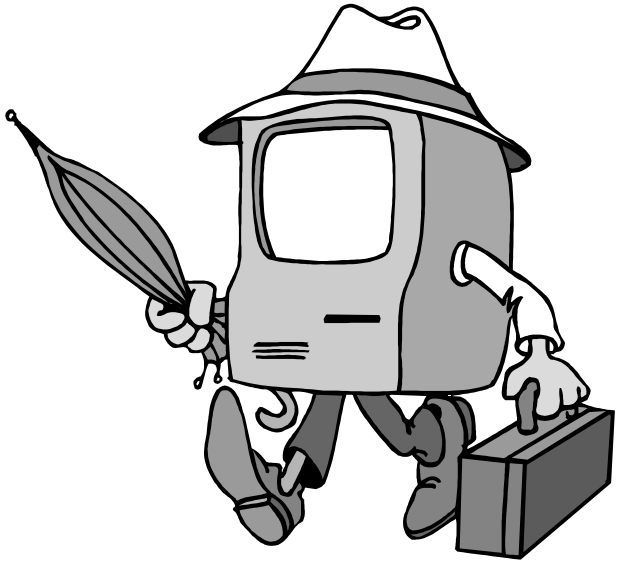
Shari Lawrence Pfleeger is president of Systems/Software, Inc., a consultancy specializing in software engineering and technology. Pfleeger is well known for her work in empirical evaluations of software technology, and her clients include major international corporations. Currently, she is investigating ways to transfer expert knowledge to new software engineers, and to improve our decision-making skills in software engineering. From 1976 to 1989, Pleeger worked on a variety of government and industrial development and maintenance projects, as a designer, programmer, requirements analyst and tester. Thus, her research and consulting are based in the practical problems of software development.

Dr. Pfleeger has been associate editor-in-chief of IEEE Software, where she edited the Quality Time column. She is currently associate editor of IEEE Transactions on Software Engineering. A popular speaker and instructor, she has been named repeatedly by the Journal of Systems and Software as one of the world's top software engineering researchers. Dr. Pfleeger is also the author of many books and articles, including "Software Metrics: A Rigorous and Practical Approach" (with Norman Fenton; PWS Publishing, 1997), "Applying Software Metrics" (with Paul Oman; IEEE Computer Society Press, 1997), "Software Engineering: Theory and Practice" (Prentice Hall, 2nd edition 2001), and "Solid Software" (with Les Hatton and Charles Howell, Prentice Hall, 2001).

Solid Software: Is It Rocket Science?

Shari Lawrence Pfleeger
Systems/Software, Inc.
4519 Davenport St. NW
Washington, DC 20016-4415
+1 202 244-3740
s.pfleeger@ieee.org
www.cs.umd.edu/~sharip

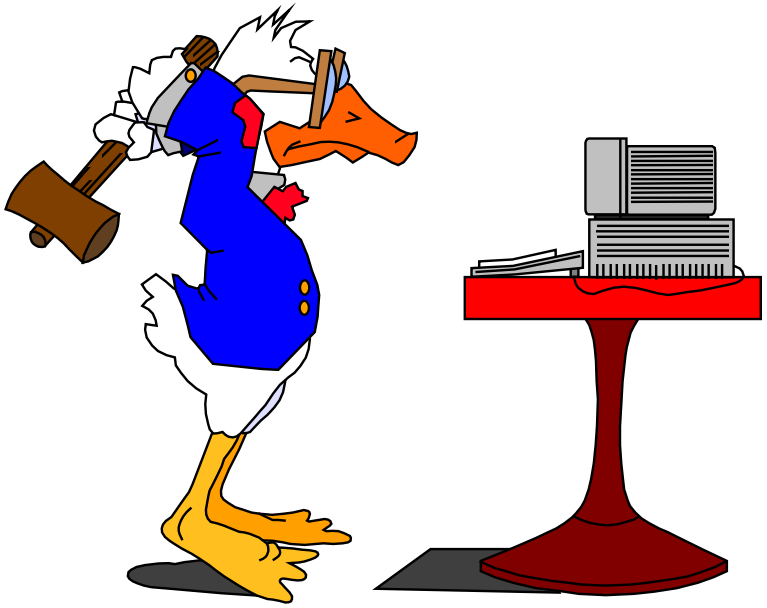
Overview



- What is solid software?
- Why is it so hard to build?
- The challenge of Strategic Missile Defense

What is solid software?

- Reliable
- Secure
- Predictable performance
- Easy to upgrade



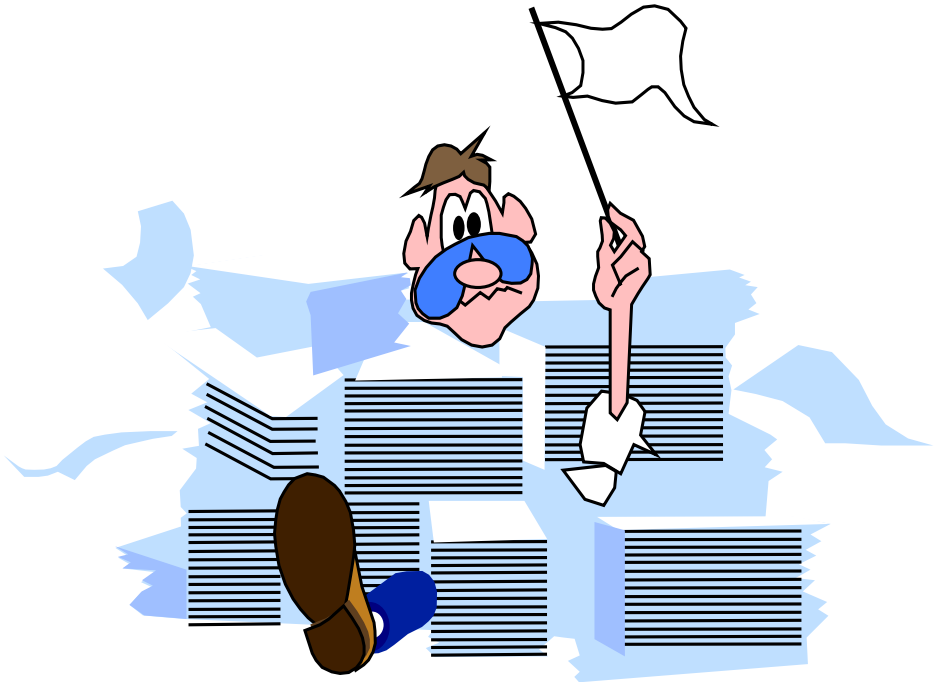
How are we doing now?

"The average customer of the computing industry has been served so poorly that he expects his system to crash all the time, and we witness a massive worldwide distribution of bug-ridden software for which we should be deeply ashamed."

Dijkstra 2001

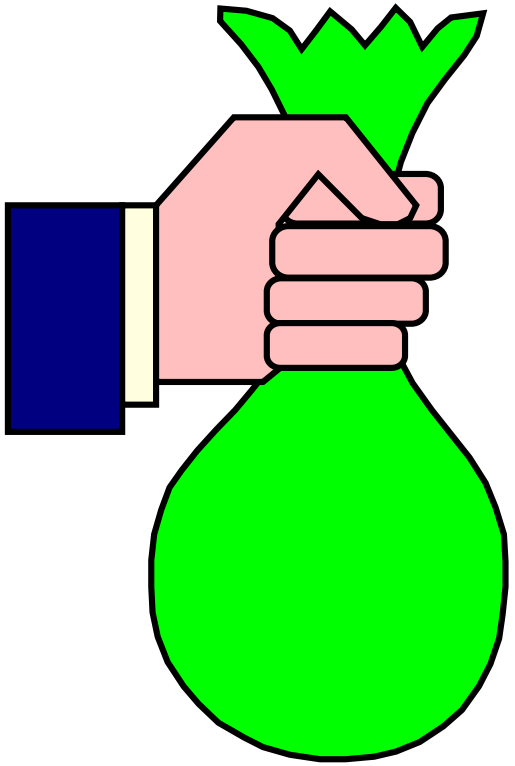
Example: A tragedy in four acts (Blair 2001)

- Act 1: Russia was tracking its nuclear weapons materials using a paper-based system: boxes of paper, paper receipts



Example: A tragedy in four acts (Blair 2001)

- Act 2: In a gesture of friendship, the Los Alamos National Lab donated to Russia the Microsoft software it uses to track its own nuclear weapons materials.



Example: A tragedy in four acts (Blair 2001)

- Act 3: Experts at the renowned Kurchatov Institute discovered that over time some files become invisible and inaccessible! In early 2000, they warned the US.

Example: A tragedy in four acts (Blair 2001)

- Act 4: The US told Russia to upgrade to the next version of the Microsoft software. But the upgrade had the same problem, plus a security flaw that would allow easy access to the database by hackers or unauthorized parties.

Computing's central challenge

According to Dijkstra (2001),

"Computing's central challenge, "How not to make a mess of it," has *not* been met.

On the contrary, most of our systems are much more complicated than can be considered healthy, and are too messy and chaotic to be used in comfort and confidence."

Is this based on evidence?

"Evidence over the past decade of *Inside Risks* and other sources suggests that we are not responding adequately to that challenge. Humans have repeatedly demonstrated our predilection for short-term optimization without regard for long-term costs."

Neumann and Parnas 2001

Why is it so hard to build solid software?

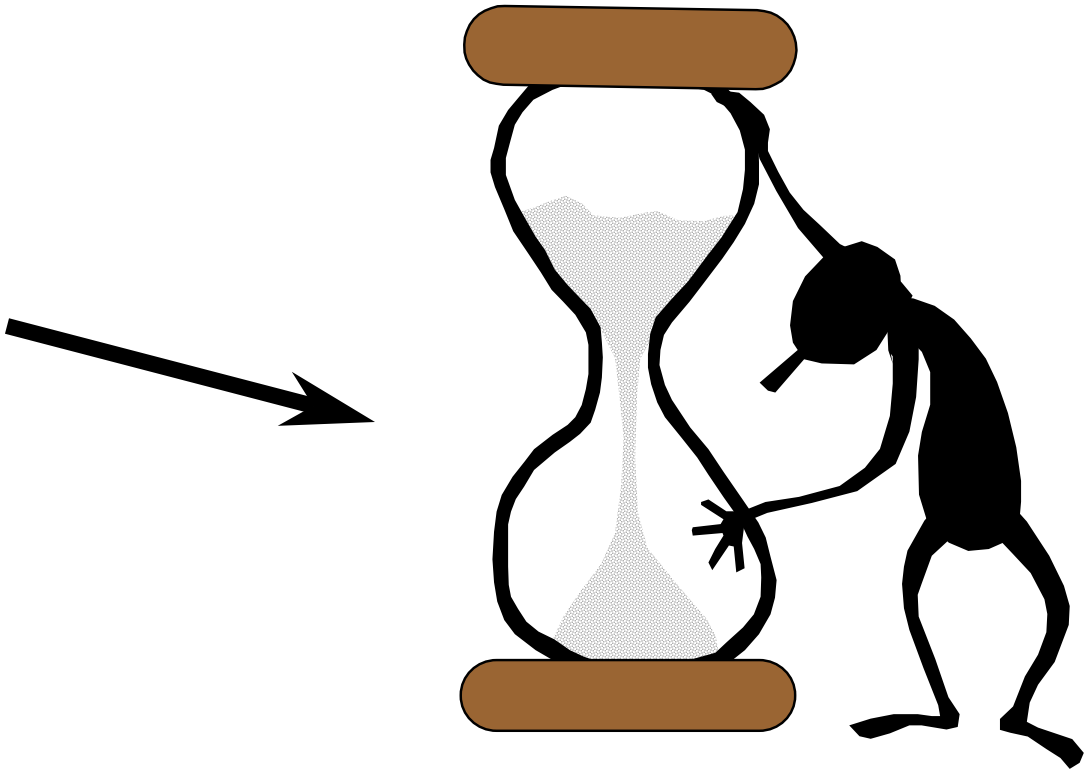
- Programmer optimism vs. gutless estimation
- Discrete vs. continuous systems
- Immaturity + rapid change = big problem!
- Repeating our misteakes misteakes

Programmer optimism vs. gutless estimation

Enthusiasm

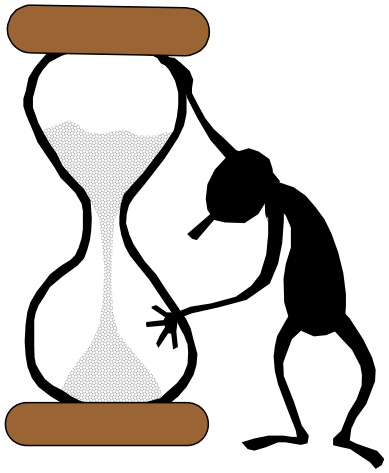
Programmer optimism vs. gutless estimation

Enthusiasm

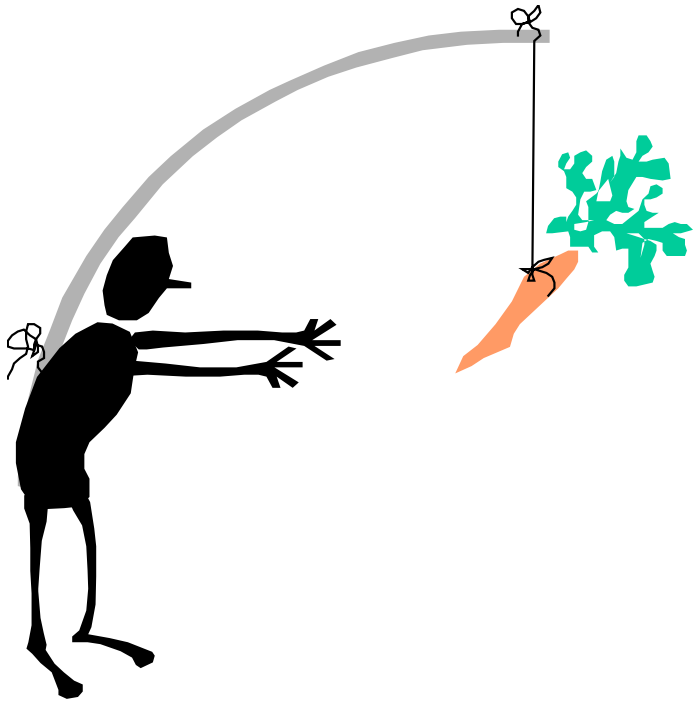


Gutless estimation

Programmer optimism vs. gutless estimation

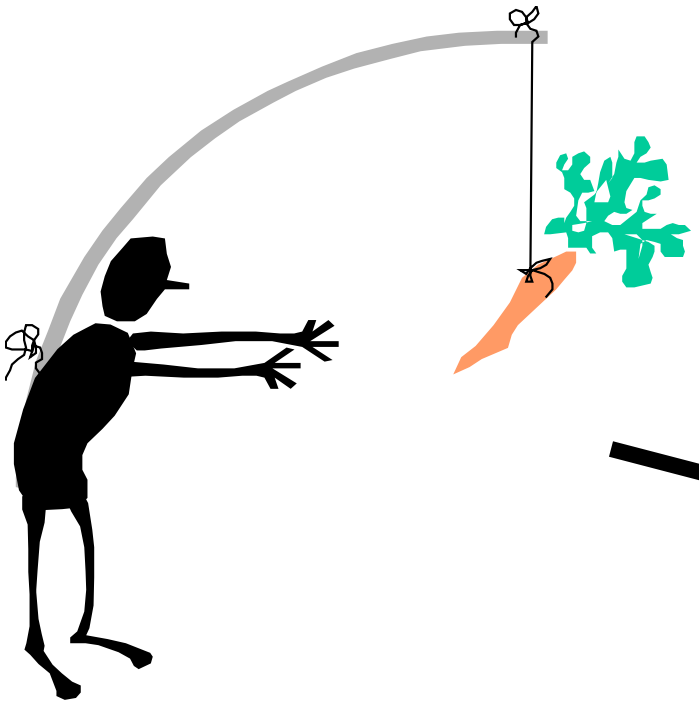


Gutless estimation

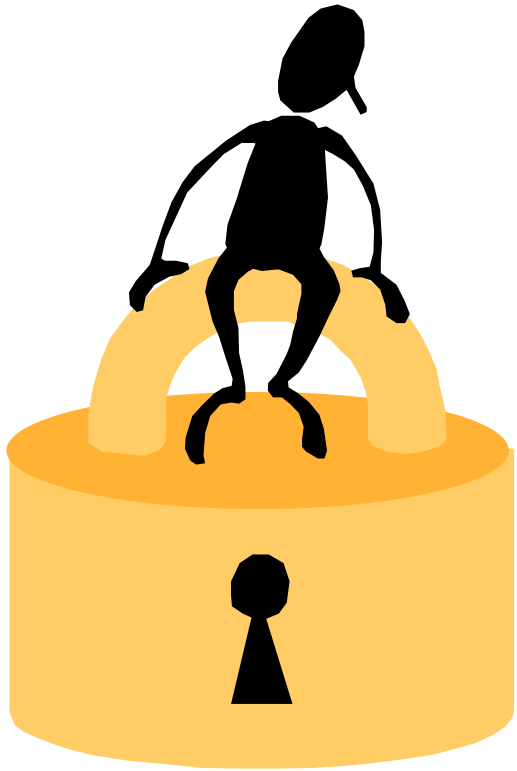


Target fixation

Programmer optimism vs. gutless estimation

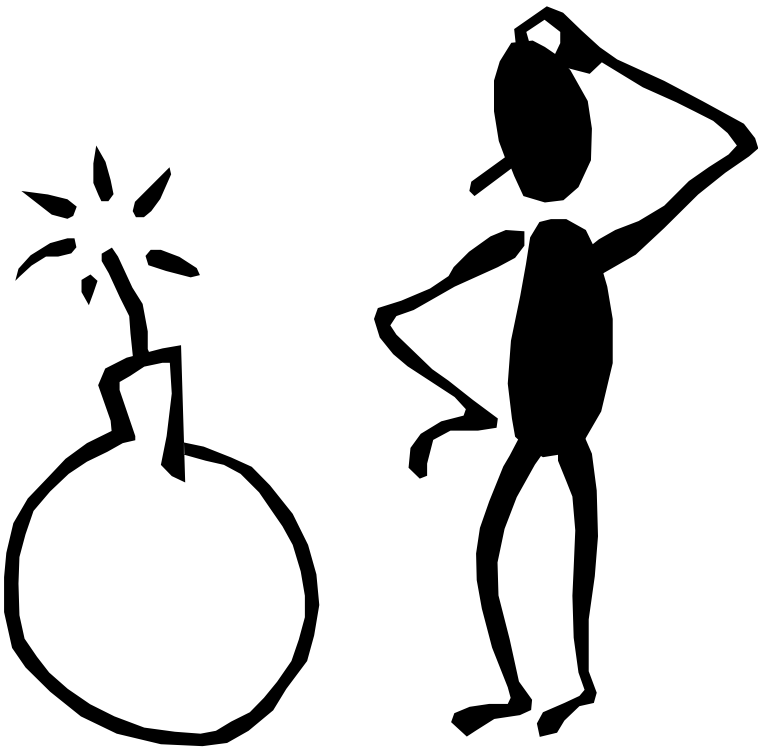


Target fixation



Unrealistic budgets and schedules

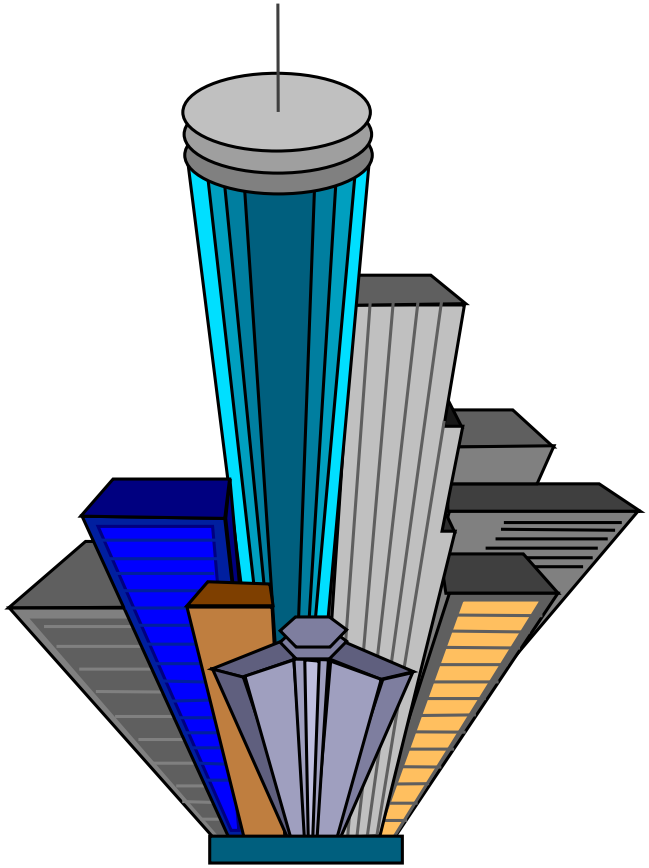
And finally ...



Unsustainable time pressure

Discrete vs. continuous systems

- **Sensitivity to small errors**
  - Example: off by one
- **Limited test interpolation**
  - Example: load testing, generalizing between two values
- **Absence of safety margins**
  - No way to overengineer



Immaturity plus rapid change

- Aggressive adoption and deployment of new technologies
  - Keeping up with the Joneses
- Limits of understanding complex systems
  - We may never have seen similar things before.

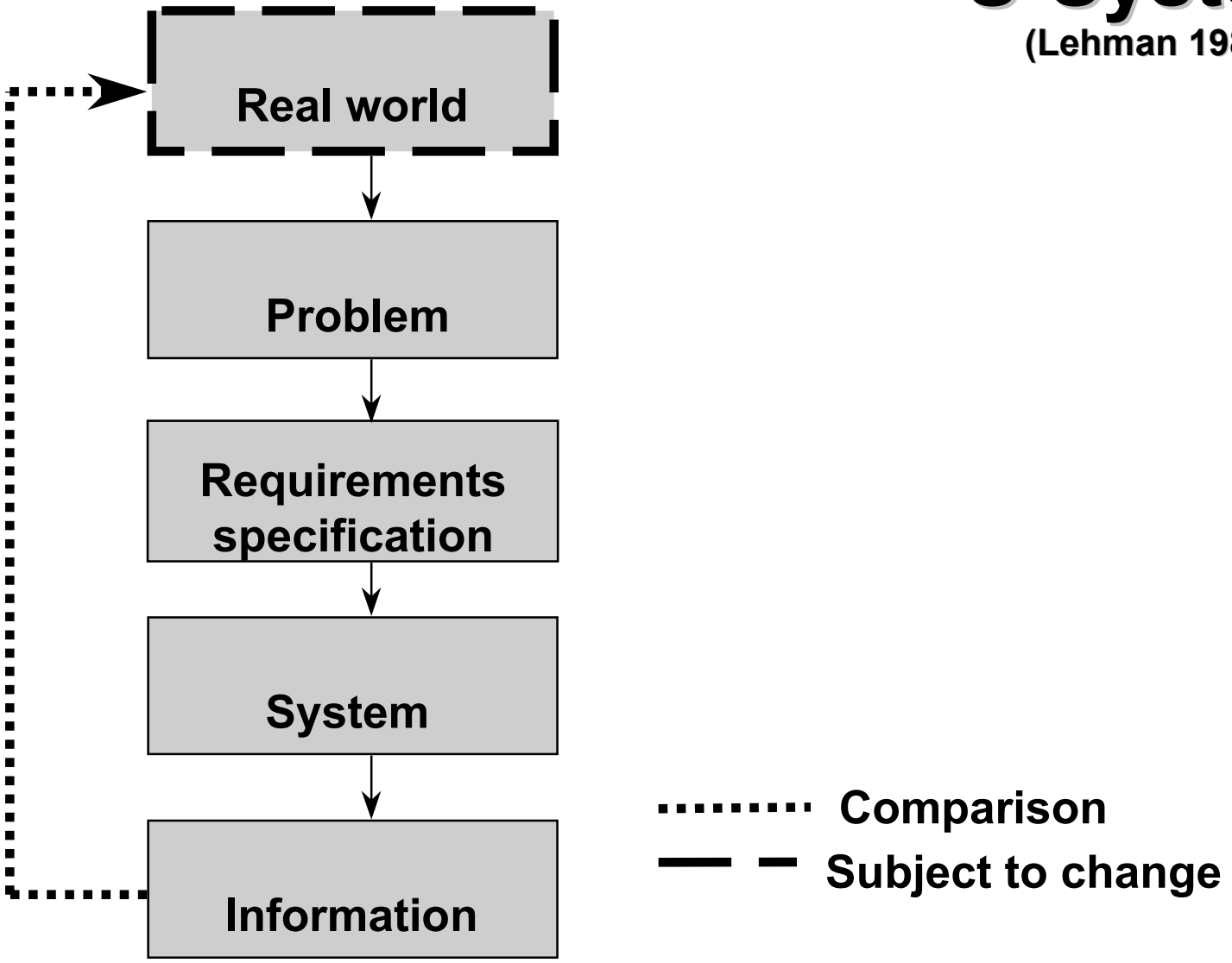
Simplicity vs. complexity

Dijkstra 2001: "While we all know that unmastered complexity is at the root of the misery, we do not know what degree of simplicity can be obtained, nor to what extent the intrinsic complexity of the whole design has to show up in the interfaces.

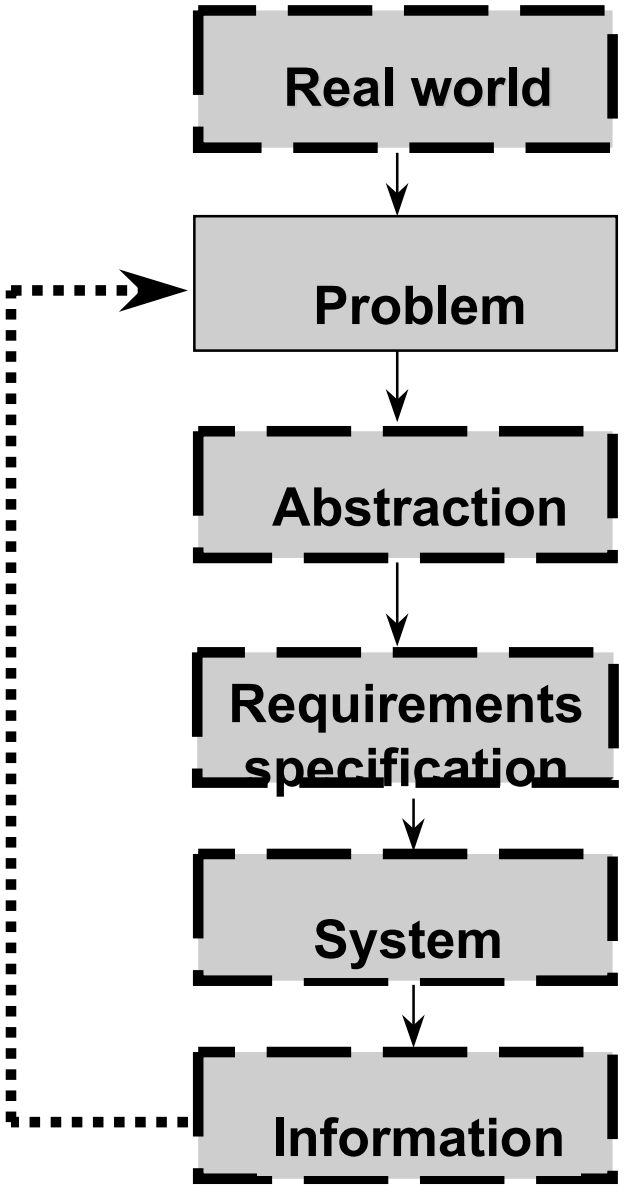
We simply do not know yet the limits of disentanglement. We do not know yet whether intrinsic intricacy can be distinguished from accidental intricacy. We do not know yet whether trade-offs will be possible."

S-System

(Lehman 1980)



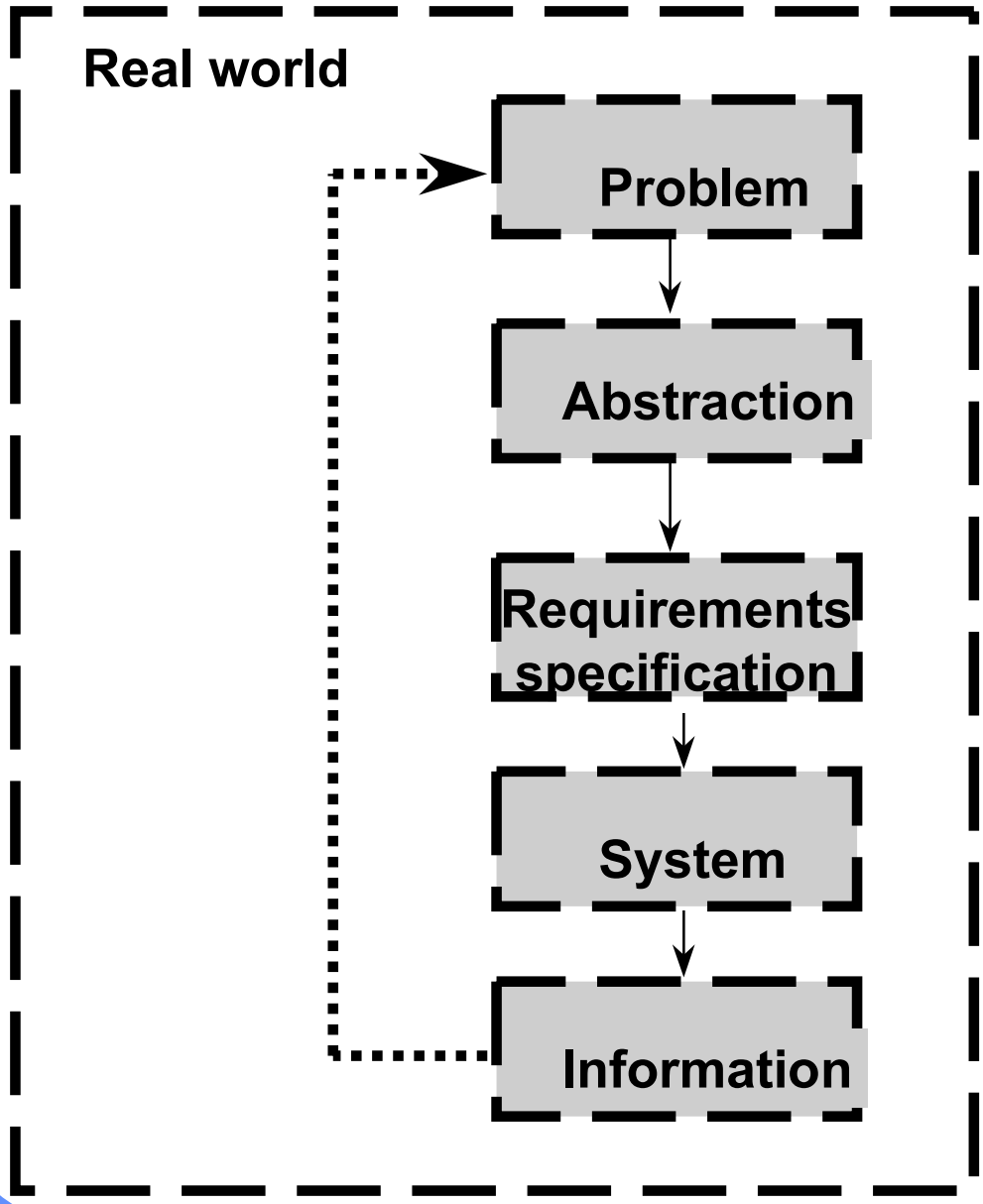
P-System



Comparison

Subject to change

E-System



Systems/Software, Inc.

Repeating our mistakes

"Most of us are so busy advancing and applying technology that we don't look either back or forward."

Neumann and Parnas 2001

Looking back and forward

"We should look back to recognize what we have learned about computer-related risks. We must look forward to anticipate the future effects of our efforts, including unanticipated combinations of apparently harmless phenomena."

Neumann and Parnas 2001

Typical responses to these problems

- Analysis paralysis
- Buzzword du jour

So what do we recommend?

- Understanding what you mean by quality
- Risk analysis
- Hazard analysis
- Careful testing
- Good design
- Effective prediction
- Peer reviews
- Static analysis
- Configuration management
- Using appropriate tools
- Trust but verify

The challenge of Strategic Missile Defense

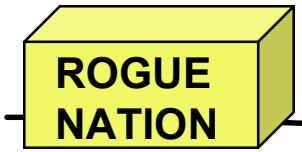
- SMD to be located in Alaska
- Attack missile: Estimated time from launch to impact is 25 minutes
- Interceptors destroy attack warheads by impact
- Interceptor must distinguish real attack warhead from decoys

Toles' view of the SMD

(cartoon in *Buffalo News*)

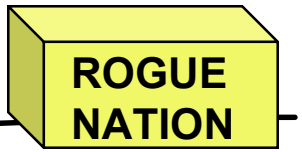
Toles' view (part 2)

It will have only one warhead and will be launched in daytime when it is easy to see.



Toles' view (part 3)

We will advise you of the exact time of launch, the exact size, heat and light characteristics of the missile, so watch out!



Toles' view (part 4)

Toles' view (part 5)

As long as the rogue nation is California ...

US DEFENSE PLANNERS

Richter's three key questions

- How reliable does the system have to be?
- How do we know how reliable it is?
- How does it handle decoys?

How reliable does the system have to be?

- Assume attack has five missiles.
- Suppose chance of one interceptor finding and destroying real warhead is 4 out of 5.
- Then chance of killing all five warheads is $(0.8)^5$, or about 1 in 3.
- That is, there is a 2 out of 3 chance that a warhead will come through.

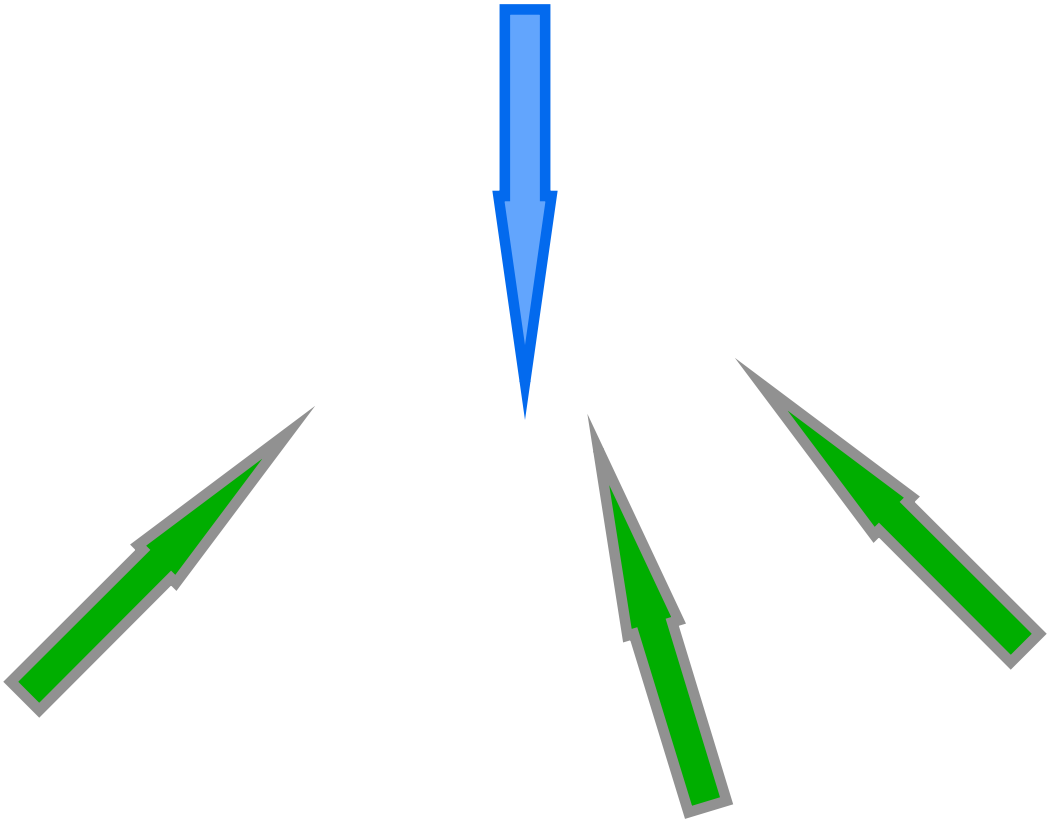
So what if we increase the reliability?

- If single interceptor reliability is 90% (that is, hitting 9 out of 10), the risk of one getting through is reduced to 1 chance in 3.
- To bring the risk of a warhead getting through down to 1 in 10, the single interceptor reliability has to be at least 98%.

No space launch system has ever achieved a reliability of 98%.

Can we improve the reliability somehow?

- We can fire more than one defensive missile for each attacking missile.



For example ...

- To reach a 1-in-10 risk, a single interceptor needs only 85% reliability if two interceptors are fired for each attacking missile ...
- ... and only 73% if three interceptors are fired, and so on ...
- but ONLY if each failure has a different cause.

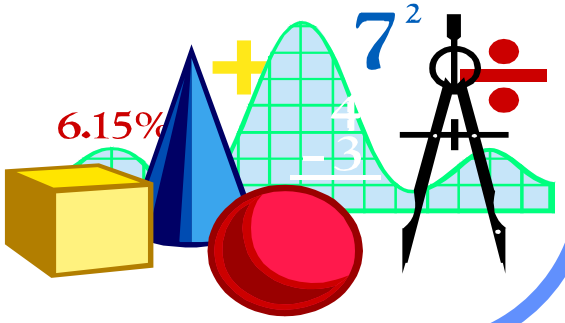
If all failures arise from the same problem, there is no increase in reliability when you fire more interceptors.

How can we tell how reliable it is?

- Testing and measurement
- The SMD is using a sequential set of tests: pass one hurdle and go on to the next.

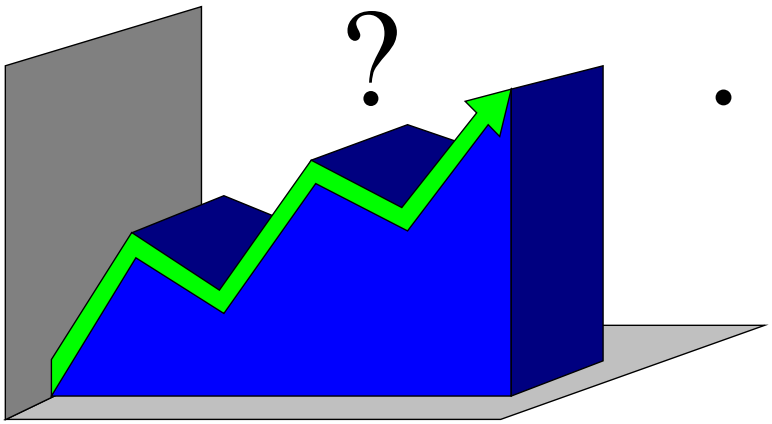
Passing one hurdle implies:

- 100% confidence that the system is more than 0% effective
- 0% confidence that the system is 100% effective



Systems/Software, Inc.

So how many tests do we need?



- With one test: we have 50% confidence that the system is 50% reliable.
- To reach 1 in 10 odds that a warhead will get through with two defenders for each attacking missile, we would need 18 tests (95% confidence that each interceptor is at least 85% reliable) with no failures of the full system.

How does the system handle decoys?

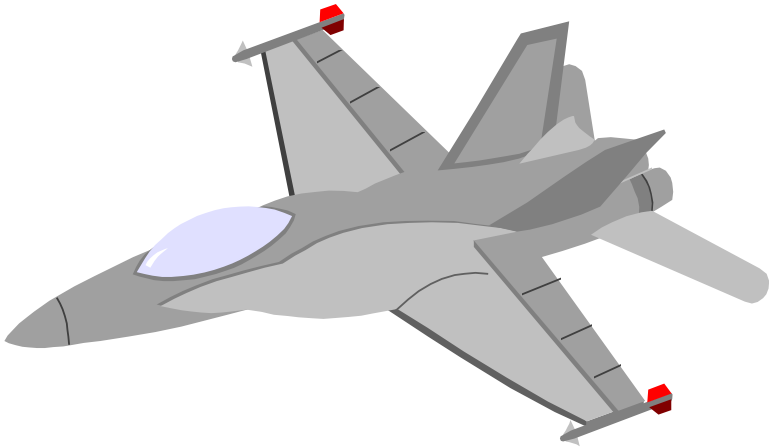
Three types of decoys:

- **Dumb: Look the same as warhead to the radar**
  - **Example: balloons**
- **Average intelligence: same shape, plus same temperature**
  - **Example: balloon with light bulb and battery**
- **Genius: more levels of similarity**
  - **Example: surround the warhead with a cooled balloon**

The proposed system considers only dumb decoys.

Richter's conclusions

"The proposed system is not ready to graduate from development to deployment, and probably never will be."



What about software reliability concerns?

- Safety-critical usually means 10^9 hours of failure-free operation.
- That means we would have to test the system for over 114,000 years!

What should we do?

"We must strive to make sure we maximize the benefits and minimize the harm. Among other things, we must build stronger and more robust computer systems while remaining acutely aware of the risks associated with their use."

Neumann and Parnas 2001

Recommendations revisited

- Understanding what you mean by quality
- Risk analysis
- Hazard analysis
- Careful testing
- Good design
- Effective prediction
- Peer reviews
- Static analysis
- Configuration management
- Using appropriate tools
- Trust but verify

We leave the last word to Shoe:

We leave the last word to Shoe:

We leave the last word to Shoe:

References

- Bruce G. Blair, "Nukes: A lesson from Russia," *Washington Post*, Wednesday, July 11, 2001, page A19.
- Edsger W. Dijkstra, "The end of computing science?" *Communications of the ACM*, 44(3), March 2001, page 92.
- M. M. Lehman, "Programs, life cycles and the laws of software evolution," *Proceedings of the IEEE*, 68(9), 1980, pages 1060-1076.
- Peter G. Neumann and David L. Parnas, "Computers: Boon or bane?" *Communications of the ACM*, 44(3), March 2001, page 168.
- Shari Lawrence Pfleeger, Les Hatton and Charles C. Howell, *Solid Software*, Prentice Hall, Upper Saddle River NJ, 2001.
- Don Phillips, "With one crash, Concorde ranks last in safety," *Washington Post*, Sunday, July 30, 2000, page A26.
- Burton Richter, "It doesn't take rocket science; to test missile defense, start with basic math," *Washington Post*, Sunday, July 23, 2000, page B2.

Cartoons

- Toles cartoon, *Buffalo News*, week after first successful SMD test, 2001
- Jeff MacNelly's Shoe, by Chris Cassatt and Gary Brookins, used with permission, from *Washington Post*, 26 April 2001

Code-Based Automated Software Testing In Visual Basic

Let Your VB Project Test Itself

By Jeff Muller, Sr. Systems Analyst
Bidtek, Portland, OR

1 - Abstract

Software firms have struggled with implementation of software test automation for decades. Several commercial products have been developed to address this need with mixed results. Many firms have given up entirely on automated testing, returning to alpha manual testing coupled with beta and post-beta testing by customers. After evaluation of these options, the only practical approach for a small software manufacturer was to develop a custom “code-based” test automation method that utilized existing project architecture, Visual Basic code, object structure, and data dictionary. This automation method creates a clone of the project under test, adding several forms, modules, and classes to handle the test automation. The cloning process scans each code object associated with the project, and adds special controls and events as necessary to allow project forms to initialize and close automatically. These special controls also provide a callback to a special Visual Basic class to handle the activity and logging loop through form input sequences in the data dictionary. This approach eliminated many of the drawbacks associated with scripted automation tools that examine application executables through operating system hooks. This test process is dynamic, since the cloned test code can be destroyed and refreshed at any time. The cloning operation does not substantially disturb the base code, and the development staff did not require training or modification of standard coding practices. Test automation has been a major benefit to Bidtek, allowing the skilled manual testers to concentrate on high-level operations requiring human judgment release to release.

2 - Introduction

2.1 - Company Profile and Product Under Test

Bidtek is a manufacturer of high-end accounting software for the construction industry. The company has been providing Unix-based software to this market since 1976. In the early 1990’s, we planned a conversion of our Unix product to the Windows operating system to meet market demands. After a period of evaluation to determine language and database tools, we established a group of developers to learn the new environment and to serve as the core for this conversion effort.

After about 4 years, we had a prototype in place with our core accounting modules. Soon after, we established primary beta sites with customers using duplicate systems for regular processing and testing of the new software. Success in that arena led to development of additional accounting modules and a full marketing effort by our sales group. At the time of this writing, Bidtek has completed all the product modules, and is in the phase of adjusting our product based on reported bugs and requested enhancements.

At the time of this writing, the company employs approximately 70 people with a QA staff of 3. Due to the complex functionality of Bidtek software, QA staff members are chosen from the support group and have accounting rather than programming backgrounds. The testing methodology is exclusively manual (other than the test automation method described in this

paper.) The company maintains an internal exception database to which anyone in the company can make entries. Bugs are assigned to the 13-member development group and are not cleared until reviewed by the QA group.

The Bidtek product is a full-featured accounting package primarily targeting the vertical market of Heavy and Highway road builders. It is programmed in Visual Basic with a Microsoft SQL Server database back-end. The VB code is organized into 17 product modules, and includes approximately 700 Visual Basic forms, 120 classes, and 80 base modules. The SQL Server database consists of approximately 700 tables, 700 indexes, 850 views, 12,000 registered data inputs, 100 user datatypes, 1900 stored procedures, and 1100 table-associated triggers.

2.2 - Our Main Problem

As a relatively small firm, Bidtek did not have the resources to establish a QA team with the technical expertise necessary to implement test automation. Our strategy was to promote individuals from support that had enough experience with the advanced accounting features of our product to serve as manual product testers.

As our product found acceptance in the marketplace we established a quarterly release process. In addition to the myriad of problems controlling the release of a huge code base and synchronization of databases with our customers, we discovered that the development group introduced low-level functionality bugs that were not found by our manual testers during the three-month release cycle. These new unreported bugs would, of course, be compiled into the product and shipped to our customers.

The QA staff simply did not have the time to test basic functionality such as opening all forms, movement between inputs, and launching of data lookups and setup forms; focusing instead on reported bugs and the more complex areas of our product. In addition, this complexity made it impossible for QA to verify with certainty that any bug fix did not disturb some other functionality that was not directly involved in the fix.

Consequently, we found our software arrived at our customers with functionality that worked before a release but that now produced errors. Needless to say, this produced high-level management attention. It was decided that some form of test automation was necessary to alleviate this problem.

3 - Initial Experiences

3.1 - Our First Attempt at Automated Testing

Since we had no experience in automated testing, our management decided to add a new member to the QA staff that would be responsible for this effort. Rather than promoting from the support group as had been our strategy, the new person was a recent college graduate with a general technical background, but who had no experience in our product and no direct experience in automated testing. After a few months of working unsuccessfully with Visual Test, this person was reassigned to support and the automation effort was suspended.

The failure of this initial effort actually benefited our next effort because management had learned several important lessons. Test automation isn't as easy as the automation tool vendors advertise. A firm can't purchase an off-the-shelf package and be operational in a few weeks. Test automation also requires commitment by management of top-level personnel with programming experience. The manager of a test automation must be given adequate time to implement a meaningful strategy that will be useful throughout the life of the product.

3.2 – Our Second Attempt

Management then came to me, a senior member of the development team. I was asked to study the situation and make a recommendation on whether we could implement any form of test automation with our product. I had worked on the development team for about four years and was experienced in our code and data architecture. This proved to be the key to the success we have achieved, as we ended up creating an automation test scheme that integrated intimately with our code and data.

My highest goal was to create an effective testing harness that could be handed back to QA and operated by a non-developer. The questions that needed to be answered immediately were "what is effective and what tool should we use". We of course first looked for a prepackaged application that could be used immediately, or a case study that we could be adapted to our needs. My research told me one thing: the software automation tool vendors say their tools work, but the experts that run the test consulting firms say that they are difficult and sometimes impossible to implement. I met with our top managers to give them my conclusions. We decided not to give up completely on a prepackaged tool without direct experience. Since Visual Test was the most popular automation tool available, I wanted to determine if we had improperly applied this product during our first effort at automation.

We decided that I should attend a class in Visual Test to make sure we knew how to use the software properly. The class confirmed our preliminary suspicions that the problems Bidtek was trying to solve with automation were well beyond the scope of Visual Test or any other script-based automation tool. We just did not have the manpower to create or maintain the test scripts utilized by this kind of tool. Another major problem was the difficulty Visual Test has in handling the custom controls used extensively in our product.

3.3 - Research Conclusions

There are several excellent articles on test automation in the literature (see References) that I was able to refer Bidtek's management. This literature supported the following conclusions:

- There was no test automation software that would work for us. Since our product exists in a constant state of flux, test suites would be obsolete shortly after creation. Our test department would have to be half the size of our company, staffed with experienced testers familiar with our complex software. The high relative manpower and maintenance requirements, and poor relative effectiveness, produced a high real cost that made this approach completely impractical for our small firm.

- Since no prepackaged existed, an experienced senior developer needed to write a custom application.

- Measurable results would take a minimum of 6-12 months, with continued commitment to produce a mature test harness that could be operated by a non-developer (QA). Continued maintenance of the application by the senior developer would be required.

- Results would be realized in phases, with testing of simple functionality progressing to more complex operations such as record inserts, updates and deletes; and finally to data validation after processing or batch operations.

- For our firm, test automation would augment manual testing rather replace it. Advanced tests that include highly complex interactive cognitive processes cannot be automated. These tests will always be left to skilled human testers.

Since the management team at Bidtek has a long-term view of product development and considers internal utilities valuable, I was given the green light to see what I could come up with.

4 – Code-Based Automation

4.1 - Basic Test Module Architecture

Existing case study literature on test automation was clear in pointing us to a "data-driven" or "framework-driven" solution using Visual Test. I concluded that neither of these approaches would work for us. Visual Test examines a form's objects (text boxes, option buttons, command buttons, etc) through the Windows API by reading properties of the objects according to the objects' window handles, values which must be consistent during the life of a form's instance on the screen. While we could certainly refresh our list of window handles for a form when a form loads, I found that the window handles for each cell in the editable grids we use were not consistent within a form's instance. Given that the handles can change each time a cell is entered, this whole approach became impractical since grids are a central part of our application suite.

I began searching for a solution that would allow us to test our product through a Visual Basic front-end. This would allow us to integrate a VB test harness with our existing VB/SQL Server code. Any member of the development group could maintain or amplify the system in the future. No one would need to be trained in Visual Test.

The following were the building blocks that provided the keys to the solution:

- Our application utilizes a set of standard class routines called by every form to handle form initialization, data display, menu/toolbar operations, and record operations.
- Bidtek had developed an extensive data dictionary that mapped all data inputs to forms, providing references used by the standard routines.
- Visual Basic utilizes simple text files to initialize projects and forms, providing an outside application easy access to the complete profile of an application.
- Visual Basic allows use of command buttons to trigger form events. We could thus install a hidden command button on a form and use it to remotely trigger the test operation.

I performed the following test to remotely control a VB form.

- Added a class called cBtkAutoTest to one of our projects to hold a test loop. In the initial phase, this was just a collection of a couple of operations on the form such as setting focus on a few of the controls.
- Added some code a form's Activate event to make a call to the test loop class. The Activate event occurs when the form becomes the active window. Thus, when the form is initialized in a call from the menu, the Activate event will fire and trigger the test loop.
- Added a simple controller form to trigger the test.

This worked on one form, but the next step was more challenging. How would we make the loop run an optional set of operations that could be logged for success or failure? My approach was as follows:

- Added a hidden label to the form that could be set to a numeric key indicating the operation to be performed.

- Added a hidden command button whose Click event could be fired by setting it to 'True'. The Click event would trigger the desired form operation based on the value in the hidden label.

For example, if the hidden command button was named cmdAutoTest and the hidden label `lblAutoTest`, we could set the label to a number between 0 and 14, then set `cmdAutoTest = True`, and the desired standard form routine code would be triggered. In this example, `mBtkForm` is an object variable referencing the class that serves as the portal to the Bidtek form standard routines.

```
Private Sub cmdAutoTest_Click()
    Dim rc As Integer
    Select Case lblAutoTest
        Case 0 'NEW_RECORD
            rc = mBtkForm.StdMnuRecClick(MNU_REC_NEW)
        Case 1 'SAVE_RECORD
            rc = mBtkForm.StdMnuRecClick(MNU_REC_UPDATE)
        Case 2 'UNDO_RECORD
            rc = mBtkForm.StdMnuRecClick(MNU_REC_CANCEL)
        Case 3 'DELETE_RECORD
            rc = mBtkForm.StdMnuRecClick(MNU_REC_DELETE)
        Case 4 'REFRESH_RECORD
            rc = mBtkForm.StdMnuRecClick(MNU_REC_REFRESH)
        Case 5 'MOVE_FIRST
            rc = mBtkForm.StdMnuRecClick(MNU_REC_FIRST)
        Case 6 'MOVE_PREVIOUS
            rc = mBtkForm.StdMnuRecClick(MNU_REC_PREVIOUS)
        Case 7 'MOVE_NEXT
            rc = mBtkForm.StdMnuRecClick(MNU_REC_NEXT)
        Case 8 'MOVE_LAST
            rc = mBtkForm.StdMnuRecClick(MNU_REC_LAST)
        Case 9 'INPUT_PROPERTIES (F3)
            rc = mBtkForm.StdMnuWindowClick(MNU_WINDOW_DEFAULT)
        Case 10 'INPUT_LOOKUP (F4)
            rc = mBtkForm.StdMnuWindowClick(MNU_WINDOW_LOOKUP)
        Case 11 'INPUT_SETUP (F5)
            rc = mBtkForm.StdMnuWindowClick(MNU_WINDOW_MAINTENANCE)
        Case 12 'CLOSE_FORM
            rc = mBtkForm.StdFormUnload(False)
        Case 13 'btkSendKeys_Tab
            btkSendKeys_Tab
        Case 14 'ShiftTab
            btkSendKeys_ShiftTab
    End Select
End Sub
```

In the following `Form_Activate` code, the `FormActivated` static variable prevents the test loop from being run repetitively if the form is called as a dialog form by another form. The `RunAutoTest` method is a member of the `cBtkAutoTest` class declared as `mBtkAutoTest` that holds the main test loop

```
Private Sub Form_Activate()
       Static FormActivated As Boolean
        If Not FormActivated Then
             FormActivated = True
             Wait 1, True
             mBtkAutoTest.RunAutoTest Me, mBtkForm, _
             frmAutoTestController, lblAutoTest
             Set mBtkAutoTest = Nothing
             Unload Me
        End If
End Sub
```

4.2 - Cloning the Project

This approach was promising for us, but there were several of major drawbacks in permanently changing the Bidtek foundation code.

- I could corrupt the code with new bugs.
- Since the rest of the group was unfamiliar with what I was doing,.other developers could innocently disable the test code.
- The test code would complexity not associated with the purpose of our product.

One solution to these problems would be to add the test code to a temporary copy of the code. However, adding the necessary code manually to each of the 700 forms would not be practical. I had to come up with a method to copy the application into a temporary clone that includes the new test loop class and controller form, and inserts the command button and Form_Activate code in each of the forms.

It might be possible to examine each of the text files that comprise a Visual Basic project, copying the files line by line and adding the necessary code. A VB project file is actually an initialization routine that tells the VB compiler what external code libraries (references), non-standard form controls (custom controls), classes, modules, forms, and project-level properties are included in the project, as well as their physical location within the machine. A VB form is also a text file that includes an initialization segment that tells the compiler the properties of the form, which specific controls are on the form and their location and properties, plus a code section that allows the programmer to control the behavior of the controls and the form itself. Classes and modules are also text files that include private descriptions of their properties to be read by the compiler, plus public areas for programming. All the important objects that we needed to clone are text files, and manipulation of text is a relatively simple and proven technology. A reliable cloning operation would not only avoid the problems associated with permanently modifying our base coude, it would provide some additional benefits:

- Maintenance problems associated with script-oriented automation would be eliminated. Tests would be self-updating since the cloned project could be deleted and refreshed at any time.
- We would not have to train the development group in how the test system worked so they would avoid disarming it, or ask them to remember to include code hooks or add special controls to new forms. Standard practices in the development group would be unchanged.
- Packaged properly, it could be run by the Bidtek non-technical QA personnel.

We had thus found our first main direction and the effort started to take shape. A new VB project was created to handle the project cloning, with a wizard-type form that easily leads a user through the creation of the ‘AutoTest’ project. Microsoft’s Scripting Runtime technology was used (sccrun.dll). This set of objects make it relatively easy to scan and copy a text file (the project file and its’ forms, classes, and modules), inserting additional code or form controls as needed:

- TextStream Properties
    - AtEndOfLine
    - AtEndOfSteam
    - Column
    - Line

- TextStream Methods
    - Close
    - Read
    - ReadAll
    - ReadLine
    - Skip
    - SkipLine
    - Write
    - WriteBlankLines
    - WriteLine

4.3 - Challenges in the Cloning Operation

Several challenges were encountered in developing the cloning process.

The first was learning how Visual Basic compiler interprets and records file locations. As stated previously, the project file is simply a list of objects contained in the project with their locations. These locations are recorded in a path syntax that is relative to the project file itself since a VB project can pull objects from any location known by the machine on which the compiler resides. The challenge was to interpret this syntax, copy it to another project file, and test the copied project file. Complicating matters was the fact that the copied project file could be in a different location on the machine or even on another machine. Thus a component file location that was understood as relative to one project file in its' location would not be understood by another project file in a different location.

Once I understood how this worked, the cloning process was developed with the following steps:

- Update the project to be tested from Visual Source Safe if applicable. This is an optional operation because we isolate our production and test code from the development code for a period of time based on where we are in our release period. This ensures that the database objects will be in sync with the VB code. During normal development, this period is one week and can be several weeks as we prepare for a compilation to be shipped to customers. The refresh operation from Visual Source Safe can only occur at the very beginning of one of these freeze periods.

- Copy the original project file and add the necessary lines to associate the test controller form, code modules and external references used in the automation loop.

- Copy each form, class, and module as reference lines are added to the clone project file, including any necessary test code hooks. Several custom situations arose that had to be handled in this step as well.

- Add a command to show the test controller form to Sub Main when encountered and so it will be displayed when the clone project is launched.

- Add a Visual Source Safe source code control file if not in existence or add the clone project to the existing file.

4.4 - Changes to Permanent Code Base

Several permanent changes to our code base could not be avoided, most especially to handle decision-type message boxes. All message boxes would stop the test waiting for a response from the console. Simple information-type message boxes (those that do not offer the user a choice

but only an OK button) would be easy to disable during the cloning operation by proceeding after skipping the message box and sending the message to the test log. Decision-type message boxes were more challenging. There was no was to know the best choice without examining the setting for each instance. Permanent code that made a best choice had to be manually inserted, allowing the program to proceed after logging the situation. Conditional compilation statements were used to ensure that the permanent changes would not be included in the versions that go to our customers. This was not a perfect solution, but one that would work until a more sophisticated test was necessary to actually test the choice options.

4.5 - Error Trapping and Logging

Error handlers were used in almost all of our thousands of subroutines. The standard error handler made a call to a standard routine class that read the error and determined how to proceed, so it was a simple task to include a conditional compilation statement within that class to branch off to a logging operation if we were running under test.

Since each of our forms references our standard routines through a family of classes, it was a simple matter to send a reference to all the properties of the classes to the error handler for logging. This reference produces an extensive source of information about the state of the form and its underlying code when an error is logged.

4.6 - A Second Wave of Challenges

Experience in running the cloned projects brought new challenges.

- The loop successfully called forms by walking our menu, but the forms did not close before the next form opened, causing overflow errors. Since the form under test made a callback to the class that called the form, the loop had to be adjusted with a wait routine and the processes carefully managed.

- Some forms were not testable under the standard loop, usually because they included custom situations. These forms were excluded from the test and logged as ‘not tested’ in the test log.

- Some forms needed to be temporarily excluded until the test project matured and was able to handle greater complexity. Our project architecture included three basic form types: maintenance, batch, and processing. Maintenance forms are designed to maintain data through direct binding to the database; batch forms hold data input from user input intended to be processed through a set of standard validation and processing routines; and processing forms perform non-batch processing tasks. Processing forms were indefinitely reserved for the future since record distribution testing required complex database setup and checking routines. Data input operations on maintenance and batch forms were immediately testable since errors could be trapped with our standard error trapping mechanisms. The initial focus therefore, was on simple navigation among inputs on standard forms with bound inputs. Eventually, different loops were developed for each of these form types due to other complexities.

- Since we focused on input navigation, it was necessary to establish a dedicated SQL Server database that included a record for each situation that could be experienced by the form. If the state of a control (such as selection in a check box, option button, or combo box) changed the visibility or enabling of other controls, each of these situations needed a permanent record in the database that could be called up by the loop in sequence to ensure that all controls were entered and all situations tested. The loop had to be adjusted to enter record keys for each form situation to be tested.

- Some forms were not listed on our menu and were only called as dialog forms from other forms. The test loop needed to know when the form under test opened a dialog form and performed the operation.

- Our bound data inputs allow users to directly open the maintenance form that maintains that data for that input. We call this a “setup” form, and one of the basic operations we were testing was this operation. Calling a setup form would fire the Form_Activate code, triggering a test on itself. Many forms were thus being tested repetitively, needlessly extending the test and inflating the log. This was especially acute when a data input occurred on numerous forms throughout a module. This situation was neutralized with a static variable in Form_Activate that allowed the test to be run on the form only once.

- To avoid errors generated during the attempt at logging, the logging operation needed to account for which properties of the standard routine classes were available at the point of error. A routine was written to handle this problem.

- Form operations called by the loop on the form under test needed to be flexible so that specialized tests could be developed in the future. This was accomplished by setting up the command button and label added to the form during the cloning process with code to trigger calls to the standard routine responsible for the form operation desired. The loop would set the label to a unique value that the command button would interpret as the case to select when it was set to ‘true’. The command button would thus run the exact same call to our standard routines as a user would when performing the operation live on the form.

- A special set of databases had to be added to our development environment to serve the testing tool. The first database holds maintenance data that allows the application to navigate each form situation that can be encountered. The second database contains data to verify success of record operations (inserts, updates, and deletes) and batch processing operations. These additional databases are updated on a weekly basis from our primary development database using a special procedure that assures the integrity of the data.

5 – Summary

5.1 - Current Use

We are continuing to develop the test application and learn as we go, with tests still being run by the developer of the tool. We are confident that the application will mature with the addition of more complex operations, and will eventually be turned over to the QA department for regular use. We currently use the automation tool as follows:

- The heaviest use of the tool comes at the end of our quarterly release cycle as the QA manager looks to assure the soundness of basic form operations. As the group enters an interim build period, the code and database are frozen and copied to a special cleanup area. After all critical issues are cleared, a final compilation is made and shipped to beta release customers. Automated tests are run on the frozen code and database until all testable forms are without error.

- Before our code is frozen for a shipment, the automation tool is run against our standard development code. During this period, we run a weekly compilation with an update of the database used by QA to make sure that their database matches the code base against which they are running. We apply the same concept to the automated test environment

by running a weekly database update to its database, and copying the week's code to a private directory to be used as the test target code.

- During both periods, a complete set of reports is generated covering every form in the system. Manually repeatable failures are logged to our in-house bug reporting system and assigned to the developer responsible for the form. Test log reports are distributed to management and module lead programmers.

- Special tests are under development, as requested by development group members and management, to cover critical processing operations that are particularly troublesome and sensitive to recoding. A good example is payroll processing, which is particularly sensitive to our customers, requiring re-verification after any significant modification.

5.2 - Conclusions

Bidtek continues to develop our custom test harness, and is committed to making it an essential part of its QA effort for the future. As with any software development effort, we are learning as we go. A key to our effort is that management recognizes that the potential benefits are as great as the investment required. The following are a few conclusions from our experience:

- **Software test automation is a difficult enterprise** that requires a long-term commitment of time and highly qualified personnel. Management must accept realistic expectations for measurable results.

- **The development effort must be carefully planned with realistic management expectations.** Bidtek management agreed from the very beginning that testing should begin simply and progress to more complex operations as we learned.

- **Script-based test automation is impractical for large, complex systems.** Operating the test harness through our VB code eliminated many of the serious drawbacks inherent in script-based automation software. We view our test application as operating "from the inside" through our code, rather than looking "from the outside" by examining screen representations of an executable. We avoided the complex and special language these script-based packages utilize to make calls to the operating system to read and manipulate the executable's representation on the screen.

- **Testing through our Visual Basic code makes the test harness relatively easy to understand by anyone in the group**. Avoiding the custom languages inherent in script-based packages allows another developer to pick up the test harness development effort if necessary.

- **Utilizing a "clone, test, and destroy" approach insulated the test from the base code and the development group.** We kept the addition of permanent test hooks to the base code to a minimum and eliminated onerous script maintenance. Also, implementation of our automation effort did not require us to train development group personnel or modify practices.

- **Our method requires a detailed data dictionary and a mature set of standard routines.** This foundation allowed the test harness to manipulate data inputs and trigger form operations with simple class method calls and data dictionary references. Adapting this method to an application without this foundation would require modification of the cloning routine to log all controls on each form with a modification of the test loop. Some functionality may be limited.

6 - References

A number of excellent articles exist in the literature containing case studies on software test automation. The following are the principal references that guided our thinking as we came to find our own method for automating our software tests.

Test Automation Snake Oil - James Bach
http://www.satisfice.com/articles/test_automation_snake_oil.pdf

Classic Testing Mistakes - Brian Marick
http://www.testing.com/writings.html

Improving the Maintainability of Automated Test Suites - Cem Kaner
http://www.kaner.com/writing.htm

Totally Data-Driven Automated Testing - Keith Zambelich
http://www.sqa-test.com/w_paper1.html

Seven Steps to Test Automation Success - Bret Pettichord
http://www.io.com/~wazmo/papers/

Automated Software Testing - A Perspective - Kerry Zallar
http://www.testingstuff.com/autotest.html

Software Testability:
Lessons from Hardware Testing

Alan A. Jorgensen, Ph.D.
Senior Research Scientist
Center for Software Engineering Research
Florida Institute of Technology.

ABSTRACT

Years ago computer system hardware faced quality and reliability problems similar to those faced today by computer system software. The hardware problems have seen dramatic resolution in the past few decades. A major reason for this improvement was the integration of testing and test automation into the hardware development process. When hardware development responsibility included testing, design features were added to support and reduce the cost of testing. Perhaps parallels can be drawn with software. This paper looks at some hardware testability features and attempts to define parallels in software. Hardware testability features include parts lists, full reset, test points, controls, 2nd sourcing, self checking, test jigs, and vendor qualification. Each of these has an analogy and application to software testability engineering. Simply identified the software parallels are respectively, component lists, full variable initialization, test access to data (fetch and store), compatibility of low level functionality, assertions, limitation of allowed calling sequences and qualification of outsourced software components. This paper details those analogies with examples.

This paper was inspired by and should be attributed, in part, to the 2nd Austin Workshop on Test Automation (Software Testability)[1].

Biographical Information

Alan Jorgensen has been developing test automation for computer systems since 1963. He developed technology for hardware and system diagnostics for real-time process control systems. He now performs research in software testing and teaches software engineering, particularly software testing at the Florida Institute of Technology, Melbourne, Florida.

[1] The 2nd Austin Workshop on Test Automation (AWTA 2), January 2001, Austin, Texas.
AWTA 2 focused on software design for testability. Participants were Alan Jorgensen, Allen Johnson, Al Lowenstein, Barton Layne, Bret Pettichord, Brian Tervo, Harvey Deutsch, Jamie Mitchell, Cem Kaner, Keith Zambelich, Linda Hayes, Noel Nyman, Ross Collard, Sam Guckenheimer and Stan Taylor.

Software Testability: Lessons from Hardware Testing

Rationale

This paper is about improving software quality. Software quality has been elusive, even evading clear definition. I am going to work to a broad and abstract view of quality that encompasses such ideas as minimizing maintenance costs (or optimizing service contract profit) and customer satisfaction, or (hope against hope), even customer "genuine delight." [Denning 1992]. For well-researched definitions of quality and software quality, see Eriksson, McFadden, and Tiittanen's paper, [Eriksson 1996].

In the early years of computing, similar problems existed in computer systems hardware. For instance, the ALWAC IIIE, a computer system from the late 1950's, possessed a Mean Time To Failure (MTTF) of about 8 hours. A typical application was the inversion of a 32x32 matrix requiring something around 8 hours to complete. After a week of overnight runs, there was some hope that a single computation could be completed. To really put this in perspective, the ALWAC IIIE did not have as much computing capability as the average digital wristwatch today.

Software, conversely, was much less complicated and consequently much less prone to design error. A single application typically ran on a single machine and was never subject to a changing environment. A single function was considered quite large if it occupied 256 words of memory. Current technology produces software of massive scale that is subject to frequent changes of environment.

Today, the cost of software failures far outweighs the cost of hardware failures. In our newspapers it is unusual to read of significant computer hardware failures even annually where we read of significant software failures nearly every day.

In the sixties we fought to improve those hardware quality problems hearing much the same arguments we hear today.

> "You are not supposed to do that."
>
> "That is the way it is supposed to work."
>
> "It works fine on my system."

We fought our way out of this in two ways: customers demanded improved performance and quality, a driving force; and improved hardware testing. Improving testing wasn't always easy. What was supposed to be tested? Why doesn't it work the same way when I do the same thing twice in a row? How can I see what is going on inside? Why can't I skip over that two-hour time-out? Why do I have to test the same thing over again? Was it really tested before? If Company A supplies some bad parts; can I buy better ones from Company B?

We hammered out solutions to these questions and called what we were doing, "Improving the Testability" of the hardware. It wasn't always easy, because there is a cost to everything. But there is also a cost for bad quality.

But during this same period, the capacity of the hardware systems limited the complexity of software systems. Software complexity was much less that hardware complexity; software took less time to develop and test while hardware was increasingly complex and unreliable. This trend reversed.

In their year 2000 system quality survey, PC Magazine found that of 6282 office desktop PCs reported, 1981 required repair within a 12 month period [Bsales 2000]. Assuming a single failure per unit needed repair that means that office desktop hardware exhibits at least 27772 Hours MTTF. Compare that to 8 hours MTTF some 40 years ago (even assuming equal functionality!). We've come a long way, hardware wise, and improvements in hardware test software and hardware testability contributed significantly to this improvement in hardware quality.

This paper suggests possible parallels between hardware and software testability features. This parallelism might be exploited to provide improved testability of software.

Some of the software testability features mentioned here may already exist in some form.

This paper does not attempt to determine if the features suggested here actually improve the testability of software; verification is left for later work. It avoids being too specific about the mapping from hardware to software testability to avoid preconditioning the reader. The purpose here is to stimulate thought and discussion on software testability using hardware testability as a model.

What is Testability?

Here, again, a somewhat abstract definition may well serve: Testability features are those features of a product that reduce the cost of testing. Note that this may include features that are part of the product for other reasons but added in such a way that testing costs are reduced. Such costs must be weighed in terms of the benefit in cost of maintenance (including the "cost" of customer dissatisfaction).

Pettichord [Pettichord 2001] defines testability in terms of visibility and control with refinement including other "-ilities" such as operability, observability, simplicity, etc.

Other definitions include [IEEE 1990]:

"*…the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met*."

ISO [ISO 1991] provides a slightly different slant:

"*Attributes of software that bear on the effort needed for validating the modified software*."

Mapping Hardware Testability onto Software Testability

This paper suggests a technique for identifying potential testability features by examining hardware features that contribute to testability and see if a way can be found to map that feature into characteristics of software design.

Many testability features serve multiple purposes such as enhancing manufacturability. Many testability features in hardware have been implemented with automation.

The properties of hardware design for testability to be examined are:

- Parts Lists
- Test Jigs
- Full Reset
- Test Points
- Test Controls
- 2nd Sourcing and Vendor Qualification
- Self Checking

Parts Lists

Hardware parts lists itemize the components and/or subassemblies of an assembly. For complex assemblies they are hierarchical in that an assembly parts list may include reference to other assembly parts lists. What does this have to do with testability? A lot. First of all, it is a list of all the things that must have been tested for there to be any hope that the assembly will work correctly. It forms a checklist of sorts that nicely defines the scope of work for testing. The assembly and all of its components must be tested. Further, given the quality characteristics of each item on the list and knowledge of the quality of construction, quality characteristics of the assembly can be estimated.

Now with hardware, it is not likely that an assembly can be constructed without parts lists. Hardware parts lists were not devised for the purpose of easing testability, but they surely do help.

Software could have parts lists. Indeed, it is not entirely uncommon to have "cross reference" lists. The suggestion here is that formal, maintained listings of software component dependencies would aid considerably in the planning and implementation of software test activities. Accurate lists identifying component coupling would be very useful for identifying areas requiring retest after modification and correction of that component.

By the way, hardware parts lists are also used to locate defective parts when a design or testing shortfall has been discovered. If you discover there is an error in *fclose* in *libc* requiring a calling sequence change, will you need to search your entire code base?

Test Jigs

Test jigs on the other hand, are designed for testing. They are not part of the shipped product. But they place a requirement on the hardware to conform to certain standards of consistent form and fit. Over the years there have been many standards, each one designed to fit a particular mounting scheme and electrical connection. Fourteen pin DIP (Dual In-line Package) integrated circuits come to mind. This standard is still in use. Even axial lead components have this standard form and fit property. True, consistent form and fit greatly aids construction as well as allowing uniformity in test jig configurations. (Perhaps there is a suggestion here for ease of software design as well as testability).

In software, this can map into a uniformity and limitation of software component interface formats, e.g., a limited number of calling sequence structures. This author has had considerable success designing test automation software with no more than four types of component interface specifications:

Function(void) returns Integer;

Function(Integer) returns Integer;

Function(Integer, Integer) returns Integer;

Function(String) returns Integer;

Though this led to many more functions in the system, component cohesion was extremely high, most components were generated automatically, as were the test drivers and stubs (thus easing the effort required to test the test automation system), and system diagnosis was much easier. Small functions with simple formal parameter lists are much easier to test avoiding the "Input Explosion" problem identified in [Jorgensen 2000]. (This technique translates "Input Explosion" to "State Explosion." In the author's opinion it is easier to handle only this latter problem, rather than to handle both.)

Designing unit test drivers for such a collection of functions is substantially easier than it is for a wide variety of complex function call formats.

Some work exists and commercial tools are becoming available to automate the creation of "test drivers."

Full Reset

During the transition phase from ad hoc hardware testability design to carefully designed testability features, this capability proved difficult to define and implement. In hardware logic design, we generally accept the characteristic of storage elements (flip-flops) allowing them to be preset to a known condition. This is required, of course, to meet the definition of a test:

- Start from a known state
- Apply a known stimulus
- Measure Response
- Compare measured response to a predicted result

Having a known starting state was not that easy to obtain because of sneak feedback paths in the logic. This problem was solved for hardware when logic analysis tools became available that identified these hidden "storage" elements so that they could be set to a specific condition on initialization.

Perhaps software is more straightforward. Simply set all variables to a known condition. It is a good practice to dynamically initialize all variables. Sound easy? Not likely. This is not easy even if we restart our application. Full application restart is not an efficient technique for repetitive test sequences.

Test Points

In hardware we found that to determine the correctness of intermediate results we needed to add test points. These points were simply connections to points "inside the system" that were normally not accessible by means of the terminals available to normal "users" of the component. Our design rules forbade using test points for any purpose other than testing. Later, when hardware became more complex and automated testing was necessary to support the volume of tests necessary to have a reasonable assurance that the hardware functioned properly, programmatic test points were added: internal status values that could be read by the "diagnostics." Enforcing these rules was much more difficult and sometimes we found it necessary to add status lines that made no sense in designs utilizing new technology because a software developer found some use for a programmatic test point.

In software, a test point might be as simple as providing access to a private variable only for test purposes. Modern compilers and linkers, however, do not support this concept. We can use conditional compilation, of course, but experience has shown that the best practice is "test what you ship."

Test Controls

In hardware, in order to provide efficient testing, it was sometimes necessary to add the ability to set hardware in an unusual state or provide some non-standard type of input. Providing the ability to force the early completion of a long timeout delay would be an example. In one instance a test control function saved an expensive design change. We had designed a synchronous communications controller but had to anticipate the HDLC standard. In order to provide a more testable CRC generator we provided the ability to set the CRC to any initial value, though on reset it was set to zero. This "testing" function became a useful design feature when the HDLC CRC was specified starting with all bits set.

The possible parallel in software may be similar to the software test point feature, allowing variables to be set so that the software to be set to special, otherwise hard to reach states. I have found this to be very useful for testing the boundary conditions of very large stacks, for instance. I could avoid long tests requiring 10^7 items being added to a stack to ensure that stack overflow logic operates correctly. Again, it would be desirable if such a function were only available during testing activities, since manipulating stack properties in an application would be fraught with danger.

Vranken, Witteman, and Wuijtswinkel [Vranken 1996] suggest a technique that embodies test points and test controls at the inter-component communication level (Point of control and observation, PCO), however, there may well be a need to observe or control the inner workings of a software component to provide the required testability features.

2nd Sourcing and Vendor Qualification

Hardware and software construction takes place by the assembly of many different components from many different vendors. In hardware, there is usually choice over a wide range of vendors at a very low component level. For hardware, we found it unnecessary to retest components that had previously been thoroughly tested. At first we found it necessary to test every component that came in the door, but eventually we found it wiser to ask our vendors to supply us with test data for the components being supplied. In many cases it was easier to ask the vendor to test additional requirements than it was for us, since the vendor had the jigs and test equipment already in place.

In software we have not achieved this capability. We should be able to ask a software component vendor to provide test data. What tests were performed? What was the success rate? What error rate can be expected from the shipped product? Who will bell the cat and refuse to purchase software without such quality certification information? Whitehouse.gov, are you listening?

One experience of this author took place manufacturing computer systems where the parent company also had a solid-state device manufacturing capability. Since it was all "in-house" we did not (or could not) demand the same vendor qualification practices and we accepted a shipment of marginally bad diodes (they *usually* worked). I spent a lot of hours on the road in strange facilities finding those diodes.

Self Checking

Self-checking circuits in computers are nearly as old as computers themselves. We still use parity check in memory. Bi-quinary (2 out of 7) notation in IBM 650 *circa* 1960 allowed the design of a circuit to check to ensure that the machine added correctly.

Similar functionality in software is dynamic assertions. When are assertions necessary and when unnecessary? What should be checked and what should it be checked against? What should happen when an assertion proves false? Verified best practice standards do not yet answer these questions.

Some languages support automatic subscript range checking, a "built in" form of dynamic assertions. Are compilers smart enough to know when this is unnecessary? How about formal parameter range checking? When is it necessary? What is the impact on performance? Again, our current compiler/linking tools do not adequately support this testability feature.

The "Garbage In, Garbage Out" paradigm has been an excuse for poorly performing software for decades now. This author subscribes to the "Garbage In, Apology Out" paradigm.

Conclusions

There are not many conclusions to be drawn from this paper. This paper has presented some suggested analogies between hardware testability features and potential software testability features. Table 1 summarizes this mapping.

Table 1 – Testability Feature Mapping Summary

Hardware Testability Feature	Suggested Software Testability Feature
Parts Lists	Dependency Tables
Test Jigs	Limited Calling Sequence Structures
Full Reset	Static and Dynamic Total Initialization
Test Points	Test Access to Variables
Test Control Points	Test Modification Access
2nd Sourcing and Vendor Qualification	True Vendor Choice, Public Test Data
Self Checking	Dynamic Assertions

Software testability has not been a high priority design requirement; our tools have few capabilities designed to support the implementation of testability. Perhaps we can learn from the examples of other engineering technologies. Perhaps, someday, software quality engineering will, itselft, become the classic example for new, emerging engineering technologies.

References

[Bsales 2000] Bsales, Jamie M., Mary E. Behr, Bill Howard and Ben Z. Gottesman "13th PC Magazine Service and Reliability Survey," *PC Magazine*, http://www.zdnet.com/pcmag/stories/reviews/0,6755,2594454,00.html, 2000.

[Denning 1992] Denning, P. J., "What is software quality?," *Communications of the ACM*, Vol. 35, No. 1, pp. 13-15. 1992

[Eriksson 1996] Eriksson, Inger V., Fred McFadden, and Anne M. Tiittanen : "Improving Software Development Through Quality Function Deployment," in Wrycza, J. Zupancic (eds.), *Proceedings of the Fifth International Conference "Information Systems Development - ISD'96"*, pp. 523-534, Gdansk, Sept. 1996.

[IEEE 1990] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York, NY: 1990.

[ISO 1991] ISO/IEC 9126, Information technology – *Software product evaluation – Quality characteristics and guidelines for their use, First edition*, International Organization for Standardization (Ed.), Geneva, Switzerland, 1991.

[Jorgensen 2000] Jorgensen, Alan A. and James A. Whittaker, "An Application Program Interface (API) Testing Method" *Conference Proceedings, STAR EAST 2000, International Conference on Software Testing Analysis and Review, May 1-5, 2000, Orlando, Florida, USA*, Software Quality Engineering, Orange Park, Florida, 2000.

[Pettichord 2001] Pettichord, Bret, "Design for Testability" *Proceedings, Software Test Automation Conference*, San Jose, CA March 2001.

[Vranken 1996] Vranken, H.P.E.; M.F.Witteman,and R.C. van Wuijtswinkel, "Design for testability in hardware-software systems." *IEEE Design and Test of Computers*. 13, nr. 3, pp. 79-87, 1996.

Using Oracles in Test Automation

Douglas Hoffman, BACS, MSEE, MBA
Software Quality Methods, LLC.
24646 Heather Heights Place
Saratoga, California 95070-9710
Phone 408-741-4830
Fax 408-867-4550
doug.hoffman@acm.org
www.SoftwareQualityMethods.com

Douglas Hoffman is an independent consultant with Software Quality Methods, LLC. He has been in the software engineering and quality assurance fields for 30 years and now is a management consultant in strategic and tactical planning for software quality. He is a past Chairman of the Santa Clara Valley Software Quality Association (SSQA), a Task Group of the American Society for Quality (ASQ) and has just completed two years as Chairman of the Silicon Valley Section of ASQ. He has been a presenter and participant at dozens of software quality conferences and has been Program Chairman for several international conferences on software quality. He is a member of the ACM and IEEE and is active in the ASQ as a Senior Member, participating in the Software Division, the Silicon Valley Section, and the Software Quality Task Group. He has earned a BA in Computer Science, an MS in Electrical Engineering, an MBA, a Certificate from ASQ in Software Quality Engineering, and has been a registered ISO 9000 Lead Auditor.

Douglas' experience includes consulting, teaching, managing, and engineering across the computer systems and software industries. He has twenty years experience in creating and transforming software quality and development groups, and has worked as an independent consultant for the last ten years . His work in corporate, quality assurance, development, manufacturing, and support organizations provides a broad technical and managerial perspective on the computer industry.

Key points to get from this paper:

- Automated tests should be planned and engineered
- Most automated tests need verification steps to be useful
- More powerful automated tests are made possible with oracles
- There are different types of oracles we can use
- There are several strategies for verification in automated tests

Using Oracles in Test Automation

PNSQC 2001

Douglas Hoffman

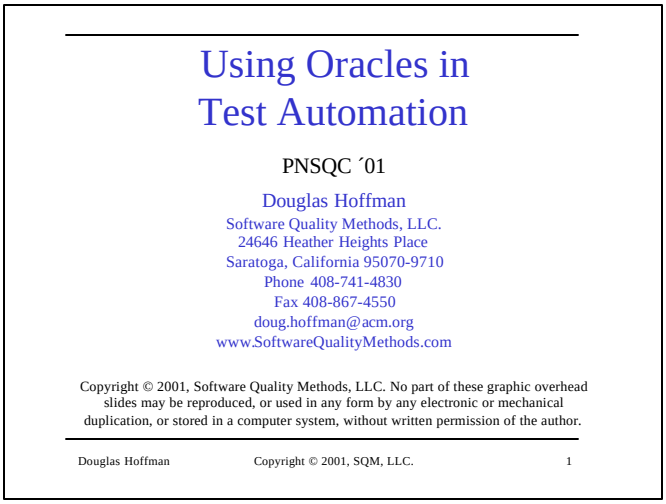
Summary

Software test automation is often a difficult and complex process. The most familiar aspects of test automation are organizing and running of test cases and capturing and verifying test results. When we design a test we identify what needs to be verified. A set of expected result values are needed for each test in order to check the actual results. Generation of expected results is often done using a mechanism called a test oracle. This paper describes the purpose and use of oracles in automated software verification and validation. Several relevant characteristics of oracles are included with the advantages, disadvantages, and implications for test automation.

Real world oracles vary widely in their characteristics. Although the mechanics of specific oracles may be vastly different, a few classes can be identified which correspond with automated test strategies. Oracles are categorized based upon the strategy for verification using the oracle. Thus, a verification strategy using a lookup table to compare expected and actual results can use the same type of oracle as one that uses an alternate algorithm implementation to compute them.

Background

Most software test automation begins with conversion of existing (manual) tests. In many instances the expected results are either embedded in the test or captured in a logging file for later verification. This approach is straightforward and can be used for a variety of tests. Tests automated in this way run automatically, but they are less likely to find errors than their manual counterpart for several reasons. A test automated this way will do the same thing each time it runs, especially since the inputs are provided by an automaton. The test will also verify only and exactly the results that we write into the test. Automated result comparison depends on having the results in a computer readable form.



Using Oracles in Test Automation

PNSQC ´01

Douglas Hoffman
Software Quality Methods, LLC.
24646 Heather Heights Place
Saratoga, California 95070-9710
Phone 408-741-4830
Fax 408-867-4550
doug.hoffman@acm.org
www.SoftwareQualityMethods.com

Copyright © 2001, Software Quality Methods, LLC. No part of these graphic overhead slides may be reproduced, or used in any form by any electronic or mechanical duplication, or stored in a computer system, without written permission of the author.

Douglas Hoffman Copyright © 2001, SQM, LLC. 1

For manual tests, a person provides the input and evaluates results, while automated tests use programs to do the work. A person will not do exactly the same thing the same way even when they try, while an automaton will tend to do exactly the same thing every time. Testers mis-key and correct their typing and people are also subject to variations in timing, so some possibly material characteristics can change simply because we aren't automatons. A person running manual tests can easily vary the test exercise and evaluate the responses of the software under test (SUT). Manually rerunning tests introduces new variations and exercises, improving the likelihood of finding new problems even with an old test.

A person running a manual test is also able to perceive unexpected behaviors for which an automated verification doesn't check. This is a powerful advantage for manual tests; a person may notice a flicker on the screen, an overly long pause before a program continues, a change in the pattern of clicks in a disk drive, or any of dozens of other clues that an automated test would miss. The author has seen automated tests "pass" and then crash the system, a device, or the SUT immediately afterwards. Although not every person might notice these things and any one person might miss them sometimes, an automated test only verifies those things it was originally programmed to check. If an automated test isn't written to check timing, it can never report a time delay.

We need to approach automated testing differently from manual testing if we want to get equal or better tests.

What Makes Automated Tests Different

In order to check the results, inputs to the SUT must be tracked and some means of generating a prediction of the resultant behaviors provided for some or all of the same dimensions. In a manual test, the tester usually controls or checks preconditions and inputs, and can quickly adjust the system when they encounter unexpected results. An automated test must rely upon the test design and system setup to control the important inputs. It must also include some mechanism for knowing or getting the expected results (typically from an oracle). Regardless of the test exercise, an automated test will be poor with poorly selected inputs or results, with poor results oracles, or if there is limited visibility into the relevant values.

Automated verification assumes that we know what to check to know whether the software did what it was supposed to do. It also assumes that we know the correct values for whatever we check. We identify what the software was supposed to change (or not change) when we design the test. Then we check it to confirm that it happened after we run the test. However, when the SUT fails it can change almost anything. Our tests should look for these failures because it does little good for a test to encounter an error and then ignore it. When manual testing, a person can be effective without knowing in advance exactly what the test results are. The tester will learn and adapt, checking different things at different times. If something doesn't seem right, the tester can dig into the details to figure out if the SUT behaved correctly. This is not true for automated tests. A tester can manually verify one or several conditions and data values, and doesn't have to do the same thing each time a test is run. For automated tests, this is established in advance.

The question of what to check is a big one that's often overlooked. We may have successfully entered an order and properly updated inventory counts and financial books, but, did we check to

see if there was any effect on other orders? We successfully added a user and set their permissions, but, did the permissions change for any other users? Software errors can cause an infinite variety of changes in the system and we can check only a few. With automated tests we define what to check in advance and limit the verification to exactly that.

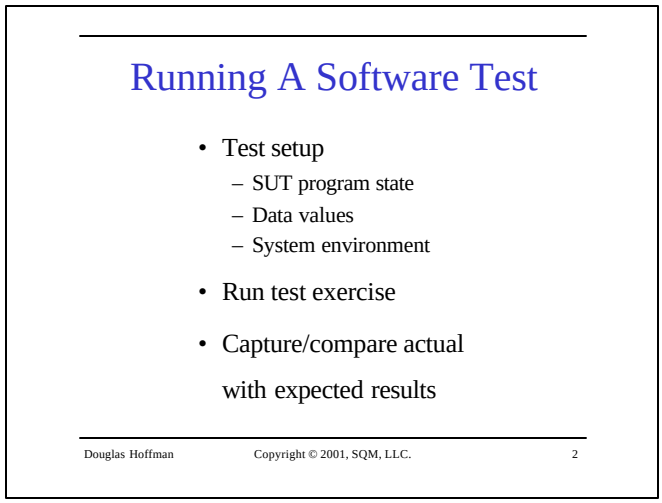
One reason we avoid checking many possible results is the difficulty of knowing what the results should be. This is the role of an oracle – to generate the expected result (or at least answer whether the actual result seems plausible). An oracle is critically important if we are to create automated tests that are equal or better than manual tests. We can generate millions of inputs in an automated test and verify proper SUT behavior using an oracle. Automated tests may be able to recover from unexpected responses if we have oracles that predict correct responses (and thus can show us a path to get back on track). Usually we need multiple oracles for these more sophisticated tests because we verify more than one thing.

Automated verification of results can also be quite difficult technically. How do you verify that the sound track synchronized with the motion picture? How do you verify that the image printed on the page is what you expected? Testers' perceptions can easily and effectively analyze results that we have a tremendous amount of difficulty getting a computer to verify. Some test automation problems are not cost effective to solve today.

This is not to say that automated tests aren't useful. Automated tests can be very powerful for finding certain kinds of errors. Manually rerunning the same tests every time anything changes is time consuming and boring for people, but machines are designed for doing this kind of task. People are also easily trained about what to expect from a test and can cognitively miss seeing errors after only a few repetitions. Machines do what we tell them to do, as many times as we want. And for massive numbers of data points, test iterations, and combinations there may not be any way to run tests except using automation. Some testing problems simply cannot be solved manually, such as performance analysis and system load testing.

Software Test Models

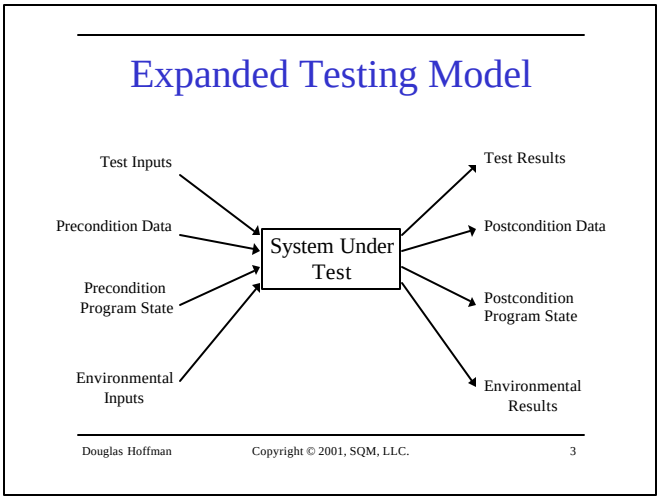
A software test consists of three steps: setting up the conditions in the system, providing stimulation to the SUT, and observing the results. This applies for manual and automated tests (shown in **Slide 2**). The setup creates the conditions necessary for some errors to manifest (assuming the errors are there). The test run exercise takes the SUT through the suspect code. Then we can confirm whether there are errors in the software.



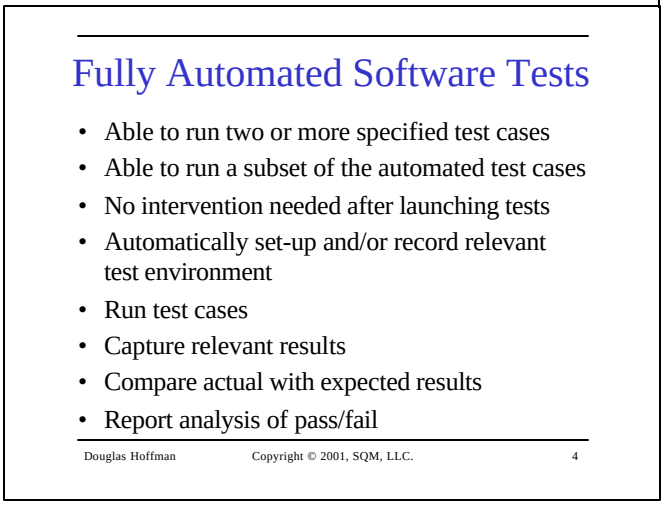
The test setup is often neglected, assuming that a test will work from whatever state the SUT, data, and system are in. Yet, it is obvious that software will do things differently based on

the starting state of the SUT and system. Is the SUT running? Is the account we are accessing already in the database, or do we need to add it? Is an error dialog currently on the screen? What permissions does the current user have? Are all the necessary files on the system? What is the current directory? Testers note and correct for all such conditions when manually testing. But these are potentially major issues for automated tests.

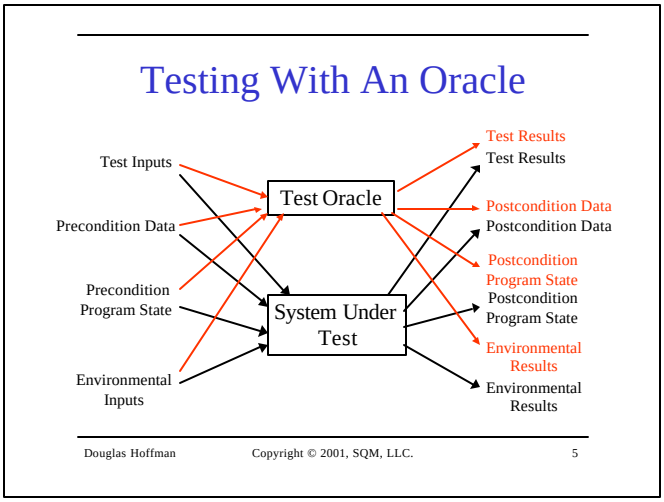
Although the test designer typically is conscious only of the values directly given to the SUT, the SUT behavior is influenced by its data, program state, and the configuration of the system environment it runs in. Test setup consists of monitoring or controlling things in all these domains. The domains include the data and program state information, which are somewhat manageable by the software test automation management system and within the automated tests. The environment, however, gets very difficult to scope out and manage. The SUT may behave differently based on the computer system environment; operating system version, size of memory, speed of the processors, network traffic, what other software is installed, etc. Even more difficult to analyze or manage is the physical environment – I've worked on errors due to temperature, magnetic fields, electrostatic discharge, poor quality electric grounding, and other physical environmental factors that caused software errors. These errors may be unusual, but they occur, so we need to keep them in mind when automating tests. **Slide 3** illustrates a model of actual inputs and results in a software test.



The term "automated software test" has many different meanings, depending upon the speaker and context. For the purposes of this paper, automated software testing has the eight characteristics shown in ***Slide 4****. The test consists of performing some exercise of the SUT, observing some results, comparing them with expected result values, and reporting the outcome.*



The running of an automated test exercise is often the easiest part of testing. Given that the SUT, data, and system are in the proper initial state, the automated test can feed the test data into the SUT[1]. (For the purposes of this paper, I'm going to assume that it is that simple[2].) The exercise puts the SUT through its paces and either encounters errors or not.



Some of the biggest difficulties in software test automation are in knowing what results are expected from the SUT. There are many issues with the huge number of potentially relevant results and how to record them. As just described, we are dealing with multiple input domains, and not surprisingly, the same domains can be effected by the SUT. (This is especially true when we consider that there may be errors in the SUT, so the actual result is outside of the expected realm.) Often, it is extremely difficult to predict what the SUT should do and what outcomes are expected, even for the set of expected results. **Slide 5** illustrates a model incorporating an oracle to predict expected results from actual inputs.

Several observations can be made from the model. Different types of oracles are needed for different types of software and environments. The domain, range, and form of input and output data varies substantially between programs. Most software has multiple forms of inputs and results so several oracles may be needed for a single software program. For example, a program's direct results may include computed functions, screen navigations, and asynchronous event handling. Several oracles may need to work together to model the interactions of common input values. If we consider a word processor, pagination changes are based upon the data being displayed and characteristics such as the page width, point size, page layout, and font. In Windows, the current printer driver also affects the pagination even when nothing is printed. Just changing the selected printer to one with a different driver can change the pagination of a Word document. Although an oracle may be excellent at predicting certain results, only the SUT running in the target environment will process all of the inputs and provide all of the results.

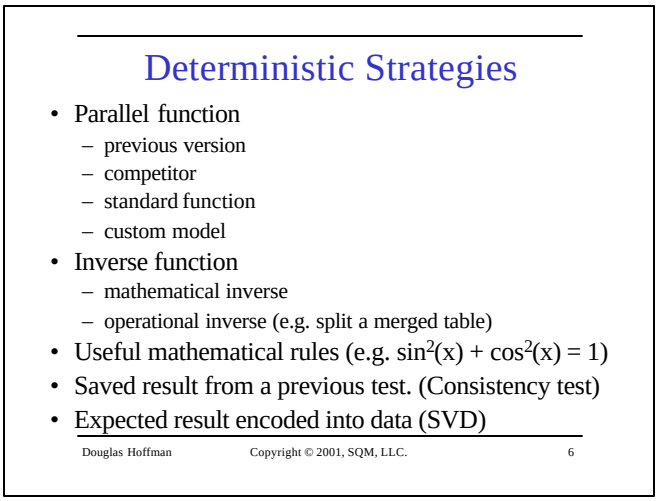
Characteristics of Oracles

When we think of software testing and the test results, it's usually in a binary sense: the result is right or wrong; the test passed or failed. We generally don't allow for "maybe it's OK" or "possibly right" as outcomes. We consider the test outcomes to be deterministic; there is one right answer [and we know what it is]. **Slide 6** lists some examples of such deterministic verification strategies. Each example provides some means for determining whether or not a test

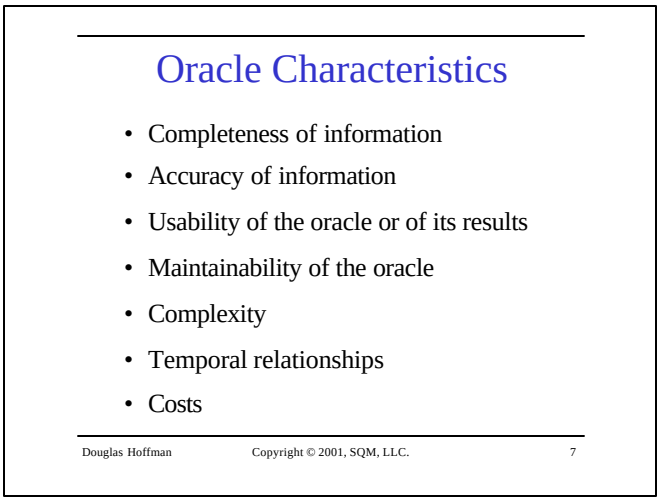
[1] "data" in this case includes input data, control information, and whatever else is needed for the test to stimulate the SUT.

[2] Much of the work in test automation to date has focused on the mechanics of feeding and manipulating data and controls in the test exercise. The focus here is on oracles and verification of the test results.

result is correct. It is useful to note that although the strategy allows us to pass or fail a particular result, in many cases there are ways that the SUT can give us a wrong result, and yet the test can pass (e.g., an undetected error in the previous version or the competitor's product).



There are several interesting characteristics relating an oracle to the SUT. **Slide 7** provides a list of some useful characteristics based on the correspondence between the oracle and the SUT. The results predicted by an oracle can range from having almost no relationship to exact duplication of the SUT behaviors. Completeness, for example, can range from no predictions (which may not be very useful) to exact duplication in all results categories (an expensive reimplementation of the SUT).



Completeness of information:

- Input Coverage
- Result Coverage
- Function Coverage
- Sufficiency
- Types of errors possible

Accuracy of information:

- How similar to SUT
  - Arithmetic accuracy
  - Statistically similar
- How independent from SUT
  - Algorithms
  - Sub-programs & libraries
  - System platform
  - Operating environment
  - Close correspondence makes common mode faults more likely and reduces maintainability
- How extensive
  - The more ways in which the oracle matches the SUT, i.e. the more complex the oracle, the more errors
- Types of possible errors
  - Misses actual wrong value
  - Flags correct data as an error
  - Some oracles may allow one or both types of errors

Usability of the oracle or of its results:

- Form of information
  - Bits and bytes
  - Electronic signals
  - Hardcopy and display
- Location of information
- Data set size
- Fitness for intended use
- Availability of comparators
- Support in SUT environments

Maintainability of the oracle:

- COTS or custom
  - Custom oracle can become more complex than the SUT
  - More complex oracles make more errors
- Cost to keep correspondence through SUT changes
  - Test exercises
  - Test data
  - Tools
- Ancillary support activities required

Complexity:

- Correspondence with SUT
- Coverage of SUT domains and functions
- Accuracy of generated results
- Maintenance cost to keep correspondence through SUT changes
  - Test exercises
  - Test data
  - Tools
- Ancillary support activities required

Temporal relationships:

- How fast to generate results
- How fast to compare
- When is the oracle run
- When are results compared

Costs:

- Creation or acquisition costs
- Maintenance of oracle and comparitors
- Execution cost
- Cost of comparisons
- Additional analysis of errors
- Cost of misses
- Cost of false alarms

The more complete and accurate an oracle is, the more complex it has to be. Indeed, if the oracle exactly predicts all results from the SUT it will be at least as complex. In some cases an oracle is more complex than the SUT when the simulators, operating systems, etc., are all considered. The better that an oracle provides expected results, the more complex it is and the more likely that detected differences are due to errors in the oracle rather than the SUT. Likewise, the more an oracle predicts about program state and environment conditions, the more sensitive the oracle is to changes in the SUT and operating environment. This dependence makes the oracle more complex and more difficult to maintain. It also means that errors may be missed because of common mode errors where both the SUT and the oracle generate the same wrong result due to sharing of a component with an error.

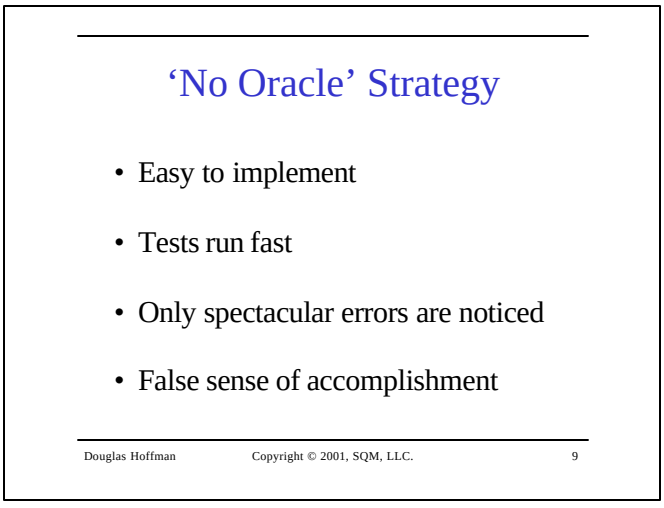
Oracle Strategies For Automated Test Verification

Four types of oracle strategies (and not using any oracle) are identified and outlined in **Table 1**. The strategies are labeled True, Consistency, Self Referential, and Heuristic. Each strategy is expanded upon below.

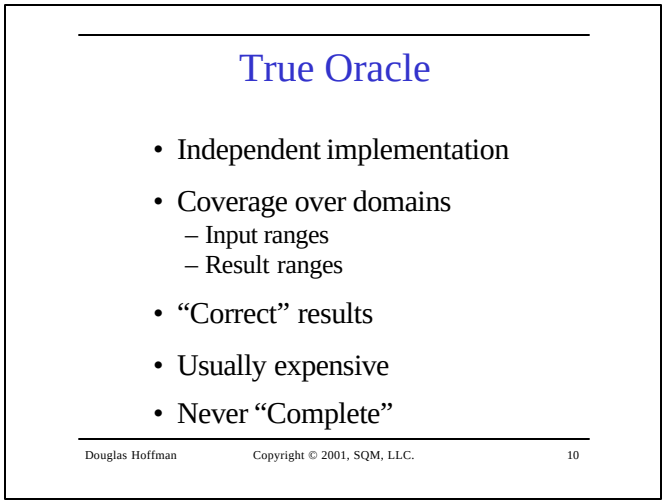
	No Oracle	True Oracle	Consistency	Self Referential (SVD)	Heuristic
Definition	• Doesn't check correctness of results	• Independent generation of all expected results	• Verifies current run results with a previous run (Regression Test)	• Embeds answer within data in the messages	• Verifies some characteristics of values
Advantages	• Can run any amount of data (limited only by the time the SUT takes)	• All encountered errors are detected	• Fastest method using an oracle • Verification is straightforward • - Can generate and verify large amounts of data	• Allows extensive post-test analysis • Verification is based on message contents • Can generate and verify large amounts of complex data	• Faster and easier than True Oracle • Often much less expensive to create and use
Disadvantages	• Only spectacular failures are noticed.	• Expensive to implement • Complex and often time-consuming when run	• Original run may include undetected errors	• Must define answers and generate messages to contain them	• Can miss errors • Can miss systematic errors

Table 1: Five Oracle Strategies

It is possible to automate the running of a test without checking results. I have encountered organizations that knew and planned such automated tests, and a few that simply hadn't thought to verify test results from their automation. This approach gets around the problems of false error reports and the costs of maintaining the oracle. It also has the advantage that it's easy, inexpensive, and tests run quickly. The major disadvantage is that only a few spectacular errors can be found this way. The automation may give



observers the impression that there is more value to the testing than there really is. There are occasions when the goal is to provide some exercise in the SUT, and the test outcomes are not important (e.g., the tests are needed just to provide a background load). However, effort and activity are not the same as accomplishment. (Running a useless exercise 1,000,000 times each night is still a useless exercise.)



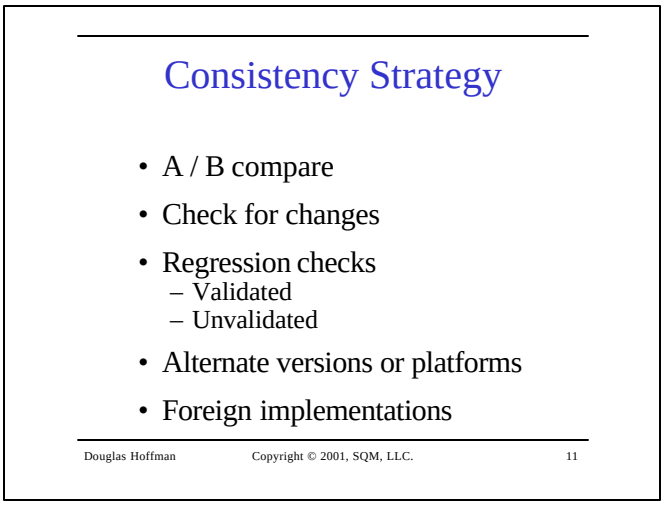
A true oracle faithfully reproduces all relevant results for a SUT using independent platform, algorithms, processes, compilers, code, etc. The same values are fed to the SUT and the oracle for verification. This type of oracle is well suited for verification of an algorithm or subroutine. For a given test case, all values input to the SUT are verified to be correct using the oracle's separate algorithm. The less the SUT has in common with the oracle, the more confidence in the correctness of the results (since common hardware, compilers, operating systems, algorithms, etc., may inject errors that effect both the SUT and oracle the same way). Automated test cases employing true oracles are usually limited by available machine time and system resources, not the oracle itself.

A true oracle may be slow or expensive to use, so tests may use a small sample to limit the amount of test data. One way this is done is for random selection of inputs within ranges where the oracle works. Different inputs can be generated each time the test is run by using a pseudo-random numbers (repeatable sequences of random numbers) to select the input values. Another small sample approach is done by creating a table of inputs with corresponding results from the oracle. Input values chosen from the table are fed to the SUT and the results verified from the table.

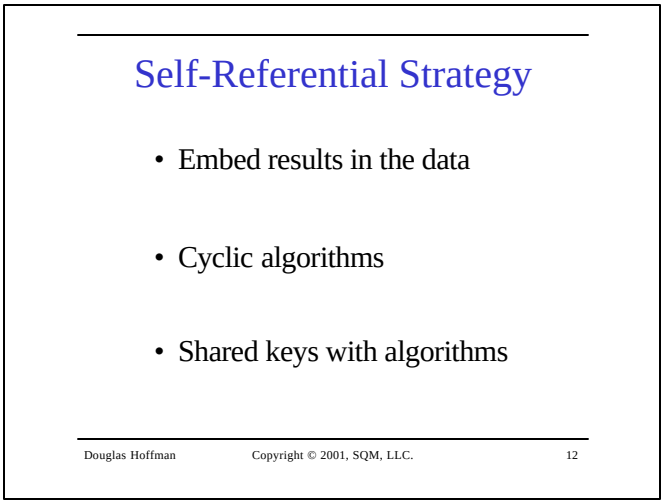
Note that true oracles do not have to be complete to be useful. In fact, a completely replicated system will not provide identical responses over all of the input and result domains described in **Slide 5**. An oracle that works for a subset of input values or covers only one characteristic result can be very effective. A test may cover only a small range of input values or we may check only that the correct sequence of screens appears without need for a complete oracle. Being a true oracle, however, means that for the subset of input values or the sequence of screens we test, the oracle correctly provides the expected results.

The consistency approach uses the results from one test run as the oracle for subsequent tests. Thus, we learn whether the results are the same as before, with differences likely the result of errors. This is probably the most used strategy for automated regression tests, as it is particularly useful for evaluating the effects of changes from one revision to another. Although it doesn't tell us whether the results are actually correct, it does expose differences or changes. The oracle can be a simulator, equivalent product, software from an alternate platform, or an early version of the

SUT. Because we don't need to know if the results are correct, we can use pseudo-random numbers to generate huge volumes of results to be compared. At the same time, the values being compared can include intermediate results, call trees, raw internal data values, or any other data extracted from the SUT. Comparing results between the SUT and the oracle tells us of any changes, which indicates something was fixed or broken. Although historic faults may remain when this technique is used, new faults and side-effects are often exposed and fixes are confirmed.



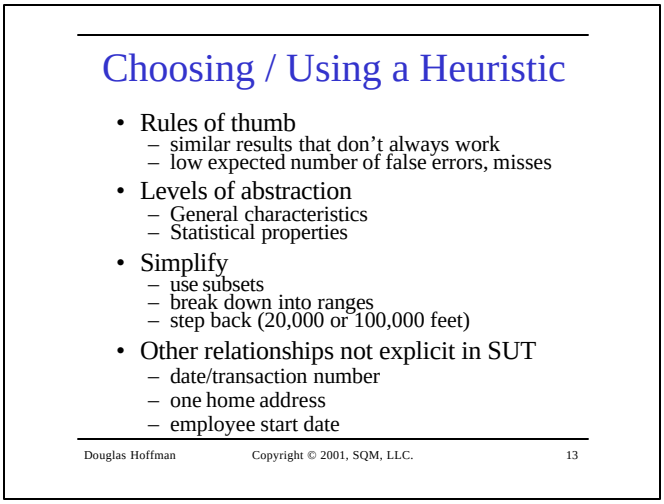
A self-referential strategy builds the expected results (answers) into the data as part of the test mechanism. For example, when testing data communications, we might include the expected communications protocol information in the message we send, so a receiver could confirm that the envelope data matches. In a test for a data base engine, a data field could describe the data base linkages expected between fields or records. The random number "seed" could be included in the randomly generated data set so that the random number series can be rerun. In this strategy, the tests are designed so that they create records with specific characteristics, and those characteristics are included within the records themselves. We then use pseudo-random numbers to generate arbitrary populations of test data. An independent analysis can be run to identify problems or inconsistencies.



A heuristic is a general "rule of thumb" we can use to quickly assess whether the result is likely to be correct or incorrect. It is not guaranteed to find all errors and it may flag correct results as errors. This strategy verifies results using simpler algorithms or consistency checks based on a heuristic. For example, a heuristic strategy for a USA zip code might check that the values have five or nine digits. A heuristic for a *sin()* function could use the mathematical identity that $sin^2(x) + cos^2(x) = 1$ to verify it (assuming that the implementation of *sin()* and *cos()* are not based on that relationship). Although the heuristic approach may accept results that are incorrect or reject results that are correct, the oracle is easy to implement (especially when compared to a true oracle), runs much faster, and can be used to quickly find many classes of errors.

A heuristic may be based on incidental relationships associated with the data. For example, if transaction numbers are generated sequentially when the transactions begin, then sorting by date

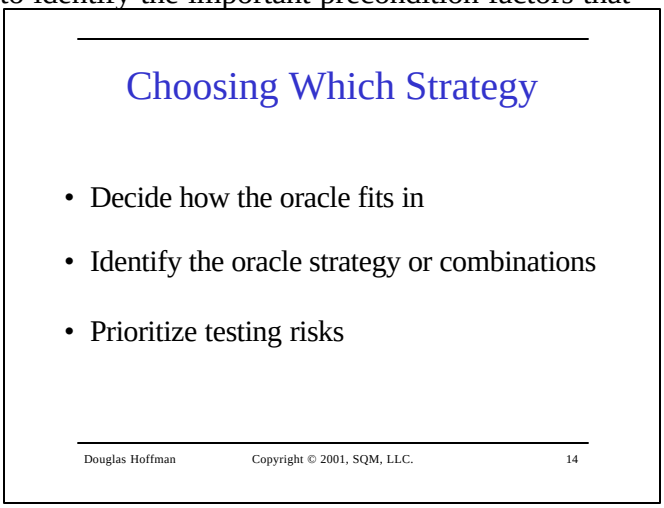
and sorting by transaction number should yield the same results. Similarly, a current employee's start date should be between their birth date and today. An employee's dependent children should be younger than the employee (but recognize that there are circumstances when the dependent may be older).



One special type of heuristic strategy uses a statistical approach based on relationships in the population statistics between inputs and results. For example, if we use uniform random numbers to generate random, symmetric, geometric figures at random locations on a page, then we might compute the mean location (X and Y coordinate) for each dot on the page. For a large number of generated figures we would expect the mean location to be the middle of the page. Likewise, we might expect the standard deviation and skew for the dots to be similar to the standard deviation and skew for the random numbers. This heuristic could allow us to create some automated tests that generate large numbers of random figures and still give us some assurance that the results are reasonable.

Choosing A Strategy

Understand what you are testing well enough to identify the important precondition factors that need to be monitored or manipulated for automated testing. Also identify the important results that need to be verified to know whether the SUT passed or failed any test. Based on the important input and result factors, decide on what oracles are needed to be able to run useful tests.



The oracle strategies will be based on availability of existing oracles, the ease of creation of new oracles, and recognition of applicable heuristics. A combination of strategies can be very effective at providing a basis for powerful automated tests.

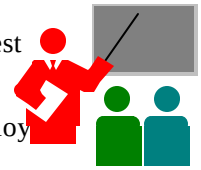
As with all testing tasks, it is important to understand the trade-offs between the various risks and the costs involved in testing. It is very easy to get lost in the wonderful capabilities of automated tests and lose sight of the important goal of releasing high quality software.

Conclusion

There are techniques to make very powerful automated tests. Test result oracles can generate predicted results or verify the correctness of actual results for automated tests. There are different types of oracles and strategies for automated verification of results. When used well, oracles can result in better, more powerful automated tests. These automated tests can be engineered from models of the SUT, its inputs and results, and test results oracles.

Summary

- Automated tests can be powerful
- Test oracles are critical factors in making good automated tests
- There are different types of test oracles available
- There are many ways to employ test oracles



Douglas Hoffman Copyright © 2001, SQM, LLC. 15

References

Hoffman, Douglas; “A Taxonomy of Test Oracles” Quality Week 1998.

Hoffman, Douglas; “Heuristic Test Oracles“ Software Testing and Quality Engineering Magazine, Volume 1, Issue 2, March/April 1999.

Hoffman, Douglas; “Test Automation Architectures: Planning for Test Automation” Quality Week 1999.

Hoffman, Douglas; “Mutating Automated Tests” STAR East, 2000.

Nyman, Noel; “Self Verifying Data - Validating Test Results Without An Oracle” STAR East, 1999.

Using Oracles in Test Automation

PNSQC ´01

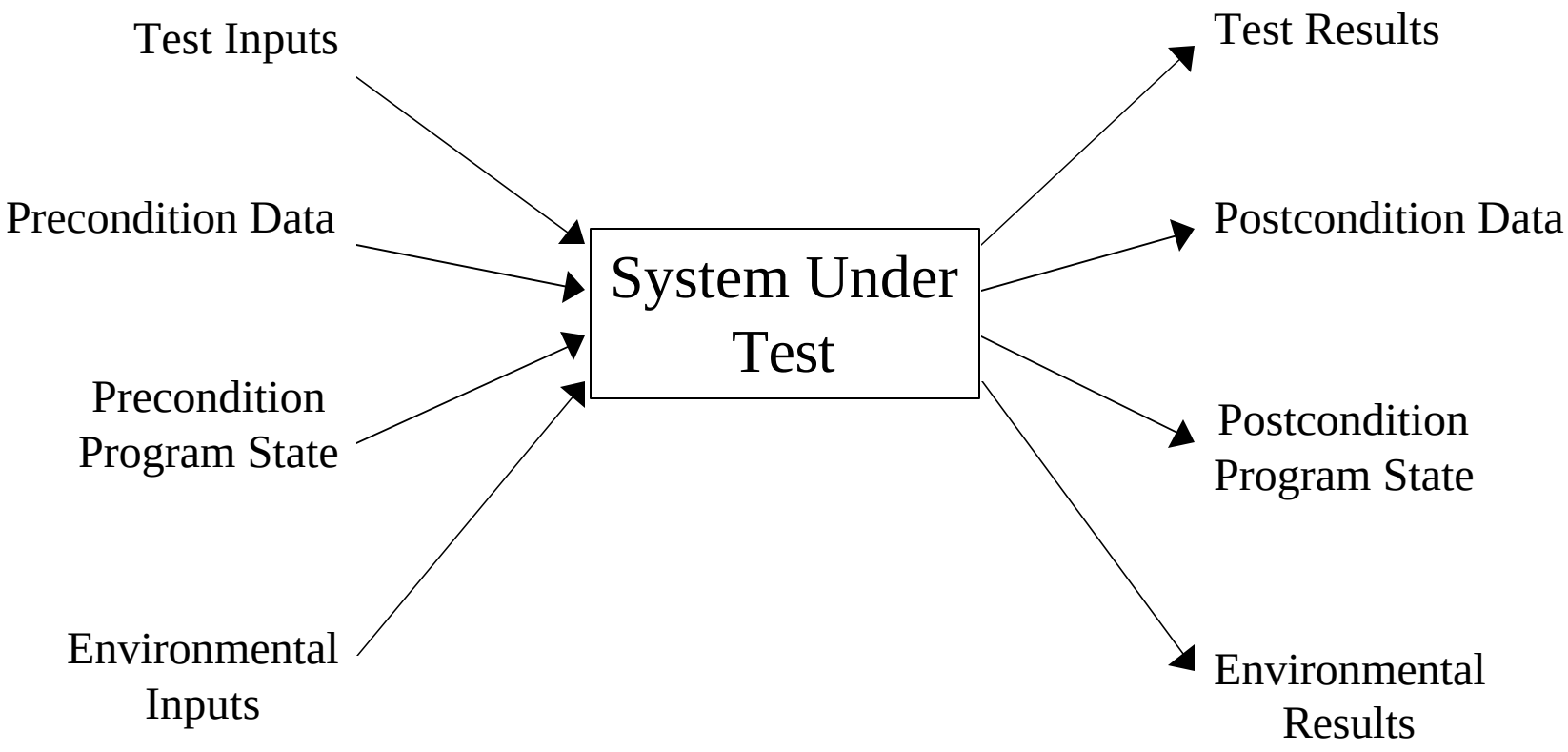
Douglas Hoffman
Software Quality Methods, LLC.
24646 Heather Heights Place
Saratoga, California 95070-9710
Phone 408-741-4830
Fax 408-867-4550
doug.hoffman@acm.org
www.SoftwareQualityMethods.com

Running A Software Test

- Test setup
  - SUT program state
  - Data values
  - System environment
- Run test exercise
- Capture/compare actual

  with expected results

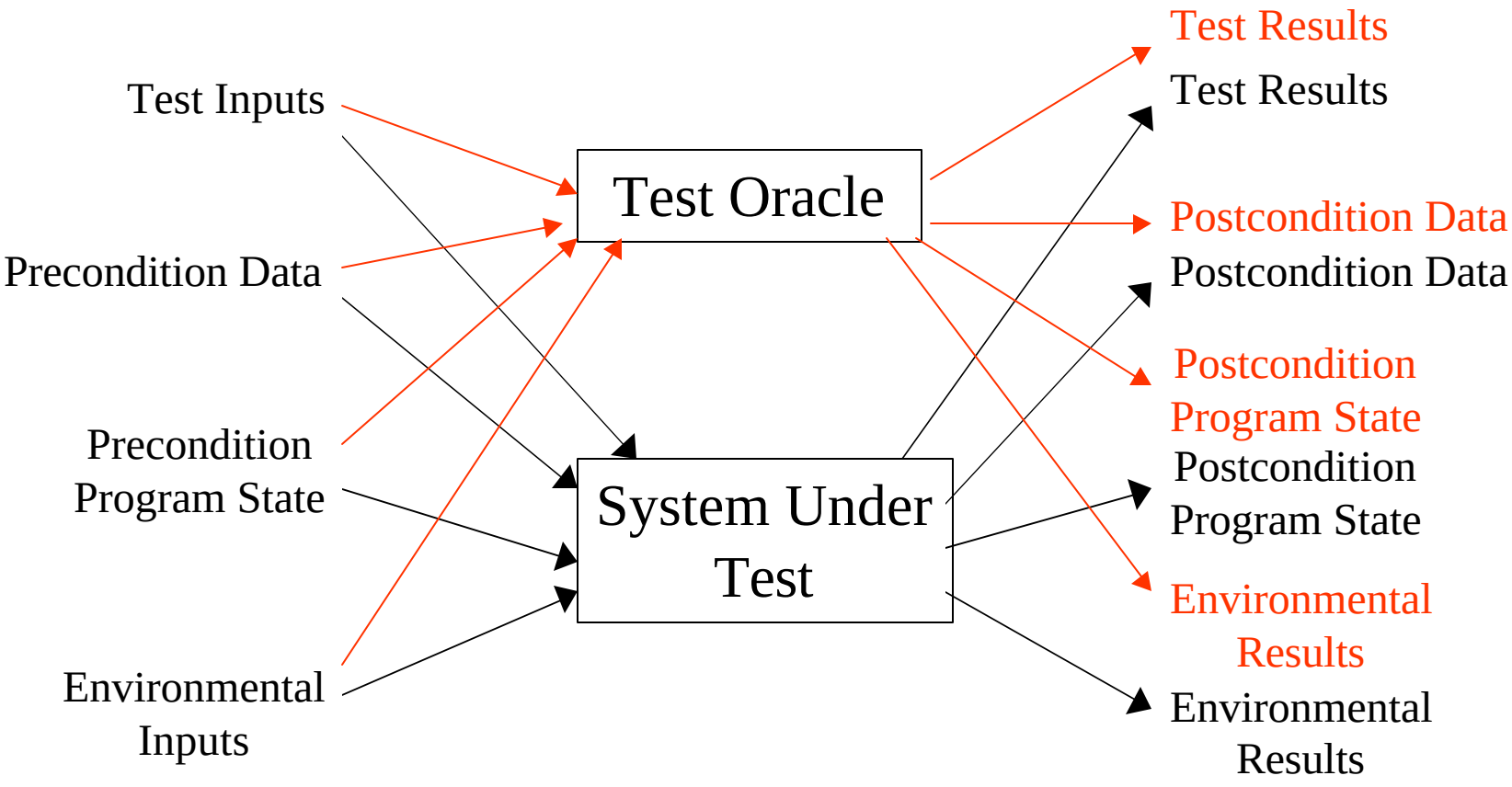
Expanded Testing Model



Fully Automated Software Tests

- Able to run two or more specified test cases
- Able to run a subset of the automated test cases
- No intervention needed after launching tests
- Automatically set-up and/or record relevant test environment
- Run test cases
- Capture relevant results
- Compare actual with expected results
- Report analysis of pass/fail

Testing With An Oracle



Deterministic Strategies

- Parallel function
  - previous version
  - competitor
  - standard function
  - custom model
- Inverse function
  - mathematical inverse
  - operational inverse (e.g. split a merged table)
- Useful mathematical rules (e.g. $\sin^2(x) + \cos^2(x) = 1$)
- Saved result from a previous test. (Consistency test)
- Expected result encoded into data (SVD)

Oracle Characteristics

- Completeness of information
- Accuracy of information
- Usability of the oracle or of its results
- Maintainability of the oracle
- Complexity
- Temporal relationships
- Costs

Oracle Strategies for Verification

	No Oracle	True Oracle	Consistency	Self Referential (SVD)	Heuristic
Definition	- Doesn't check correctness of results	- Independent generation of all expected results	- Verifies current run results with a previous run (Regression Test)	- Embeds answer within data in the messages	- Verifies some characteristics of values
Advantages	- Can run any amount of data (limited only by the time the SUT takes)	- All encountered errors are detected	- Fastest method using an oracle - Verification is straightforward - Can generate and verify large amounts of data	- Allows extensive post-test analysis - Verification is based on message contents - Can generate and verify large amounts of complex data	- Faster and easier than True Oracle - Often much less expensive to create and use
Disadvantages	- Only spectacular failures are noticed.	- Expensive to implement - Complex and often time-consuming when run	- Original run may include undetected errors	- Must define answers and generate messages to contain them	- Can miss errors - Can miss systematic errors

‘No Oracle’ Strategy

- Easy to implement
- Tests run fast
- Only spectacular errors are noticed
- False sense of accomplishment

True Oracle

- Independent implementation
- Coverage over domains
  - Input ranges
  - Result ranges
- “Correct” results
- Usually expensive
- Never “Complete”

Consistency Strategy

- A / B compare
- Check for changes
- Regression checks
  - Validated
  - Unvalidated
- Alternate versions or platforms
- Foreign implementations

Self-Referential Strategy

- Embed results in the data
- Cyclic algorithms
- Shared keys with algorithms

Choosing / Using a Heuristic

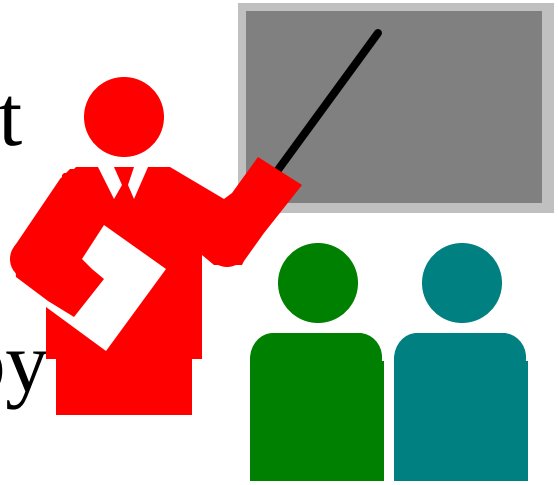
- Rules of thumb
  - similar results that don't always work
  - low expected number of false errors, misses
- Levels of abstraction
  - General characteristics
  - Statistical properties
- Simplify
  - use subsets
  - break down into ranges
  - step back (20,000 or 100,000 feet)
- Other relationships not explicit in SUT
  - date/transaction number
  - one home address
  - employee start date

Choosing Which Strategy

- Decide how the oracle fits in
- Identify the oracle strategy or combinations
- Prioritize testing risks

Summary

- Automated tests can be powerful
- Test oracles are critical factors in making good automated tests
- There are different types of test oracles available
- There are many ways to employ test oracles



Patterns of Software Test (PoST) Overview - abstract

This paper gives a brief overview of patterns. Patterns have proved to be an excellent vehicle to capture expertise and make it available to non-experts. Pattern names collectively form a vocabulary that helps testers and developers communicate better. People understand a system more quickly when it is documented with the patterns it uses.

It begins with Christopher Alexander and his architectural patterns, and gives as an example the Window Place Pattern. It then discusses the crossover to Software Patterns, and the communities and conferences around it. The Proxy Pattern from the *Design Patterns* book is presented as an example. Next, System Test Pattern's Multiple Processors pattern and Testing Object Oriented Software's Category Partition Pattern are used as examples in a review of the earlier work on software testing patterns.

The Workshop on Patterns of Software Test (PoST) is discussed including a brief discussion of what people need to learn to write patterns, with pointers to several materials on this subject. The Architecture Achilles Heel Analysis Pattern and Test Result patterns as examples of patterns from PoST are presented. Finally a mention of where software test patterns may be going is given.

Keith Stobie

Keith plans, designs, and reviews software architecture and tests for Microsoft. Keith directed and instructed in QA and Test process and strategy at BEA Systems. His most recent project was BEA WebLogic Collaborate, and previously WebLogic Enterprise. Keith was Test Architect at Informix, designing tests for the Extended Parallel Server product, and Manager of Quality and Process Improvement. With over 20 years in the field, Keith is a leader in testing methodology, tools technology, and quality process. He is a qualified instructor for Systematic Software Testing and software inspections. Keith is active in the software task group of ASQ, participant in IEEE 2003 and 2003.2 standards on test methods, published several articles, and presented at many quality and testing conferences. Keith has a BS in computer science from Cornell University.

ASQ certified Software Quality Engineer,
Member: ACM, IEEE, ASQ

Keith Stobie
Test Architect

16541 Redmond Way PMB 321-C U6
Redmond, WA 98052-4482
e-mail: pnsqc@stobie.org

Table of Contents

1 Patterns of Software Test (PoST) – an Overview

Keith Stobie

This paper is a report on the PoST workshops held January 3-5, 2001 at Rational in Lexington, MA and April 11-13, 2001 in Melbourne, FL near the Florida Institute of Technology.

This paper gives a brief overview of patterns. It begins with Christopher Alexander and his architectural patterns, and gives as an example the Window Place Pattern. It then discusses the crossover to Software Patterns, and the communities and conferences around it. The Proxy Pattern from the *Design Patterns* book [Gamma+ 95] is presented as an example. Next, System Test [DeLano+00] Pattern's Multiple Processors pattern and Binder's Category Partition Pattern [Binder99] are used as examples in a review of the earlier work on software testing patterns.

The workshop on Patterns of Software Test (PoST) is discussed, including a brief discussion of what people need to learn to write patterns, with pointers to several materials on this subject. The Architecture Achilles Heel Analysis Pattern and Test Result patterns as examples of patterns from PoST are presented. Finally a mention of where software test patterns may be going is given.

The introductory material and appendices are culled from other authors who have already eloquently and concisely expressed the concepts of patterns. In particular, Brian Marick's web site [Marick01] on test patterns (for PoST) has many good background explanations from which I have borrowed.

1.1 What are Patterns?

The term "pattern" is derived from the writings of the architect Christopher Alexander as it relates to urban planning and building architecture. Patterns are a way of helping designers. "A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts" [Appleton00]

From the Patterns Definitions section of the Patterns Home Page [PHP]:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.

1.1.1 Why Patterns?

When done well, patterns accomplish at least three things:

- They provide a **vocabulary** for problem-solvers. "Hey, you know, we should use a Null Object."
- They focus attention on the **forces** that make up a problem. That allows designers to better understand when and why a solution applies.

- They encourage **iterative** thinking. Each solution creates a new context in which new problems can be solved. [Marick01]

1.1.2 History

There was some limited prior work of using patterns for software testing prior to this year, as will be discussed later in this paper under Test Patterns. In late 2000, a few people in the software testing community decided to see if patterns could be applied to the software testing domain. This resulted in the Patterns of Software Testing (PoST) workshops sponsored by Rational and the Florida Institute of Technology.

For a slightly more detailed history of patterns and software patterns that preceded software test patterns, see 5 Appendix: Pattern History.

1.1.3 Pattern Contents

There are many ways of expressing patterns. If multiple patterns are all expressed the same way, they are said to follow a Pattern Template. Alexander in his writing usually had a narrative form similar to the following:

IF you find yourself in CONTEXT
for example EXAMPLES,
with PROBLEM,
entailing FORCES
THEN for some REASONS,
apply DESIGN FORM AND/OR RULE
to construct SOLUTION
leading to NEW CONTEXT & OTHER PATTERNS

Alexander also generally tried to illustrate all of his patterns and the *Design Patterns* book [Gamma+95] also used diagrams to illustrate its patterns.

If a set of patterns following the same template is introduced it forms a Pattern Catalog, like *Design Patterns*. If the patterns are tightly related, building on one another, they form a Pattern Language. For more information about patterns in general, see 6 Appendix: Pattern Introductions.

2 Pattern Examples

2.1 Alexander's Window Place

I pull my first example from Alexander himself in *A Pattern Language: Towns, Buildings, Construction* [Alexander77]. At the PoST workshops, Brian Marick introduced patterns using the Window Place Pattern. Brian is a software tester, but he was able to leverage the mastery of an acclaimed architect. He had a living room which nobody used. There was a big sofa in front of a large picturesque window. By studying *A Pattern Language*, he could see that his current solution did not resolve the **forces** well:

"1. He wants to sit down and be comfortable.
2. He is drawn toward the light.**"**

When people sat in the sofa, they faced away from the light which forced them away from looking through the window. The pattern clearly identified his context:

> "a room without a window place may keep you in a state of perpetual unresolved conflict and tension—slight, perhaps, but definite."

So he read the solution:

> **"**any window with a reasonably pleasant view can be a window place, provided that it is taken seriously as a space, a volume, not merely treated as a hole in the wall. Any room that people use often should have a window place."

He then applied it using some of the other patterns pointed to by this Window Place Pattern.

His living room used to be avoided, but now, by rearranging the furniture to create a Window Place, it is inviting and frequently used. An abridged version of the pattern is given in 4 Appendix: Patterns.

2.2 Beyond Architecture Patterns

In the late 1980's a group of software practitioners took the concept of Patterns from Alexander and applied to Software Design [see 5 Appendix: Pattern History]. The approach has become quite popular. Over the last decade, the software community and others have starting applying Patterns to an even wider scope. They are being used for processes, etc.

- A Development Process Generative Pattern Language, James Coplien
  http://www1.bell-labs.com/user/cope/Patterns/Process/index.html
- A Risk Management Catalog, Alistair Cockburn
  http://members.aol.com/acockburn/riskcata/risktoc.htm
- CHECKS Pattern Language of Information Integrity, Ward Cunningham
  http://c2.com/ppr/checks.html

2.2.1 Software Design Patterns

The book *Design Patterns: elements of reusable object-oriented software* [Gamma+95] is a seminal work and provides:

> "descriptions of **communicating objects and classes** that are customized to solve a general design problem in a particular context"

In *CS 342: Patterns and Frameworks* [Levine+] :

- Design patterns represent solutions to problems that arise when developing software within a particular context
  - "Patterns == problem/ solution pairs in a context"
- Patterns capture the static and dynamic structure and collaboration among key participants in software designs
  - They are particularly useful for articulating how and why to resolve non-functional forces
- Patterns facilitate reuse of successful software architectures and designs

Design Patterns have been widely adopted across the object-oriented software community. On the web and in many books people continue to identify, define, and refine patterns relating to communicating objects and classes. Learning Design Patterns deeply is a good way to understand the power of patterns for software.

2.2.2 Design Patterns Template

Because of its emphasis on communicating objects and classes, *Design Patterns* chose a very different format for presentation. Rather than the almost totally prose style of Alexander, it introduced a template having fixed sections allowing quick referencing of each part. It has a Structure section with Class Diagram, Participants section, and Collaborations section that are relatively unique to object-oriented software. Its "Sample Code and Usage" section could apply to almost any software oriented patterns.

You can contrast the sections in the [Gamma+95] template, shown in 3 Appendix: Templates under 3.1 Design Patterns Template, with the narrative style used by Alexander given previously in Pattern Contents.

As an example of the use of the Design Patterns Template, Proxy Pattern is one very commonly used. It is one of the briefest patterns as shown in 4 Appendix: Patterns. The Proxy pattern is used when one object approximates another. If you look at the pattern you see how tightly the Design Patterns template works with object-oriented code.

2.3 Test Patterns

2.3.1 Why test patterns?

We believe that testers lack a useful vocabulary, are hampered by rigid "one size fits all" methodologies, and face many problems whose solutions are under-described in the literature. Patterns can help with all of these things.
Moreover, the community of pattern writers is a healthy one that regularly spawns new and useful ideas. We testers should link up with it, and we might find its style of work useful as we look for new ideas. [Marick01]

Some of the pioneering work attempting to use patterns to describe software testing includes:

- *System Test Pattern Language* [DeLano+00]
- *Testing Object-Oriented Systems: Models, Patterns, and Tools* [Binder99]
- *Pattern Language for Testing Object-oriented Software* [Firesmith96]
  Firesmith used a variant of the Design Patterns Template, called a Pattern Language for Object-Oriented Testing (PLOOT) to describe both test drivers and test case design.
- *Parallel Architecture for Component Testing of Object-Oriented Software,* [McGregor96]

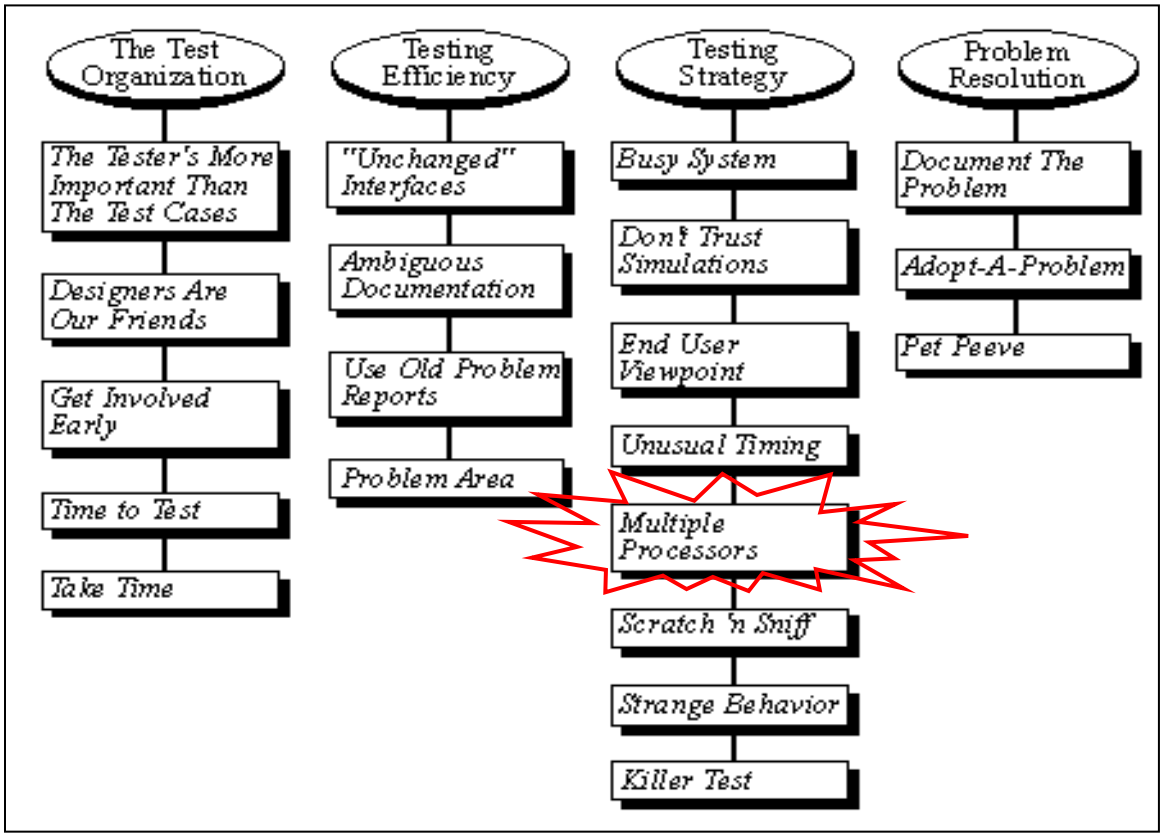
2.3.2 System Test Pattern Language

One of the test patterns with no object-oriented software testing association was the *System Test Pattern language* [DeLano+00] . It also shows a high level approach rather than a software or code centric approach.

"Testing of systems is presently more of an art than a science, even considering current procedures and methodologies that support a more rigorous approach to testing. This

becomes even more apparent during the testing of large, embedded systems that have evolved over time. To deliver the best possible product, the role of a System Tester has become vital in the lifecycle of product development. This pattern language has been derived from the experience of veteran System Testers to help System Testers evaluate the readiness of a product for shipping to the customer. Though these patterns have been derived from experience during the system test phase of the product lifecycle, many of the patterns are orthogonal to all testing phases. In addition to System Testers, these patterns may be useful to Designers and Project Managers.
These patterns have been grouped according to their usefulness in the system testing process: The Test Organization, Testing Efficiency, Testing Strategy, and Problem Resolution."

System Test Overview



Below, I pick the single smallest pattern I've seen, *Multiple Processors*, to illustrate the contents of this catalog. In addition to the headings shown below, they sometimes include: Alias, Context, and Forces. Note that none of the headings for these patterns correspond by name with any of those in Design Patterns. However, some headings are actually quite similar to Design Pattern sections. For example "Alias" heading corresponds directly to "Also Known As" section and "Context" heading corresponds to "Applicability" section.

2.3.3 Pattern: Multiple Processors

Problem	What strategy should be followed when System Testing a system comprised of multiple processors?
Solution	Test across multiple processors.
Resulting Context	Problems that occur in one processor will probably occur in other processors. Tests that pass on one processor may fail on another.
Rationale	When a problem is found in one processor, that feature will usually have problems running on other processors. A dirty feature is a dirty feature. Features that run on multiple processors aren't always designed to run on all processors.

2.3.4 Test Design Pattern Template

"*Testing Object-Oriented Systems: Models, Patterns, and Tools* [Binder99] is a guide to testing object-oriented systems. It is mainly about test design as software engineering and about the systems engineering challenges of test automation.

The book introduces the **test design pattern.** This new pattern template focuses explicitly on the key dimensions of test design: When is a particular test strategy appropriate? What kind of bugs will it find? How do you develop a test suite -- how should the implementation under test be modeled, and how are test cases produced from the model and its oracle? What kind of test automation works best? What are its advantages and disadvantages? Who has used this pattern? An overview of the test pattern template explains each of these sections in 3.2 Test Design Pattern Template [Binder99] and in http://www.rbsc.com/pages/TestPatternTemplate.htm."[1]

Significant new entries in the template are the **Fault Model** and **Strategy**. These have influenced some of the PoST-derived patterns. It uses an "Intent" section like *Design Patterns* [Gamma+95] and a "Context" heading like the *System Test Pattern Language* [DeLano+00] .

4 Appendix: Patterns provides an example from the book of the Category Partition Pattern. Notice the tight interweaving of the example into the text so that it is nearly impossible to separate the pattern from the example.

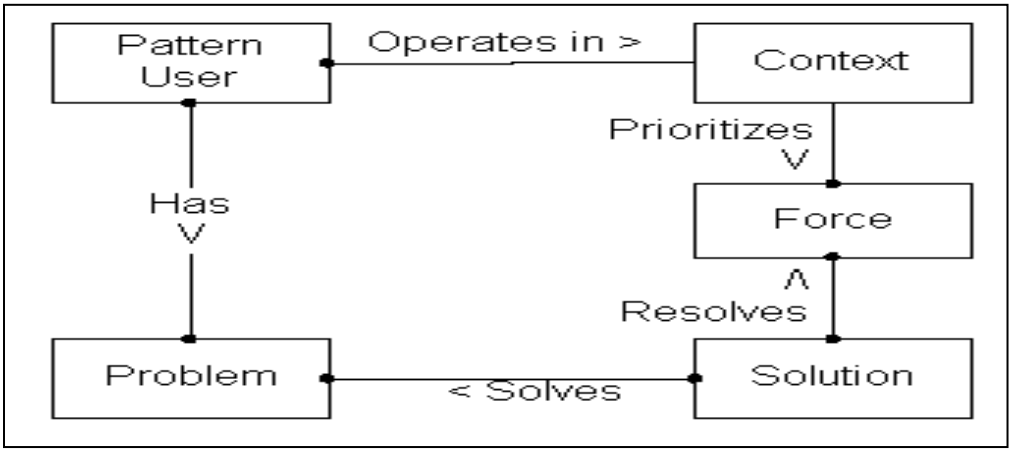
2.4 Workshops

Robert Binder used his pattern template in his pattern writing workshop at the May 2000 Quality Week. It produced four drafts of test design patterns in three hours from people who had *never* done any pattern writing before.

The first PoST workshops looked at Window Place Pattern and the Category Partition Pattern. Further, participants were expected to become familiar with some of the many

[1]From: http://www.rbsc.com/TOOSMPT.htm

templates for writing patterns. In particular we studied *A Pattern Language for Pattern Writing* [Meszaros+] which presented the following figure.



You can learn more about pattern writing and workshops in the 6 Appendix: Pattern Introductions under Pattern Writing and Workshops. Besides PoST, the original Pattern Languages conferences include:

- Pattern Language of Programs (PLoP) http://jerry.cs.uiuc.edu/~plop/
- Chili PLoP http://www.dreamsongs.com/ChiliPLoPInvite.html

Attendees of the first PoST workshop Jan. 3-5, 2001 at Rational in Lexington, MA included: James Bach (co-host), Scott Chase, Sam Guckenheimer (co-host), Elisabeth Hendrickson, Ivar Jacobson, Cem Kaner, Grant Larson, Brian Marick (co-host), Noel Nyman, Bret Pettichord, Johanna Rothman (facilitator), Keith Stobie, Paul Szymkowiak, (co-host), James Tierney, and James Whittaker.

Attendees of the second PoST workshop April 11-13, 2001 in Melbourne, FL included: Helene Astier, Hans Buwalda, Scott Chase, Elisabeth Hendrickson, Alan A. Jorgenson, Cem Kaner, Brian Marick, Florence Mottay, Melissa Mutkoski, Bret Pettichord, Nadim H. Rabbani, Jennifer Smith-Brock, Keith Stobie, Amit Singh, and Paul Szymkowiak.

The third PoST is Aug. 26-27, 2001 in Lexington, MA again. This after the submission of this paper for publication.

At the first two PoST workshops, we began by introducing patterns and pattern writing to the new attendees. The remaining days were spent writing and reviewing (workshopping) the patterns.

As Patterns for Software Testing is relatively new, we can see already the diversity of pattern templates used (Multiple Processors and Category Partition Pattern). Some participants of the PoST workshops based their templates on the Alexandrian form. Others based theirs on the work of [Meszaros+], [Binder99], or [DeLano+00] They also felt the need to extend, alter, or start from scratch the templates to express their patterns. There were nearly as many templates as people. Some of the templates were used to support multiple patterns.

One of the most well received patterns created and refined at the workshops is the Architecture Achilles Heel Analysis Pattern [Hendrickson+01].
This pattern, presented in 4 Appendix: Patterns, is used to identify areas for bug hunting relative to the architecture. Note the new headings of "Liabilities" and "Additional Benefits". The "Objective" is similar to *Design Pattern*'s "Intent". Like *Design Patterns* [Gamma+95], the figure associated with the solution is an example. The solution includes numerous example questions to ask.

I have also included the Test Results patterns Check as you Go Pattern and Batch check Pattern which I have written. The Test Results patterns solve the same problem, but in different contexts. Just from the number aliases you can tell they are popular solutions, although not presented in pattern form before.

These are samples from the start of a Pattern Language with eight patterns. The patterns closely follow the [Meszaros+] template and include anti-patterns. An **anti-pattern** is a pattern that tells how to go from a problem to a bad solution. A good anti-pattern also tells you why the bad solution looks attractive (e.g. it actually works in some narrow context), why it turns out to be bad, and what positive patterns are applicable in its stead. A well known example anti-pattern is **Big Bang Testing**.

2.5 PoST future

Besides at the PoST workshops themselves, the concept of patterns for software testing has been presented at:

- STAR East test patterns presentation by Brian Marick
- Quality Week test results pattern language presentation by Keith Stobie
- PLoP workshopping of the test results pattern language
- A mini-workshop at STAR West by Brian Marick, et al.

Brian Marick is collecting the software test patterns in a Test Pattern repository at
http://testing.com/test-patterns/patterns/

PoST is actively seeking people interested in software testing patterns. You can submit patterns to Brian Marick for the web site and request participation at the next PoST workshop. Good Luck!

3 Appendix: Templates

A detailed, side-by-side comparison of pattern templates appears in [Binder99], along with a brief compare and contrast.

3.1 Design Patterns Template [Gamma+95]

Pattern Name (Scope, Purpose)	The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.
Intent	A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?
Also Known As	Other well-known names for the pattern, if any.
Motivation	A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.
Applicability	What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?
Structure	Class diagram
Participants	The classes and/or objects participating in the design pattern and their responsibilities. **Participant Name** Responsibility for what
Collaborations	How the participants collaborate to carry out their responsibilities.
Consequences	How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently? *A consequence bullet.* Description of consequence
Implementation	What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues? *An implementation Bullet.* Description of Bullet
Sample Code and Usage	Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk. Program Listing
Known Uses	Examples of the pattern found in real systems. Include at least two examples from different domains.

Related Patterns What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

3.2 Test Design Pattern Template[2] [Binder99]

Name

A word or phrase that identifies the pattern and suggests its general approach.

Intent

What test design problem does this pattern solve? What is the test strategy? This is a very brief synopsis.

Context

In what circumstances does this pattern apply? To what kind of software entities? At what scope(s)? This section corresponds to the first problem to be solved in test design: given some implementation, what is an effective test design and execution strategy? This section corresponds to the "motivation," "forces," and "applicability" subjects of design patterns.

Fault Model

Why does this pattern work? What kind of bugs does this pattern target? What kinds of bugs are guaranteed to be found if they exist? What kinds of bugs can hide? This section explains how the test model meets the necessary conditions for revealing the targeted faults. For a test case to reveal a fault, three things must happen.

- The fault must be **reached**. The test input and state of the system under test must cause the code segment in which the fault is located to be executed.
- The failure must be **triggered**. A fault does not always produce incorrect results when it is reached. The test input and state of the system under test must cause the code segment in which the fault is located to produce an incorrect result.
- The failure must be **propagated**. The incorrect result must become observable to the tester or an automated comparator.

This section must explain how the test suite will reach the targeted faults, how it can cause a failure to be triggered, and how a failure will be propagated to become observable.

Strategy

This section explains how the test suite is produced and implemented. There are four mandatory subsections.

- *Test Model* defines a representation for the responsibilities and/or implementation which are the focus of test design.
- *Test Procedure* defines an algorithm, technique, or heuristic by which test cases are produced from the model.

---

[2] Copyright © Robert V. Binder, 1999. Reproduced with permission.

- *Oracle* defines the algorithm, technique, or heuristic by which actual results to a test case are to be evaluated for pass/no pass.

- *Automation* discusses automated approaches to test suite generation, test run execution, and test run evaluation. The appropriate extent of manual testing is discussed. In many cases, this section suggests which test automation design patterns are applicable.

The strategy is typically presented by example. The concrete result of applying a test design pattern is an uninteresting, possibly cryptic collection of input/output vectors. Instead of a commentary on test vectors, this section contains a step-by-step application of the test model to suitable examples and the resulting test suite.

Entry Criteria

This section lists preconditions for effective and efficient testing with this pattern. An implementation should not be tested before it is, as a whole, **test-ready**. This has three beneficial effects. First, this reduces time lost because the implementation under test (IUT) is too buggy to test. Second, this reduces waste of testing resources on bugs that can be more easily revealed and removed by an antecedent process, typically testing at a smaller scope. Third, this reduces the number of bugs that will escape from a given scope of testing. As test scope increases, the extent to which individual components can be exercised decreases. For example, when testing at use case scope, we are typically do not attempt to achieve statement coverage on each component that supports the use case. Instead, we try to achieve coverage of all the inter-component message paths in the use case. We cannot hope to achieve individually adequate coverage when we are testing at higher scope. If lower scope testing is inadequate or skipped, it is likely that many bugs will not be revealed by higher scope testing, even if it is comprehensive. Therefore, meeting the entry criteria will improve both efficiency and effectiveness of testing.

Exit Criteria

What objective criteria must be met to achieve a complete test with this pattern? This section defines the extent to which the modeled responsibilities should be exercised and a corresponding coverage metric for an implementation at the scope (e.g., branch coverage at method scope.) or other conditions that indicate adequate testing has been achieved.

Consequences

What are the general prerequisites, costs, benefits, risks, and considerations for using this pattern?

Known Uses

What are the known uses of this test design pattern? What are the known uses of test models and strategies incorporated in this pattern? What are the efficiency and effectiveness of this pattern or similar strategies, as established by empirical studies?

Related Patterns

Are there any test design patterns that are similar or complementary? This section is omitted if there no related patterns.

4 Appendix: Patterns

I use ellipses (…) in the patterns from books below. My goal is not to provide the full pattern in all its gory detail, but to provide the flavor of using the pattern. By using the elipses I wish to avoid confusing the reader into thinking these extracts are the complete pattern. The full patterns have much richer details to make them applicable.

4.1 Window Place Pattern[3] [Alexander77]

The numbers in parentheses after a pattern refer to the page in *A Pattern Language: Towns, Buildings, Construction* having the pattern.

Window Place - Context

This pattern helps complete the arrangement of the windows given by **ENTRANCE ROOM** (130), **ZEN VIEW** (134), **LIGHT ON TWO SIDES OF EVERY ROOM** (159), **STREET WINDOWS** (164). According to the pattern, at least one of the windows in each room needs to be shaped in such a way as to increase its usefulness as a space.

Window Place

*** * ***

Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them.

Window Place – Conflict . . .

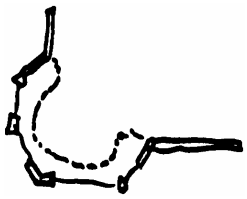
These kinds of windows which create "places" next to them are not simply luxuries; they are necessary. A room which does not have a place like this seldom allows you to feel fully comfortable or perfectly at ease. Indeed, a room without a window place may keep you in a state of perpetual unresolved conflict and tension—slight, perhaps, but definite.

Window Place - Forces

This conflict takes the following form. If the room contains no window which is a "place," a person in the room will be torn between **two forces:**

1. **He wants to sit down and be comfortable.**
2. **He is drawn toward the light.**

Obviously, if the comfortable places—those places in the room where you most want to sit—are away from the windows, there is no way of overcoming this conflict. You see, then, that our love for window "places" is not a luxury but an organic intuition, based on the *natural desire a person has to let the forces he experiences run free.* A room where you feel truly comfortable will always contain some kind of window place.



Window Place - Solutions

A bay window.
A window seat.

---

[3] Copyright © Christopher Alexander, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977 Reproduced with permission.

A low sill.
A glazed alcove.

And, of course, there are other possible versions too. In principle, any window with a reasonably pleasant view can be a window place, provided that it is taken seriously as a space, a volume, not merely treated as a hole in the wall. Any room that people use often should have a window place. Window places should even be considered for waiting rooms or as special places along the length of hallways.

Window Place - Summary

Therefore:
In every room where you spend any length of time during the day, make at least one window into a "window place."
Make it low and self-contained if there is room for that— **ALCOVES** (179); keep the sill low — **LOW SILL** (222); put in the exact positions of frames, and mullions, and seats after the window place is framed, according to the view outside — **BUILT-IN SEATS** (202), **NATURAL DOORS AND WINDOWS** (221). Set the window deep into the wall to soften light around the edges — **DEEP REVEALS** (223). Under a sloping roof, use **DORMER WINDOWS** (231) to make this pattern . . .

4.2 Proxy Pattern[4] [Gamma+95]

Intent

Provide a surrogate or placeholder for another object to control access to it.

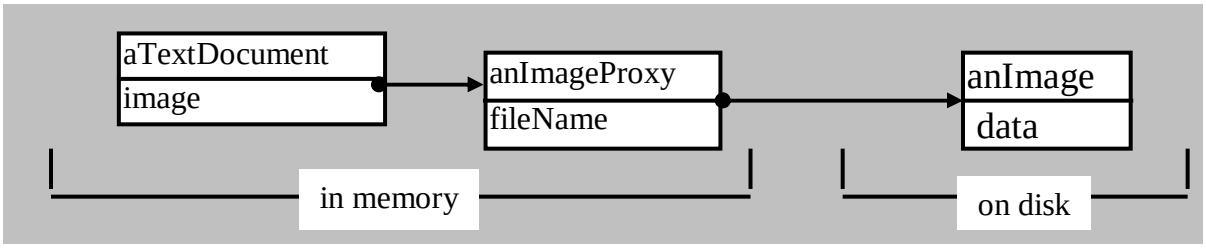
Also Known As

4.2.1.1.1 Surrogate

Motivation

. . . to defer the full cost of its creation and initialization until we actually need to use it.
. . . large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once
. . . But what do we put in the document in place of the image?

The solution is to use another object, an image proxy, that acts as a stand-in for the real image. The proxy acts just like the image and takes care of instantiating it when it's required.



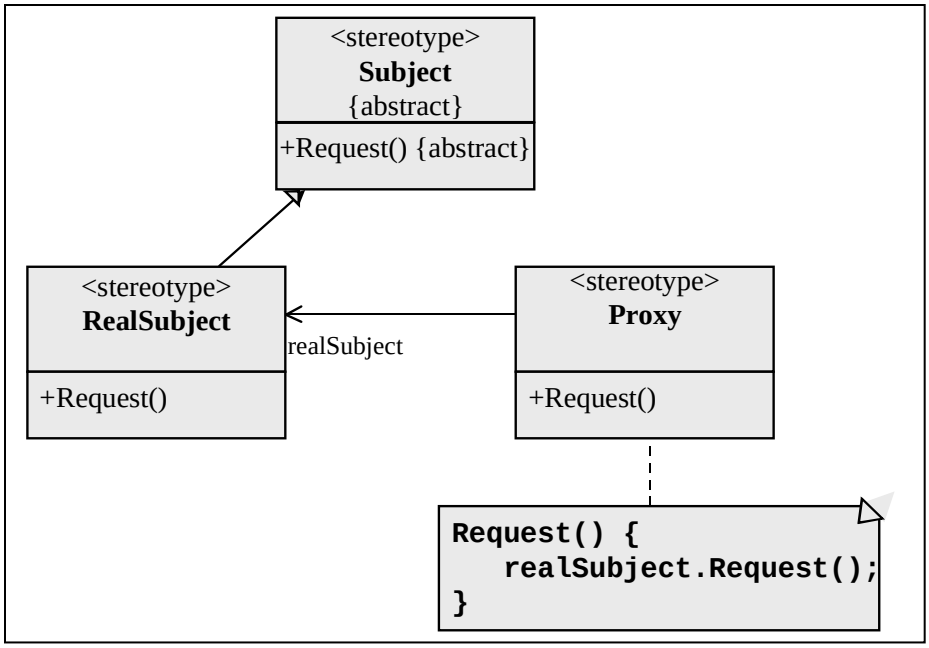
[4] Copyright © Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Reading Mass.: Addison-Wesley, 1995.. Reproduced with permission.

Applicability

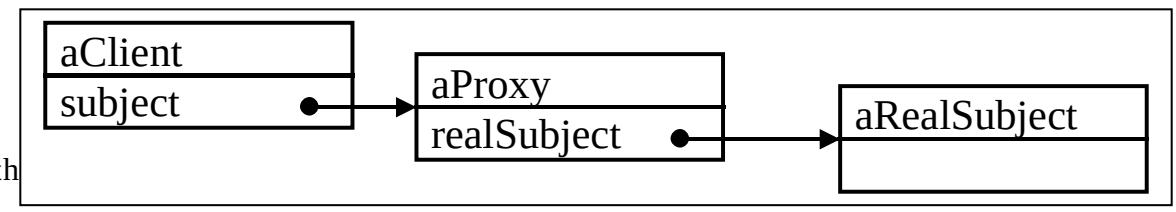
Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

1. A **remote proxy** provides a local representative for an object in a different address space. NEXTSTEP uses the class NXProxy for this purpose. Coplien calls this kind of proxy an "Ambassador."
2. A **virtual proxy** creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.
3. A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights. For example, KernelProxies in the Choices operating system provide protected access to operating system objects.
4. A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include
    - counting the number of references to the real object so that it can be freed automatically when there are no more references (also called smart pointers).
    - loading a persistent object into memory when it's first referenced.
    - checking that the real object is locked before it's accessed to ensure that no other object can change it.

Structure



Here's a possible object diagram of a proxy structure at run-time:



Keith

Participants

- Proxy (ImageProxy)
  - maintains a reference that lets the proxy access the real subject. . . .
  - provides an interface identical to Subject's . . .
  - controls access to the real subject . . .
  - other responsibilities depend on the kind of proxy:
    remote proxies , virtual proxies , protection proxies
- Subject (Graphic)
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- RealSubject (Image)
  - defines the real object that the proxy represents.

Collaborations

- Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

Consequences

The Proxy pattern introduces a level of indirection when accessing an object. . . .

1. A remote proxy can hide an object in a different address space.
2. A virtual proxy can perform optimizations . . .

Implementation

. . .

Sample Code

. . .

Known Uses

Virtual proxy example in Motivation section is from ET++ text building block classes.
NEXTSTEP uses proxies . . .
McCullough discusses using proxies in Smalltalk to access remote objects.
Pascoe describes how to provide side-effects on method calls and access control with "Encapsulators."

Related Patterns

Adapter (139): An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject. . . .
Decorator (175): Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.
Proxies vary in the degree to which they are implemented like a decorator. . . .

4.3 Category Partition Pattern[5] [Binder99]

[*Per the introduction in 4 Appendix: Patterns the material below is not the complete pattern. For example under Test Procedure below only the first sentence is provided which gives a good summary of the step. The book of course provides much more explicit guidance about each step which is useful in applying the pattern, but I don't think is necessary in understanding how the template is used. It is nice to know that the steps for a Stratgey are given. It is less important for this paper to fully understand the steps.*]

Intent
Design method scope test suites based on input/output analysis.

Context
How can we develop a test suite to exercise the functions implemented by a single method? A method may implement one or several functions, which each may present varying levels of cohesion. Even cohesive functions can have complex input/output relationships. A systematic technique to identify functions and exercise each input/output relationship is needed.
The Category-Partition pattern is appropriate for any method that implements one or more independent functions. For methods or functions that lack cohesion, the constituent responsibilities should be identified and modeled as separate functions. This approach is qualitative and may be worked by hand. It provides systematic implementation-independent coverage of method-scope responsibilities.
If the method selects one of many possible responses or has many constraints on parameter values, consider using the ***Combinational Function Test*** pattern.

Fault Model
The Category-Partition pattern assumes that faults are related to value combinations of message parameters and instance variables and these faults will result in missing or incorrect method output. Offutt and Irvine found that a Category-Partition approach was able to reveal 23 common C++ coding blunders [Offutt+95]. However, faults that are only manifested under certain sequences or which corrupt instance variables hidden by the method under test (MUT's) interface may not be revealed by category-partition tests.

Strategy
Test Model
Category-partition was first introduced as a general-purpose black box test design technique for software modules with parameterized functions [Ostrand+88].
Development of a method scope category-partition test suite is illustrated with C++ member function `List::getNextElement( )`. This member function returns successive elements of a `List` object. A position in a list of *m* elements is established at the *n*th element by other member functions. A call to `getNextElement( )` returns element *n* + 1. Successive calls to `getNextElement( )` return element *n* + 2 , *n* + 3, etc. If no position has been established by a previous operation or the previous position

[5] Copyright © Robert V. Binder, 1999. Reproduced with permission.

no longer exists due to an intervening change or delete, a `noPosition` exception is thrown. An `emptyList` exception is thrown if the list is empty.
Test Procedure
The test suite is produced in seven steps.
1. Identify the functions of the MUT. ...
2. Identify the input and output parameters of each function. ...
3. Identify categories *for each input parameter.* ...
4. Partition each category into choices. ...
Choices should include legal and illegal values ...
5. Identify constraints on choices. ...
6. Generate test cases by enumerating all choice combinations. ...
7. Develop expected results for each test case using an appropriate oracle. ...
Automation ...

Entry Criteria

- Small population of cases.

Exit Criteria

- Every combination of choices is tested once. ...
- If the method under test can throw exceptions for incorrect input or other anomalies, the test suite should force each exception at least once. ...
- Executing the test suite should exercise at least all branches in the MUT ...

Consequences
The Category-Partition pattern is general-purpose, non-quantitative, and straightforward. It does not require advanced analysis or automated support. However, identification of categories and choices is subjective[6]. Skilled and novice testers may identify different categories and choices. Individual blind spots may reduce effectiveness. Bugs that are only manifested under certain sequences of messages to other methods or which corrupt instance variables hidden by the MUT's interface may not be revealed.
The size of a Category-Partition test suite is ...
If the parameters have more than two interesting choices, we could easily generate thousands of test cases. This may be practically infeasible. The ***Invariant Boundaries*** pattern may be used to produce a smaller test suite whose size is a sum of the number of choices plus one, not the product (see below.)
With proper attention to interface details, a superclass Category-Partition method test suite may be reused to test an overriding subclass method. ...

Known Uses
Elements of the Category-Partition approach appear in nearly all black-box test design strategies -- for example, [Myers 79] [Marick 95] [Beizer 95]. An automated test tool for conventional code that supports the Category-Partition approach is presented in [Balcer+89]. An Z-based approach to automating the category-partition model is presented in [Laycock 92].

[6] Underlying is mine (Keith Stobie) not in the original by Robert Binder.

A study of manually developed Category-Partition test suites for a small C++ system found 55 out of 75 inserted faults. The inserted faults were selected to represent common C++ coding errors. Of 20 faults not found by the Category-Partition test suite, nineteen were memory leaks and one was masked by test tool initiation sequence [Offutt+95].

Cited Sources

[Balcer+89] Mark Balcer, William Halsing, and Thomas Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification.* New York: ACM Press. December 1989.
[Beizer 95] Boris Beizer. *Black box testing*. New York: John Wiley & Sons, 1995.
[Laycock 92] Gilbert Laycock. Formal specification and testing: a case study. *Journal of Software Testing, Verification, and Reliability* 2(1):7-23, May 1992.
[Marick 95] Brian Marick. *The craft of software testing.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1995.
[Myers 79] Glenford J. Myers. *The art of software testing.* New York: John Wiley & Sons, 1979.
[Offutt+95] A. Jefferson Offutt and Alisa Irvine. Testing object-oriented software using the category-partition method. In *Proceedings, TOOLS 17.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc. 1995.
[Ostrand+88] Thomas J. Ostrand and Marc J. Blacer. The category-partition method for specifying and generating functional tests. *Communications of the ACM* 31(6):676-686, June 1988.

4.4 Architecture Achilles Heel Analysis Pattern[7]

[Hendrickson+01]

Name: Architecture Achilles Heel Analysis

Elisabeth Hendrickson, Grant Larsen

Objective: Identify areas for bug hunting – relative to the architecture

Problem: How do you use the architecture to identify areas for bug hunting?

Context: You are a tester and want to make best use of the testing time. Your goal is to find high severity bugs. You have a physical, functional, or dynamic map of the system in some form (or can create one with *Architecture Reverse Engineerin*g). You know what characteristics of the system are most important *(e.g.* is reliability more important than performance?). You may or may not be doing formal test planning.

Forces:

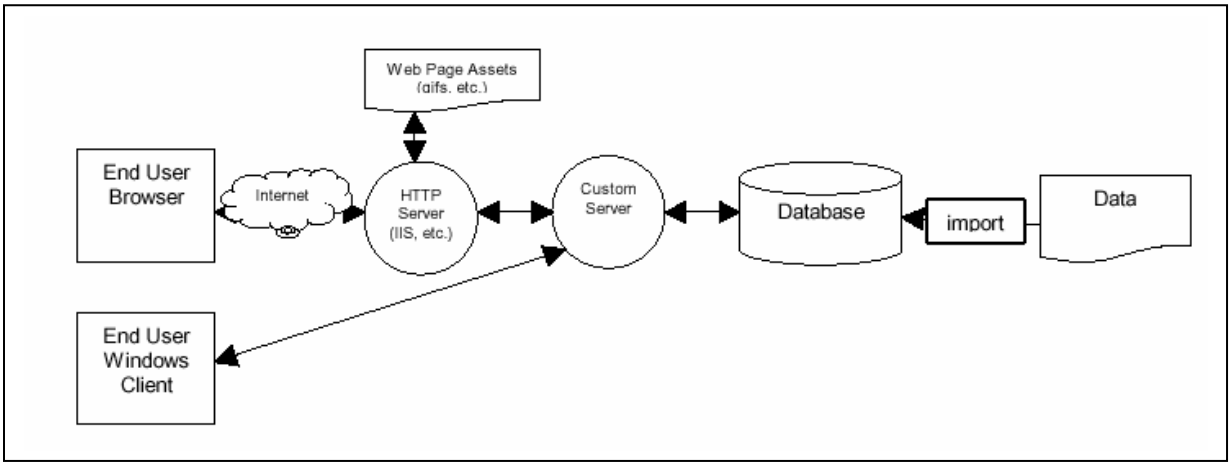
- You may not be able to answer all your own questions about the architecture.
- Getting information about the architecture or expected results may require additional effort.
- The goals of the architecture may not have been articulated or may be unknown.

Solution:

An architectural diagram is usually a collection of elements of the system, including

[7] Used with permission of Elisabeth Hendrickson.

executables, data, etc., and connections between those elements. The architecture may specify the connections between the elements in greater or lesser detail. Here's an example deployment architecture diagram. This technique works for deployment, physical, functional, or dynamic architecture maps.



In this example, we have two user interfaces, a Web-based interface and a Windows client. The Web based interface connects over the Internet to an HTTP server that in turn connects to the custom server process that reads and writes data to the database. The Web version relies on web site assets (gifs, etc.). The Windows client connects directly to this custom server. Data from the development environment is imported into the database with a command line import utility.
In analyzing the architecture, walk through each item and connection in the diagram. Consider tests and questions for each of the items in the diagram.

Connections

- Disconnect the network.
- Change the configuration of the connection (for example, insert a firewall with maximum paranoid settings or put the two connected elements in different NT domains).
- Slow the connection down (28.8 modem over Internet or VPN).
- Speed the connection up.
- Does it matter if the connection breaks for a short duration or a long duration? (In other words, are there timeout weaknesses?)

User Interface

- Input extreme amounts of data to overrun the buffer.
- Insert illegal characters.
- Use data with international and special characters.
- Use reserved words.
- Enter zero length strings.
- Enter 0, negatives, and huge values wherever numbers are accepted (watch for boundaries at 2^{15}=32768, 2^{16}=65536, and 2^{31}=2147483648).
- Make the system output huge amounts of data, special characters, and reserved words.
- Make the result of a calculation 0, negative, or huge.

Server Processes

- Stop the process while it's in the middle of reading or writing data (emulates server down).
- Consider what happens to the clients waiting on the data.
- What happens when the server process comes back up? Are transactions rolled back?

Databases

- Make the transaction log absurdly small and fill it up.
- Bring down the database server.
- Insert tons of data so every table has huge numbers of rows.
- In each case, watch how it affects functionality and performance on the server and clients.

Multiple Users and Interfaces

- Two users doing the same thing at the same time—both trying to save changes to the same record at the same time, for example.
- Two users doing the same thing at the same time in different interfaces (Web and Windows).

Data Files

- Write a file to a full disk full.
- Write a file to an area with no write permissions.
- Use UNC path names.
- Use path names with spaces.
- Use long path names (not 8.3).
- File exists if it shouldn't or doesn't exist if it should.
- Read/write to a file locked by another process.
- Read a file with no read permissions.
- Read a file in the wrong format.
- Read in a huge file.
- Read a 0 length file.
- Read a file with line feed v. carriage return chars at end of line (often a UNIX v. NT problem).

Big System Questions

Walk through the architectural diagram asking questions. A list of sample questions follows. Depending on your architecture, you may have additional or different questions.

- For each executable in the architecture, what happens if that executable is not running?
- For each data storage mechanism (whether a relational database, file in the file system, or other persistent mechanism), what if the data is corrupted? Deleted? Lacks integrity or conflicts with other data in the system?
- For each connection, what if the connection breaks? Does it matter if the connection breaks for a short duration or a long duration? (In other words, are there timeout weaknesses?)
- Are there multiple connections going into or out of an element? Does this open up the possibility of resource conflicts or deadlocks?
- Are there shared resources where there may be performance bottlenecks?

- Do the elements of the architecture have states? Is there any possibility that those states might conflict—for example, could one part of the system try to start up a process while another part is trying to shut it down?

As you answer the questions mentally or verbally, write down possible risks, weaknesses, or tests that occur to you. These become areas of the system on which to focus your energy. Alternatively, you could use the architectural diagram while exploring the system hands-on.

Where you are unable to answer your own questions, seek the assistance of others who can help you: programmers, architect(s), other testers, technical support personnel, etc.

Rationale:

Understanding the big picture of the system under test can help testers focus their energy appropriately. Without this understanding it's very easy to spend large amounts of time investigating relatively minor bugs or risks.

Resulting Context:

You now know more about the architecture and implications of various design decisions and can use that information to guide your test effort. If the architectural goals are still unclear, perhaps an *Explicit Architectural Goal Articulation* is in order.

Additional Benefits:

- Increases communication between different groups within the project.
- Finds potential architectural flaws earlier.
- Aligns architecture with its goals.
- Provides additional fodder for test planning for current and future releases.

Liabilities:

- You may need to spend a large amount of time understanding the system before you can productively seek out bugs.
- Other members of the team or organization may need to invest more time answering questions to contribute to the test effort than anticipated.

Note that the last two liabilities may actually be good for the project or software in the long run.

- Some members of the team may feel threatened during this process.
- Information uncovered during this process may throw the project into chaos, at least temporarily.

Related Patterns:

- Architecture Reverse Engineering
- Explicit Architectural Goal Articulation

4.5 Check as you Go Pattern [Stobie01]

Aliases: In-Line check

Context

You have pre-specification of the test results.

Data from the environment is dynamically needed to evaluate correctness.[8]

[8] For example, you want to verify the timestamp on a log record. You can print the time before and after you expect the log record, but now your batch comparison requires

Either the checks are very cheap to make or the test is not highly performance sensitive, that is, you can afford to spend time to do the checks during the test.
Correctness requires specific relationships to occur, for example complex data structures like trees, or receiving an asynchronous event within a specific time period.

Problem

Do you compare actual results to expected results at each step as you go, or all at the end?

Forces

- Future results may be invalid if early results don't pass.
- Special code may need to be run to gather diagnostic info depending on the failure.
- Constantly interspersing checking code in testware can make the logic of a test case unclear.

Solution

Check each result in-line as finely as possible immediately after its inputs are submitted. You may need to incorporate current specific data to have the results compare.
If a validation fails then log the failure and optionally don't proceed forward with the rest of the test. For example, when bounding the time of the result, if a connection isn't made within a timeout period, abort the test rather than waiting to get more output. This concept applies both to testing post-conditions within an individual test case and execution of test cases within an automated test suite. A major feature of *Check as you Go* is to only log the software under test as passing after all relevant post-conditions have been checked. Typical GUI Testing using *Check as you Go* might look like:

```
Produce Screen1
        Verify Screen 1
Produce Screen2
        Verify Screen 2
  . . .
```

Indications

- The desired results of a test input are precisely known ahead of time.
- The test is programmable, that is, the result of each set of inputs is easily checked.
- The test is long running, and could be meaningless if there is an early discrepancy for one of the post-conditions.

Rationale

It generally aids comprehensibility of the tests if the expected results appear in the same file and as close to the inputs as possible.

Resulting Context/Consequences

Complete list of post-conditions being checked may be spread throughout the test code.

Related Patterns

See 4.6 Batch check Pattern [Stobie01] for the same problem, but different context.

---

relative checks (less than and greater than) instead of just equals. This is usually a significantly more difficult comparison algorithm.

Examples/Known Uses

Frequently used for API/Class Drivers approach.

Junit – See http://www.junit.org/
Expect tool – See http://expect.nist.gov/
POSIX Verification Test Suite – See http://www.opengroup.org/testing/downloads/vsx-pcts-faq.html

Code Samples

Note below that the result is checked as you go in the code and not by some external entity.

Java/C++

```
result = squareRoot(1);
if (result != 1) {
      LogFail( “squareRoot(1) resulted in “+result
              +” where 1 was expected” )
      }
      else LogPass( “squareRoot(1)” );
result = squareRoot(4);
if (result != 2) {
      LogFail( “squareRoot(4) resulted in “+result
              +” where 2 was expected” )
      }
      else LogPass( “squareRoot(4)” );
```

Shell

```
result=`squareRoot 1`
if [ “$result” != “1” ] ; then
      logFail “squareRoot 1 resulted in $result, where 1 was
expected.”
else
      logPass “squareRoot 1”
fi
result=`squareRoot 4`
if [ “$result” != “2” ] ; then
      echo “squareRoot 4 resulted in $result, where 2 was
expected.”
else
      logPass “squareRoot 4”
fi
```

4.6 Batch check Pattern [Stobie01]

Aliases: Benchmark, baseline, golden results, canonical results, gold master - where a benchmark file containing expected results is used.

Context

You have pre-specification of the test results or you can manually judge the output for correctness when exact expected results are not necessarily known. The post-conditions are independent, that is don't use when you need to check one post-condition before choosing to check another one. The set of inputs is not easily separable (for example compiler input file). The output can be compared easily with minimal filtering. The checks are expensive to make or the test is highly performance sensitive and it is relatively cheap to just record the results. Don't need to check result of each step to determine if future steps of the test are valid.

Problem

Do you compare results at each step as you go or all at the end?

Forces

- Future results may be invalid if early results don't pass.
- Special code may need to be run to gather diagnostic info depending on the failure.
- Output data may have specifics (e.g. node name or time) the test doesn't care about.
- Constantly interspersing checking code in testware can make the logic of a test case unclear.
- Avoiding false positives is more important than avoiding false negatives

Solution

Provide a benchmark file of expected results. Collect actual results as the test executes. At the end compare the expected and actual results. Use filtering programs to ignore specifics the test doesn't care about.

Indications

- A failure near the start of the test doesn't invalidate the results that follow.

Rationale

Tests are very easy to develop. Expected results can be generated by the program once, hand-checked for accuracy once, and then reused again and again. Expected results can be updated without affecting any code (since they are in a separate file). Batch processing may be the nature of Item Under Test (IUT).

Resulting Context/Consequences

One dangerous Consequence frequently seen is testers get lazy when the expected output has to change and don't scrutinize the initial results carefully enough for correctness. In this case, the incorrect actual output gets canonized as the expected output [Bach99]. Batch check can make maintenance more difficult *if* the relationship between inputs and outputs is not very clear.

Frequently special filtering patterns (regular expressions) are needed to ignore uncontrollable extraneous differences, for example machine names, time stamps, etc. This filtering has a small risk of missing incidental problems, such as the time being reported incorrectly. Generally you rely on other tests to specifically verify what most of these types of tests ignore.

Related Patterns

See 4.5 Check as you Go Pattern [Stobie01] with an alternate solution due to different context.

Examples/Known Uses

Compiler testing or any transformation type program. It is generally too expensive to test each feature completely individually, and a great deal of common setup exists to test any one feature.

An example of expensive checks goes like this:
The author was testing a new transactional capability of a database. The transactions allowed the commit or abort of a set of operations (insert, update, delete). The original test was written to update the database and keep a separate file of the expected results. The extra logging of expected results allowed enough time that many race conditions in the transaction system were missed when several transactions were supposed to be occurring in parallel. The Batch check solution was to rely on the concept of database integrity via a constraint. A particular column had the constraint that it was always divisible by 10,000. Transactions could do a series of operations such that the constraint would continue to hold if all or none of the operations occurred. For example, insert record A with column value of 5,000 and update record B's column value by adding 5,000. This allowed large numbers of concurrent transactions to be run at once. The batch check was to verify the integrity of the column constraint at the end of a set of transactions.

Code Samples

The abbreviated example below shows the expected output stored in a separate file and then a batch comparison done.

Shell:

```
cat <<EOINPUT |expectedOutput
input output
1 1
4 2
EOINPUT

# Set up for read from fd=4 with above data
exec 4<expectedOutput

# Read (heading) line from expectedOutput file into input_value &
output value
read -u4 input_value?"headings " output_value
# Put heading line in actualOutput file
echo $input_value $output_value | actualOutput
# While lines to read, put value into input_value & output_value
while read -u4 input_value?"input and output" output_value; do
   # Echo input to actualOutput (without a linefeed/newline)
   echo "$input_value \c"  actualOutput
   # Put actual result on end of previous line
   squareRoot $input_value  actualOutput
done

diff expectedOutput actualOutput
```

5 Appendix: Pattern History

From [Marick01]:

> The current use of the term "pattern" is derived from the writings of the architect Christopher Alexander who has written several books on the topic as it relates to urban planning and building architecture:
>
> •*Notes on the Synthesis of Form*, Harvard University Press, 1964 [Notes]
>
> •*The Oregon Experiment*, Oxford University Press, 1975 [Oregon]
>
> •*A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977 [Alexander77]
>
> •*The Timeless Way of Building*, Oxford University Press, 1979 [Alexander79]

From: **Brad Appleton** *http://www.enteract.com/~bradapp/* :

> In 1987, Ward Cunningham and Kent Beck were working with Smalltalk and designing user interfaces. They decided to use some of Alexander's ideas to develop a small, five pattern, language for guiding novice Smalltalk programmers. They wrote up the results and presented them at OOPSLA'87 in Orlando in the paper *"Using Pattern Languages for Object-Oriented Programs"*.
>
> Soon afterward, Jim Coplien (more affectionately referred to as "Cope") began compiling a catalog of C++ idioms (which are one kind of pattern) and later published them as a book in 1991, *Advanced C++ Programming Styles and Idioms*.
>
> From 1990 to 1992, various members of the Gang of Four had met one another and had done some work compiling a catalog of patterns. Discussions of patterns abounded at OOPSLA'91 at a workshop given by Bruce Andersen (which was repeated in 1992). Many pattern notables participated in these workshops, including Jim Coplien, Doug Lea, Desmond D'Souza, Norm Kerth, Wolfgang Pree, and others.
>
> In August 1993, Kent Beck and Grady Booch sponsored a mountain retreat in Colorado, the first meeting of what is now known as the Hillside Group. Another patterns workshop was held at OOPSLA'93 and then in April of 1994, the Hillside Group met again (this time with Richard Gabriel added to the fold) to plan the first PLoP conference.
>
> Shortly thereafter, the *Design Patterns* [Gamma+95] book was published, and the rest is history.

6 Appendix: Pattern Introductions

Brad Appleton's *Patterns in a Nutshell* is an excellent introduction to design patterns.
http://www.enteract.com/~bradapp/docs/patterns-nutshell.html

Brad Appleton's "*Patterns and Software: Essential Concepts and Terminology*" [Appleton00] is rather long, but nicely set up for skimming.

Doug Lea's Patterns-Discussion FAQ:
http://www.enteract.com/~bradapp/docs/patterns-intro.html
http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html

Jim Coplien has put the text of his now-out-of-print booklet, "*Software Patterns Management Briefing*", online (copyright 2000 Lucent Technologies). It's PDF:
http://www1.bell-labs.com/user/cope/Patterns/WhitePaper/SoftwarePatterns.pdf.

The Software Patterns Management Briefing, published in 1996, was an early articulation of the principles, values, and practices behind the pattern discipline. Still timeless today, it is organized around a body of dozens of example patterns. It covers a wide range of topics ranging from pattern forms, to pattern languages, to the history of software patterns, and pattern ethics.
It captured the spirit of the early and emerging directions of software patterns.
The briefing was written for an inexpert audience. No prior knowledge, either of patterns or of software, is necessary to appreciate the principles highlighted in this briefing. I find it to be an excellent first-time introduction to patterns for the curious, and a good overview that helps provide context even for practitioners who have been using patterns for some time.

6.1 Pattern Writing and Workshops

How to Hold a Writer's Workshop
http://www.cs.wustl.edu/~schmidt/writersworkshop.html

A Pattern Language for Writer's Workshops by James Coplien.
http://www1.bell-labs.com/user/cope/Patterns/WritersWorkshops/

"*A Pattern Language for Pattern Writing*", by Meszaros and Doble. [Meszaros+]

"*Seven Habits of Successful Pattern Writers*", by John Vlissades

"*Tips for Writing Pattern Languages*", by Ward Cunningham.

7 References

[Alexander77] Christopher Alexander, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977

[Alexander79] Christopher Alexander. *The Timeless Way of Building.* New York: Oxford University Press, 1979.

[Appleton00] Brad Appleton, *Patterns and Software: Essential Concepts and Terminology*, web 2000, http://www.enteract.com/~bradapp/docs/patterns-intro.html

[Bach99] James Bach "*Test Automation Snake Oil*", copyright 1999 http://www.satisfice.com/articles/test_automation_snake_oil.pdf

[Binder99] Robert V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools* (http://www.rbsc.com/TOOSMPT.htm), The Addison-Wesley Object Technology Series c. 1999

[Coplien+95] James Coplien and Douglas Schmidt (eds.) *Pattern languages of program design.* Reading, Mass.: Addison-Wesley, 1995.

[DeLano+00] David DeLano and Linda Rising, "*Patterns for System Testing*", AG Communications, web 2000 http://www.agcs.com/patterns/papers/systestp.htm

[Firesmith96] Donald G. Firesmith. *Pattern language for testing object-oriented software* in *Object Magazine* 5(9):32-28, January 1996

[Gamma+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Reading Mass.: Addison-Wesley, 1995.

[Hendrickson+01] Elisabeth Hendrickson and Grant Larsen, *Architecture Achilles Heel Analysis*, web 2001 http://www.testing.com/test-patterns/patterns/Architecture-Achilles-Heels-Analysis.pdf

[Levine+] David L. Levine and Douglas C. Schmidt, *CS 342: Patterns and Frameworks* Introduction, web: http://www.cs.wustl.edu/~schmidt/PDF/patterns-intro4.pdf

[Marick01] Brian Marick, *Applying Patterns to Software Testing*, web 2001 http://www.testing.com/test-patterns/index.html

[Meszaros+] Meszaros and Doble, *A Pattern Language for Pattern Writing*, http://www.hillside.net/patterns/Writing/patterns.html

[McGregor96] John McGregor & Anuradha Kare, *Parallel Architecture for Component Testing of Object-Oriented Software,* Proceedings of the Ninth International Quality Week, May 96.

[PHP01] *Patterns Home Page* web 2001 http://hillside.net/patterns/

[Stobie01] Keith Stobie, *Automating Test Oracles and Decomposability,* Proceedings of the Fourteenth International Quality Week, May 2001

Patterns Of Software Test (PoST) – an Overview

Based on the PoST workshops

Keith Stobie

Microsoft

kstobie@acm.org

(work originally started while sponsored by BEA systems.)

What are patterns?

Patterns are a way of helping designers.

” each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.”

- **vocabulary** for problem-solvers
- focus attention on the **forces** that make up a problem.
- encourage **iterative** thinking.

History of Patterns

1964-79 formalized by the architect Christopher Alexander.

1987, Ward Cunningham and Kent Beck presented at OOPSLA'87 the paper *"Using Pattern Languages for Object-Oriented Programs"*.
use some of Alexander's ideas for a small five pattern language for guiding novice Smalltalk programmers.

1991, Jim Coplien compiles a catalog of C++ *idioms* published as **Advanced C++ Programming Styles and Idioms**.

1991 Discussions of patterns abounded at OOPSLA'91 at a workshop.

1993, Kent Beck & Grady Booch sponsor a mountain retreat in Colorado, 1st meeting of the Hillside Group & patterns workshop at OOPSLA'93

1994, **Hillside Group** met again to plan the first PLoP conference.

1995, the *[GoF]* **Design Patterns** book was published

. . .

2001, PoST workshops

Architecture - Alexander

The term "pattern" is derived from the writings of the architect Christopher Alexander as it relates to urban planning and building architecture.

```
IF    you find yourself in CONTEXT
          for example EXAMPLES,
          with PROBLEM,
          entailing FORCES
      THEN for some REASONS,
          apply DESIGN FORM AND/OR RULE
          to construct SOLUTION
          leading to NEW CONTEXT & OTHER PATTERNS
```

Window Place - Context

This pattern helps complete the arrangement of the windows given by **ENTRANCE ROOM (130), ZEN VIEW (134), LIGHT ON TWO SIDES OF EVERY ROOM** (159), **STREET WINDOWS** *(164).*

According to the pattern, at least one of the windows in each room needs to be shaped in such a way as to increase its usefulness as a space.

Window Place

* * *

Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them.

Window Place - Conflict

. . .

These kinds of windows which create “places” next to them are not simply luxuries; they are *necessary.* A room which does not have a place like this seldom allows you to feel fully comfortable or perfectly at ease. Indeed, a room without a window place may keep you in a state of perpetual unresolved conflict and tension—slight, perhaps, but definite.

Window Place - Forces

This conflict takes the following form. If the room contains no window which is a “place,” a person in the room will be torn between **two forces:**

1. **He wants to sit down and be comfortable.**
2. **He is drawn toward the light.**

Obviously, if the comfortable places—those places in the room where you most want to sit—are away from the windows, there is no way of overcoming this conflict. You see, then, that our love for window “places” is not a luxury but an organic intuition, based on the ***natural desire a person has to let the forces he experiences run free***. A room where you feel truly comfortable will always contain some kind of window place.

Window Place - Solutions

A bay window.

A window seat.

A low sill.

A glazed alcove.

And, of course, there are other possible versions too. In principle, any window with a reasonably pleasant view can be a window place, provided that it is taken seriously as a space, a volume, not merely treated as a hole in the wall. Any room that people use often should have a window place. And window places should even be considered for waiting rooms or as special places along the length of hallways.

Window Place - Summary

Therefore:

In every room where you spend any length of time during the day, make at least one window into a “window place.”

Make it low and self-contained if there is room for that— **ALCOVES (179)**; keep the sill low — **LOW SILL (222);** put in the exact positions of frames, and mullions, and seats after the window place is framed, according to the view outside — **BUILT-IN SEATS (202), NATURAL DOORS AND WINDOWS (221).** And set the window deep into the wall to soften light around the edges — **DEEP REVEALS (223).** Under a sloping roof, use **DORMER WINDOWS (231)** to make this pattern. .

Software Patterns

Patterns can apply to more than just software design.
They are being used for processes, etc.

- *A Development Process Generative Pattern Language*, James Coplien
  http://www1.bell-labs.com/user/cope/Patterns/Process/index.html
- *A Risk Management Catalog*, Alistair Cockburn
  http://members.aol.com/acockburn/riskcata/risktoc.htm
- *CHECKS Pattern Language of Information Integrity*, Ward Cunningham
  http://c2.com/ppr/checks.html

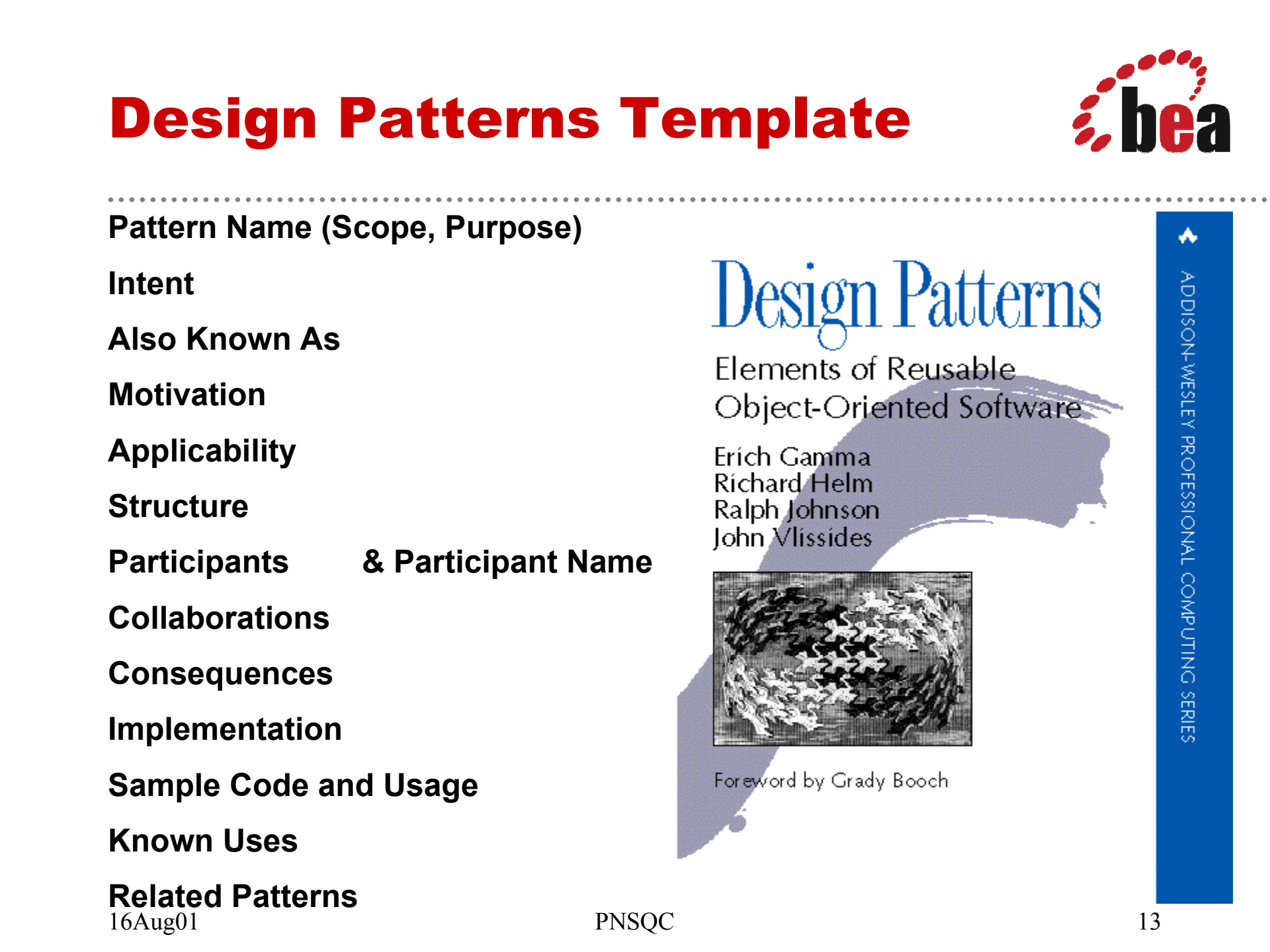
Software Design Patterns

Design Patterns: elements of reusable object-oriented software.
[GoF] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

"descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context"

- Design patterns represent solutions to problems that arise when developing software within a particular context
  - "Patterns == problem/ solution pairs in a context"
- Patterns capture the static and dynamic structure and collaboration among key participants in software designs
  - They are particularly useful for articulating how and why to resolve non-functional forces
- Patterns facilitate reuse of successful software architectures and designs

Design Patterns Template



Pattern Name (Scope, Purpose)

Intent

Also Known As

Motivation

Applicability

Structure

Participants **& Participant Name**

Collaborations

Consequences

Implementation

Sample Code and Usage

Known Uses

Related Patterns

Proxy Pattern

Intent

Provide a surrogate or placeholder for another object to control access to it.

Also Known As

Surrogate

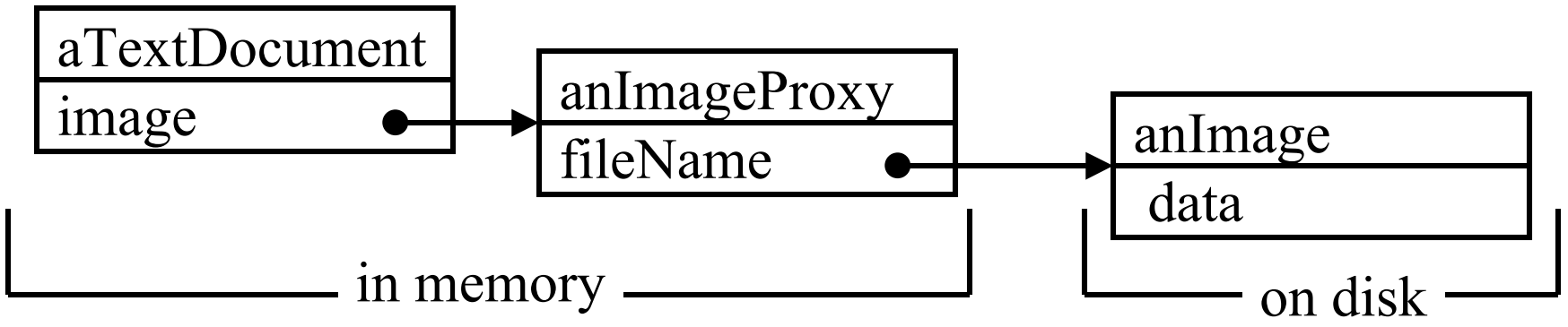
Motivation

. . . to defer the full cost of its creation and initialization until we actually need to use it. . . . large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once

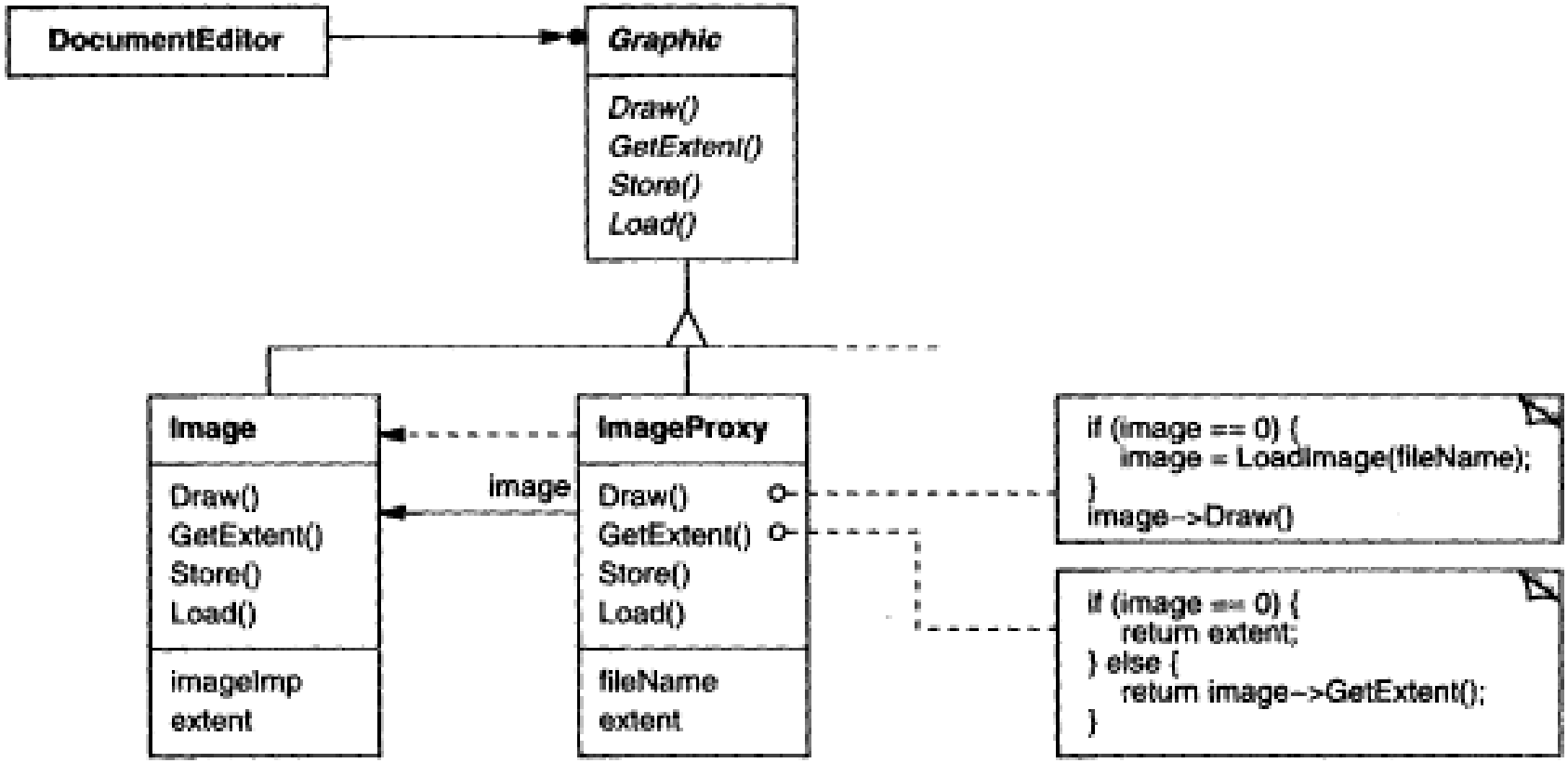
. . . But what do we put in the document in place of the image?

Proxy - Motivation

The solution is to use another object, an image **proxy**, that acts as a stand-in for the real image. The proxy acts just like the image and takes care of instantiating it when it's required.



Proxy - Example

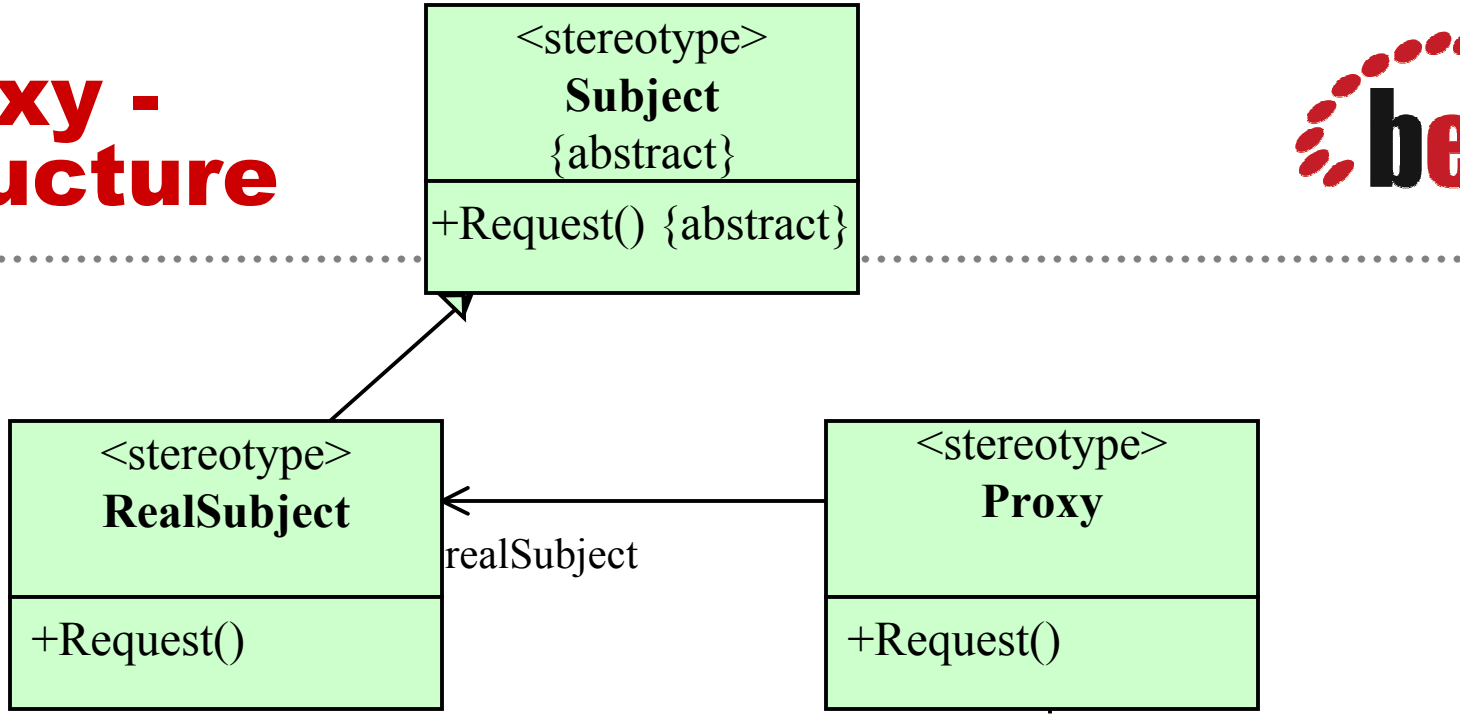


Proxy - Applicability

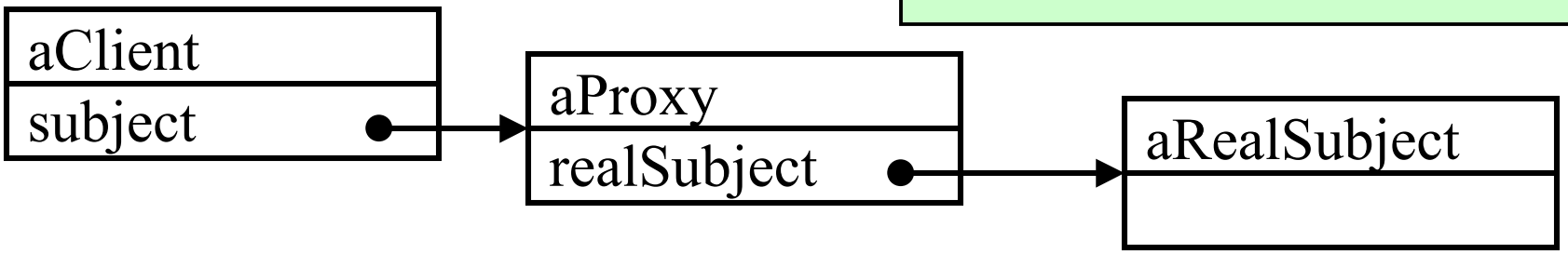
Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

1. A **remote proxy . . .**
   Coplien [Cop92] calls this kind of proxy an "Ambassador."
2. A **virtual proxy** creates expensive objects on demand. **. . .**
3. A **protection proxy** controls access to the original object. **. . .**
4. A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed.

Proxy - Structure



Here's a possible object diagram of a proxy structure at run-time:



Proxy - Participants

- Proxy (ImageProxy)
  - maintains a reference that lets the proxy access the real subject. . . .
  - provides an interface identical to Subject's . . .
  - controls access to the real subject . . .
  - other responsibilities depend on the kind of proxy:
    - *remote proxies* , *virtual proxies* , *protection proxies*
- Subject (Graphic)
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- RealSubject (Image)
  - defines the real object that the proxy represents.

Proxy - Collaborations & Consequences

Collaborations

- Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

Consequences

The Proxy pattern introduces a level of indirection when accessing an object. . . .

1. A remote proxy can hide an object in a different address space.
2. A virtual proxy can perform optimizations . . .

Implementation

. . .

Sample Code

Proxy - Uses & Patterns

Known Uses

- Virtual proxy example in Motivation section is from ET++
- NEXTSTEP [Add94] uses proxies . . .
- McCullough [McC87] discusses using proxies in Smalltalk to access remote objects. Pascoe [Pas86] describes how to provide side-effects on method calls & access control with “Encapsulators.”

Related Patterns

- Adapter (139): An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject. . . .
- Decorator (175): Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.
- Proxies vary in the degree to which they are implemented like a decorator. . . .

Test Patterns?

System Test Pattern Language
David DeLano and Linda Rising

Testing Object-Oriented Systems: Models, Patterns, and Tools
Robert Binder

Pattern Language for Testing Object-oriented Software,
Donald Firesmith, Object Magazine, January 96.

Parallel Architecture for Component Testing of Object-Oriented Software,
John McGregor & Anuradha Kare, Proceedings of the Ninth International Quality Week, May 96.

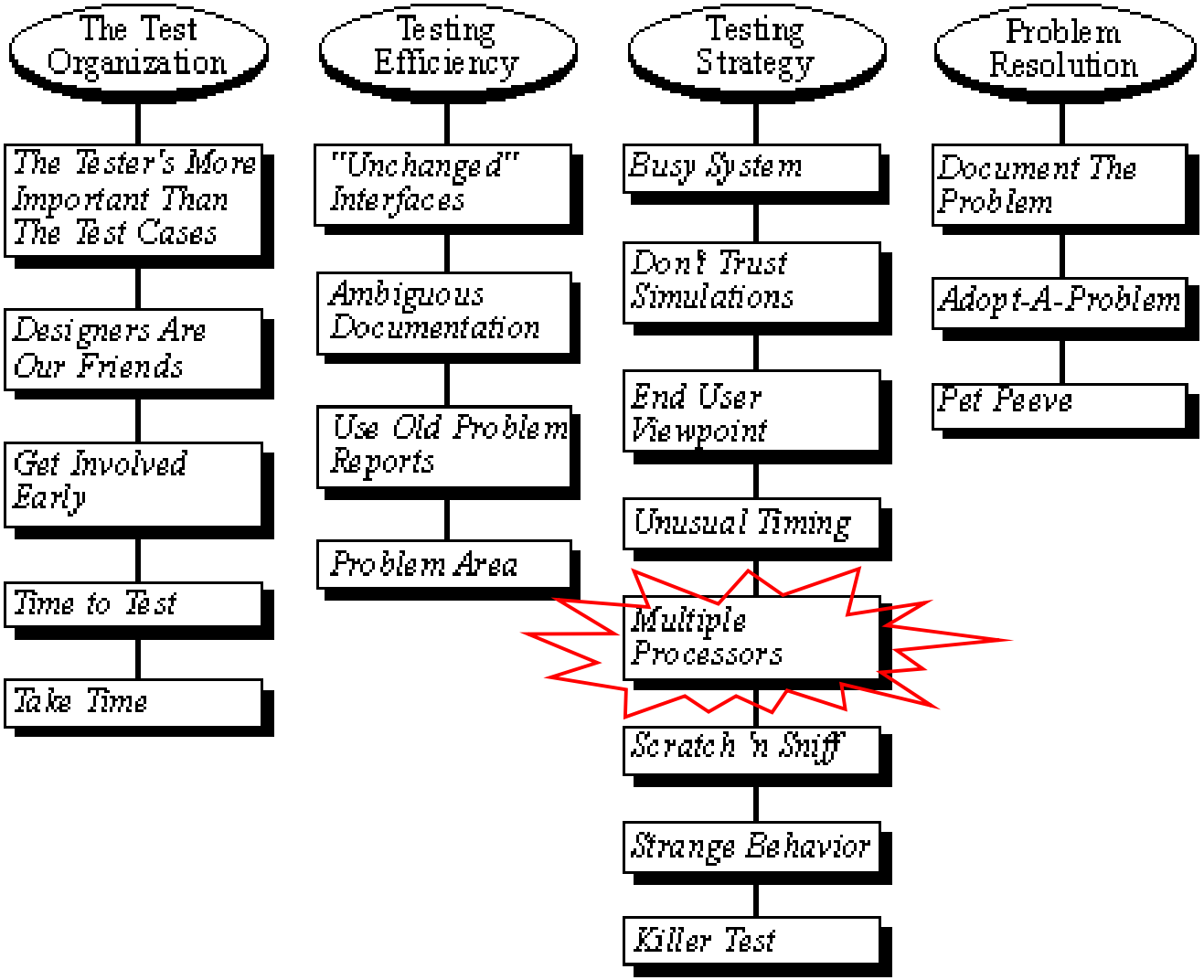
System Test Pattern Language

"System Test Pattern Language",
David DeLano and Linda Rising,
AG Communications

http://www.agcs.com/patterns/papers/systestp.htm

- To help System Testers evaluate the readiness of a product for shipping to the customer
- Many of the patterns are orthogonal to all testing phases
- This is not a complete pattern language

System Test Overview



Pattern: *Multiple Processors*

Problem What strategy should be followed when System Testing a system comprised of multiple processors?

Solution Test across multiple processors.

Resulting Context Problems that occur in one processor will probably occur in other processors. Tests that pass on one processor may fail on another.

Rationale When a problem is found in one processor, that feature will usually have problems running on other processors. A dirty feature is a dirty feature.
Features that run on multiple processors aren't always designed to run on all processors.

Object Oriented Test Patterns

Testing Object-Oriented Systems: Models, Patterns, and Tools introduces

37 test design patterns

17 design patterns for test automation

16 micro-patterns for test oracles

new pattern template focuses explicitly on key dimensions of test design:

- When is a particular test strategy appropriate?
- What kind of bugs will it find?

Binder's Test Design Pattern Template

Name

Intent

Context

Fault Model : **reached. triggered. propagated.**

Strategy : *Test Model, Test Procedure, Oracle, Automation*.

Entry Criteria : **test-ready**

Exit Criteria

Consequences

Known Uses

Related Patterns

Category Partition - Context

Intent

Design method scope test suites based on input/output analysis.

Context

How can we develop a test suite to exercise the functions implemented by a single method? . . . A systematic technique to identify functions and exercise each input/output relationship is needed.

The Category-Partition pattern is appropriate for any method that implements one or more independent functions. . . . This approach is qualitative and may be worked by hand. It provides systematic implementation-independent coverage of method-scope responsibilities.

If the method selects one of many possible responses or has many constraints on parameter values, consider using the ***Combinational Function Test*** pattern.

Category Partition - Fault Model

Fault Model

The Category-Partition pattern assumes that faults are related to value combinations of message parameters and instance variables and these faults will result in missing or incorrect method output. Offutt and Irvine found that a Category-Partition approach was able to reveal 23 common C++ coding blunders [Offutt+95]. However, faults that are only manifested under certain sequences or which corrupt instance variables hidden by the MUT's interface may not be revealed by category-partition tests.

Category Partition - Test Model

Test Model

Category-partition was first introduced as a general-purpose black-box test design technique for software modules with parameterized functions [Ostrand+88]. Development of a method scope category-partition test suite is illustrated with C++ member function `List::getNextElement( ). This member function ...`

Category Partition - Strategy

Test Procedure

The test suite is produced in seven steps.

1. *Identify the functions of the MUT.*.
2. *Identify the input and output parameters of each function.*
3. *Identify* categories *for each input parameter.*
4. *Partition each category into choices.*
   Choices should include legal and illegal values as shown ...
5. *Identify constraints on choices*.
6. *Generate test cases by enumerating all choice combinations.*
7. *Develop expected results for each test case using an appropriate oracle.*
   - Automation

Category Partition - Criteria

Entry Criteria: Small pop.

Exit Criteria

- Every combination of choices is tested once.
- If the method under test can throw exceptions for incorrect input or other anomalies, the test suite should force each exception at least once.
- Executing the test suite should exercise at least all branches in the method under test

Category Partition - Consequences

The Category-Partition pattern is general-purpose, non-quantitative, and straightforward. It does not require advanced analysis or automated support. However, identification of categories and choices is subjective. Skilled and novice testers may identify different categories and choices. Individual blind spots may reduce effectiveness. Bugs that are only manifested under certain sequences of messages to other methods or which corrupt instance variables hidden by the MUT's interface may not be revealed.

The size of a Category-Partition test suite is **...**

The ***Invariant Boundaries*** pattern may be used to produce a smaller test suite whose size is a sum of the number of choices plus one, not the product

With proper attention to interface details, a superclass Category-Partition method test suite may be reused to test an overriding subclass method

Category Partition - Uses & Sources

Known Uses

Elements of the Category-Partition approach appear in nearly all black-box test design strategies -- for example, [Myers 79] [Marick 95] [Beizer 95]. A study of manually developed Category-Partition test suites for a small C++ system found 55 out of 75 inserted faults. [Offutt+95].

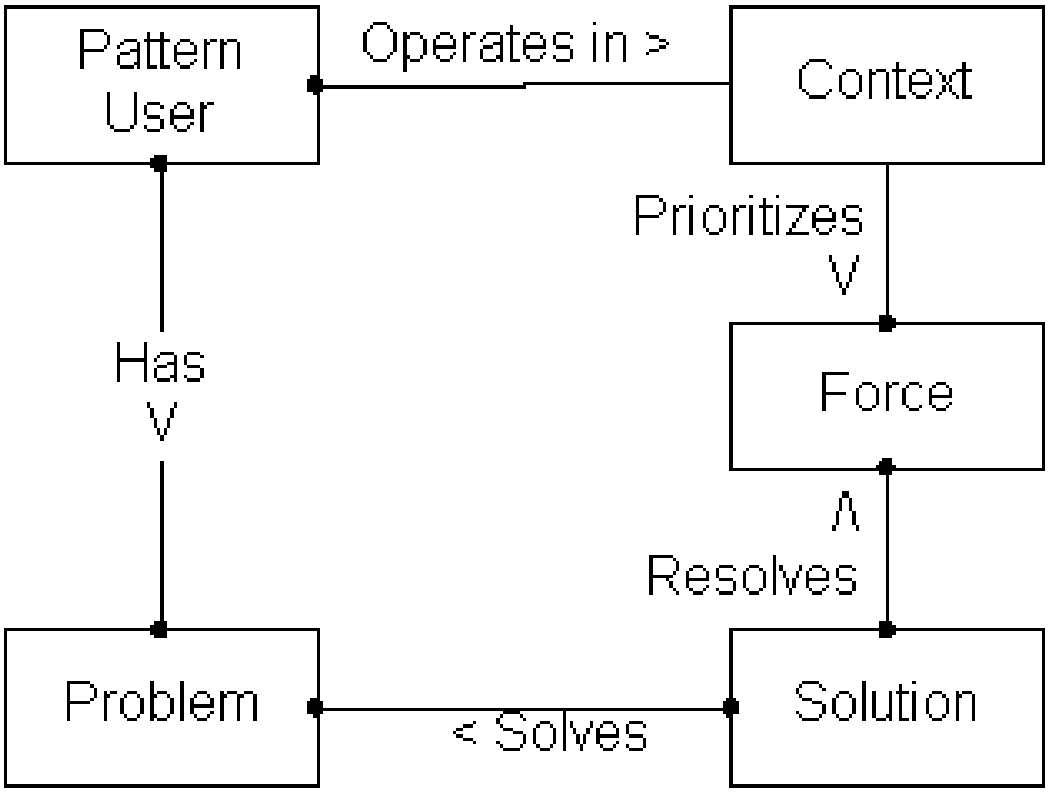
Cited Sources

[Balcer+89] Mark Balcer, William . . .

A Pattern Language for Pattern Writing

by Meszaros and Doble.

http://www.hillside.net/patterns/Writing/patterns.html



Pattern Writing Workshops

(PLoP™)

Chili PLoP™

™ PLoP is a trademark of The Hillside Group, Inc

First PoST

Jan. 3-5, 2001 at Rational in Lexington, MA

Introduction by Brian Marick

Pattern reading ("workshopping") demo

Pattern writing & new patterns workshopped.

Attendees: James Bach (co-host), Scott Chase, Sam Guckenheimer (co-host), Elisabeth Hendrickson, Ivar Jacobson, Cem Kaner, Grant Larson, **Brian Marick (co-host),** Noel Nyman, Bret Pettichord, Johanna Rothman (facilitator), Keith Stobie, Paul Szymkowiak, (co-host), James Tierney, James Whittaker

Architecture Achilles Heel

By Elisabeth Hendrickson & Grant Larson

Objective: Identify areas for bug hunting — relative to the architecture

Problem: This pattern addresses a problem defined by the following four questions:

- How can I use the architecture to evaluate the implementation of a system?
- How can I determine how to spend my time to discover the most severe bugs?
- Where can I spend my time to discover the significant risks affecting the system?
- How do you identify areas where bugs are most likely to live within a system?

Context: You are a tester and want to make best use of the testing time. Your goal is to find high severity bugs. You have a physical, functional, or dynamic map of the system in some form (or can create one with *Architecture Reverse Engineering).* You know what characteristics of the system are most important (e.g. is reliability more important than performance?). You may or may not be doing formal test planning.

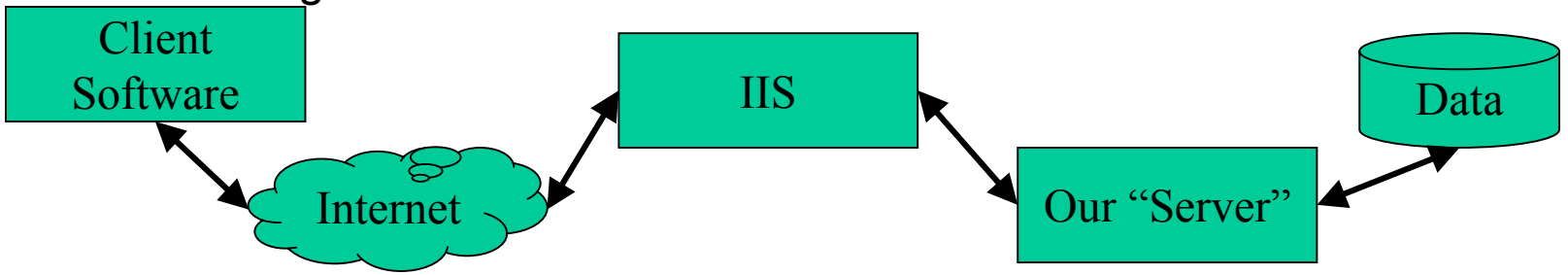
Achilles Heel - Forces

Forces:

- You may not be able to answer all your own questions about the architecture.
- Getting information about the architecture or expected results may require additional effort.
- The goals of the architecture may not have been articulated or may be unknown.

Solution:

An architectural diagram is usually a collection of elements of the system, including executables, data, etc., and connections between those elements. The architecture may specify the connections between the elements in greater or lesser detail.



Achilles Heel - Walk Thru

Walk through the architectural diagram asking questions:

- For each executable m the architecture, what happens if that executable is not running?
- For each data storage mechanism (whether a relational database, file in the file system, or other persistent mechanism), what if the data is corrupted? Deleted? Lacks integrity or conflicts with other data in the system?
- For each connection, what if the connection breaks? Does it matter if the connection breaks for a short duration or a long duration? (In other words, are there timeout weaknesses?)
- Are there multiple connections going into or out of an element? Does this open up the possibility of resource confli~ts or deadlocks?
- Are there shared resources where there may be performance bottlenecks?
- Do the elements of the architecture have states? Is there any possibility that those states might conflict—for example, could one part of the system try to start up a process while another part is trying to shut it down?

Achilles Heel - Rationale

As you answer the questions mentally or verbally, write down possible risks, weaknesses, or tests that occur to you. These become areas of the system on which to focus your energy. Alternatively, you could use the architectural diagram while exploring the system hands-on.

Where you are unable to answer your own questions, seek the assistance of others who can help you: programmers, architect(s), other testers, technical support personnel, etc.

Rationale:

Understanding the big picture of the system under test can help testers focus their energy appropriately. Without this understanding it's very easy to spend large amounts of time investigating relatively minor bugs or risks.

Achilles Heel - Result

Resulting Context:

You now know more about the architecture and implications of various design decisions and can use that information to guide your test effort. If the architectural goals are still unclear, perhaps an *Explicit Architectural Goal Articulation* is in order.

Additional Benefits:

- Increases communication between different groups within the project.
- Finds potential architectural flaws earlier.
- Aligns architecture with its goals.
- Provides additional fodder for test planning for current and future releases.

Achilles Heel - Liabilities

Liabilities:

- You may need to spend a large amount of time understanding the system before you can productively seek out bugs.
- Other members of the team or organization may need to invest more time answering questions to contribute to the test effort than anticipated.
- Some members of the team may feel threatened during this process.
- Information uncovered during this process may throw the project into chaos, at least temporarily.

Related Patterns:

- Architecture Reverse Engineering
- Explicit Architectural Goal Articulation

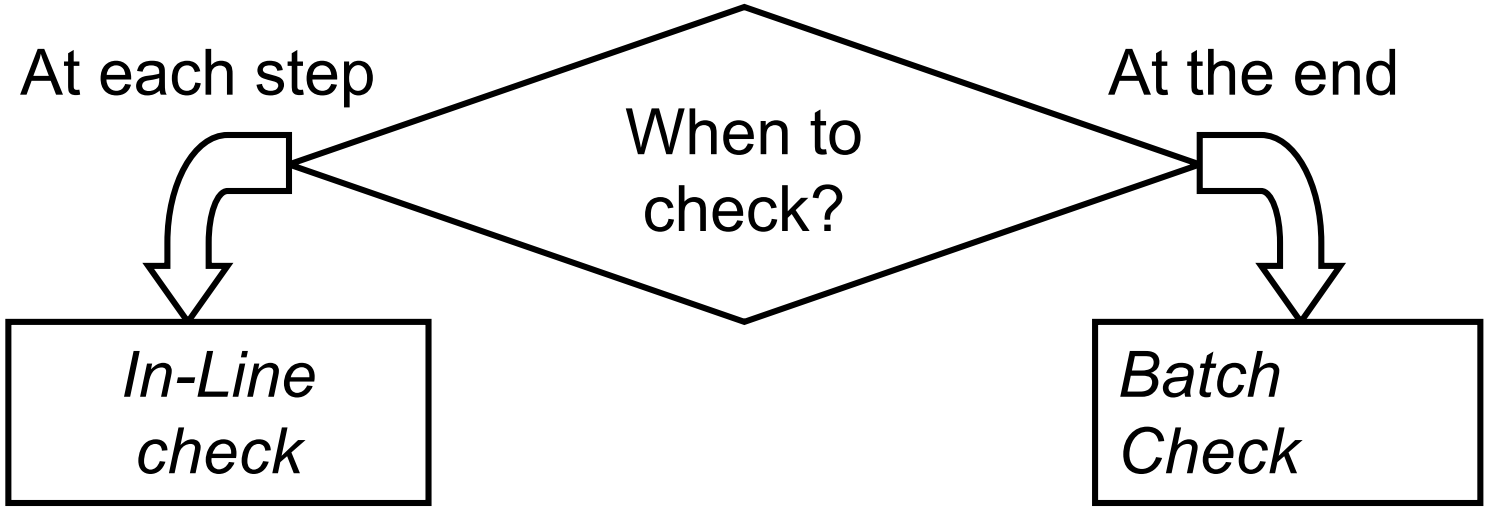
Test Results

The following patterns are a linkage between the Test Oracle Micro-Patterns for Pre-Specification approach patterns (Solved Example, Simulation, Approximation, Parametric) and Test Automation Design Patterns approaches Built-in Test and Test Cases as described in *Testing Object-Oriented Systems: Models, Patterns, and Tools*

The first two are complementary patterns for solving the same problem.

In-line check is the generally preferred method for ease of understanding.

Batch check (or *Benchmark*) is particularly useful for changing the *Judging* pattern into *Solved Example.*



Pattern Name: *In-Line check*

Context

You have pre-specification of the test results

Do you compare results at each step as you go, that is to known expected results, or all at the end?

Forces

- Future results may be invalid if early results don't pass (see also *Separable Test*s)
- Relatively exact checks of the results are desired.
- The amount of output per result is relatively small.

Solution

Check each result **in-line** as finely as possible immediately after its inputs are submitted. Show the input and expected output in the same file.

This can include, for example, bounding the time of the result.

In-Line Check - Indications/Rationale

Indications

- The results of a test input are precisely known ahead of time.
- Programmatic inputs are used

Rationale

It generally aids comprehensibility of the tests if the expected results appear in the same file and as close to the inputs as possible.

Resulting Context/Consequences

Maintainability is sometimes reduced if bulk updates of results are needed.

Reduces code reusability if inputs and results are hard coded into the code.

InLine Check - Uses & Patterns

Related Patterns

See *Batch check* for the same problem, but different forces.

Examples/Known Use

Frequently used for API/Class Drivers approach.

POSIX Verification Test Suite.

Expect tool

A suite of code was developed to check the ULOG in real time to enhance this style of programming when one of the expected results is a ULOG message. Thus some of the many javaserver configuration tests check for a ULOG message explicitly as part of the test.

InLine Check - Code Samples

Note below that the result is checked **in-line** in the code and not by some external entity.

Java/C++

```
result = squareRoot(1);
if (result != 1) {
LogError( "squareRoot(1) resulted in "+result
  +" where 1 was expected" )
}
```

Shell:

```
result=`squareRoot(1)`
if [ "$result" != "1" ] ; then
  echo "squareRoot(1) resulted in $result, where
  1 was expected."
  fi
```

Pattern Name: *Batch check*

Aliases:

Benchmark where a benchmark file containing expected results is used.

Context

You have pre-specification of the test results. The specification may be by hand judging actual results.

Problem

Do you compare results at each step as you go or all at the end?

Forces

The set of inputs is not easily separable (e.g. compiler input file).

The output can be compared easily with minimal filtering.

The output is voluminous.

Batch Check - Solution

Solution

Provide a benchmark file of expected results. Collect actual results as the test executes. At the end compare the expected and actual results.

Indications

- Output is difficult to predict and can frequently change from release to release (e.g. stub generation from CORBA IDL).
- A failure near the start of the test doesn't invalidate the results that follow.

Rationale

Very easy to develop. Expected results can be generated by the program once and hand checked for accuracy once and then reused again and again. This changes the *Judging* pattern into *Solved Example*. Expected results can be updated without effecting any code (since they are in a separate file). **Batch** processing may be the nature of item of test.

Batch Check - Consequences

Resulting Context/Consequences

One dangerous Consequence frequently seen is testers get lazy when the expected output has to change and don't scrutinize the initial results carefully enough for correctness. Then the actual, wrong, output gets canonized as the expected output.

Can make maintenance more difficult if the relationship between inputs and outputs is not very clear.

Frequently need special filtering patterns (regular expressions) to ignore uncontrollable extraneous differences, e.g. machine names, time stamps, etc. This filtering out has a small risk of missing incidental problems, like the time being reported wrong – generally you rely on other tests to specifically verify what most of these types of tests ignore.

BatchCheck - Uses & Patterns

Related Patterns

See *In-Line Check* as the generally preferred method.

Examples/Known Uses

Compiler testing or any transformation type program. It is generally too expensive to test each feature completely individually and a lot of common setup exists to test any one feature.

Code Samples

The abbreviated example below shows the expected output stored in a separate file and then a **batch** comparison done.

Shell:

```
echo 1 >expectedOutput

squareRoot(1) >> actualOutput
diff expectedOutput actualOutput
```

More PoST

STAR East presentation by Brian Marick

PoST2 - April 11-13 at Florida Institute of Technology, Melbourne, FL

Helene Astier, Hans Buwalda, Scott Chase, Elisabeth Hendrickson, Alan A. Jorgenson, Cem Kaner, Brian Marick, Florence Mottay, Melissa Mutkoski, Bret Pettichord, Nadim H. Rabbani, Jennifer Smith-Brock, Keith Stobie, Amit Singh, and Paul Szymkowiak.

Quality Week, May-June presentation by Keith Stobie

PoST3 – Aug 26-27, Lexington, MA

Test Pattern repository:

http://testing.com/test-patterns/patterns/

How Are You Going to Test All Those Configurations?

Mark Johnson
Cadence Design Systems
Portland Oregon
503.968.4801
mark.johnson@cadence.com

Abstract

Whether you are testing a traditional application such as a word processor, or you are testing web-based software, the prospect of all the possible configurations that could be tested is a nightmare. We ran head-on into this issue when we started producing web installable updates to our products on an every-other-month basis. We had too many combinations of OSes, browsers, licensing methods, product combinations, and other factors to try in the time available.

This paper describes techniques that we evolved which help us expand the set of configurations we test and increase our confidence that our products will function correctly on them. First, we refined our software development process to build the current development products and produce an installable CD image every day. Second, the test group adopted a process involving the establishment of a set of known clean baseline systems for the different configurations we want to test and the use of drive imaging software to quickly restore one of these configurations to a test system. Third, we created a table of the configuration options we use to assign unique testing configurations to each test engineer each time they restore a clean system. The combination of these techniques has allowed our project teams to increase configuration test coverage and reduce the number of configuration problems that make it to our customers.

Author Biography

Mark Johnson has been in the software and hardware development world for over twenty-five years. He enjoys being part of a product's development team. His primary interest is in helping teams and individuals to be more productive and effective in their work. He has taken many roles in past organizations, including software development and testing, team leadership and management, leading and consulting on software process improvement, and hardware test engineering. Mark has been active in the software quality and testing community for many years, volunteering with conferences, presenting papers, and writing articles. He has Bachelors and Masters degrees in Computer Science.

How Are You Going to Test All Those Configurations?

Mark Johnson

Introduction

In this paper I will talk about a set of processes that our team developed which have allowed us to expand the set of software configurations we use during testing. Configuration testing is the testing of a piece of software with different combinations of related software and hardware. Configuration testing needs to be done, because differences in the software and hardware configuration can affect how the software under test behaves. An example of configuration testing would be to test a PC program with different PC operating systems (OSes), such as Windows 98 and Windows 2000. Testing on both of these OSes would be important because the OSes provide different versions of support software for functions such as the graphical interface and the file system.

The Problem

The Orcad family of products are targeted at the shrink-wrapped Windows PC market. This means that they are sold to a large customer base, and a key objective is to minimize the customer's need for support to install and run the products. While PC hardware has become fairly standardized, there can be a fair number of supported software configurations to test. In earlier days, our products just needed some testing on Windows 95 and Windows NT. With the addition of more Windows OSes, and changes to our products that increased their dependence on optional software such as web browsers and database interfaces, our configuration testing became more complex. With the most recent Windows OSes and web browsers, we can have combinations of the following:

- Operating Systems
  - Windows 98
  - Windows NT 4.0 with service packs 4 or 6a
  - Windows 2000
- Web browsers
  - Netscape 4.x and 6.x
  - Internet Explorer 4.x to 5.x

In to these factors from the customers' environment, we have added new products that require close integration and more complex methods of installing and licensing our products. This means that product combinations and versions and installation and licensing have become important configuration factors.

In addition to the increasing complexity of our configurations, we made changes in our software development process that increased the possible configurations while at the same time reduced the testing resources and testing timeline. Traditionally our product development team had all worked together on the next big release, typically with a six to nine month release cycle. We changed by sub-divided our large team into smaller teams that worked in parallel, with each small team completing work on a specific feature area

of the product in six to twelve weeks. The work of each of these small teams is released on our website as a download that updates our existing product. The net result was that each small team would typically have one or occasionally two testing engineers, and they would need to cover the OSes, web browsers, various mixes of products, and now product updates, all in a shorter period of time.

Even with the old process of six to nine month release cycles and six to eight testers, doing a full testing pass on multiple configurations was out of the question for us. The approach we had adopted was to assign one or more testers to each major configuration. They would use their assigned configuration to do their normal testing work for the release. When we went to small teams, the one or two testers on a small team now had to cover the configurations themselves.

Fortunately over the period of time leading up to going to small teams, we had been evolving several processes that improved our ability to cope with more configurations and less time.

Source to CD Everyday

Our product development team had a tradition of doing a nightly build of the current baseline software. This let the developers know that the checked-in code would compile and link. Once or twice a week the test group would run a 'smoke' test on the build results. If the test passed the build would be copied onto a testing file server. Testers did their day-to-day testing using the software copy on the file server.

We did a limited amount of installation and configuration testing at the very end of a release, when the 'golden' CDs with the final software were available. This seemed sufficient because we only had three or four products, and we released them one at a time on separate CDs. Since all software development was done at one site and the group was small, any problems we ran into with installation and configuration testing could be fixed by talking to the right people and generating a new set of golden CDs.

Then we added two products that changed our integration needs. First, we purchased a small company located five hundred miles from our office. Their product is an add-in to ours that allows the user to access information from databases and the Internet using database support software and a web browser. Then we merged with MicroSim, a company of equal size to Orcad, located one thousand miles way. Their flagship product was the PSpice simulator. The typical user of PSpice is doing frequent cycles of experimentation during design, so there needed to be very tight integration between our design entry product and the PSpice simulator. With these changes we decided that all products should be released at the same time on the same CD.

The tight product integration made us realize we wanted to make a fundamental change to the way we set up our testing environment. As integration was developed between the products, we needed versions of each product in which changes were synchronized, so that we could test the integration features. With the tighter product integration, we decided that we wanted to begin doing our testing using configurations that would match a customer's installation.

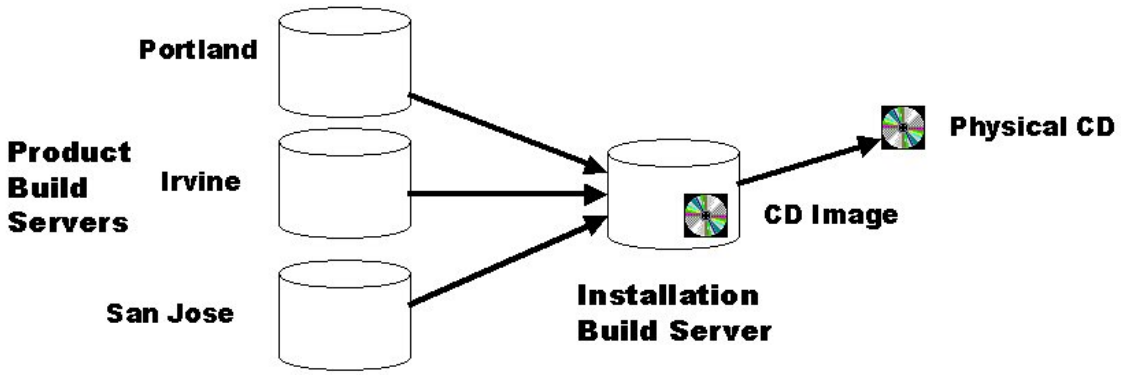


Figure 1---Source to CD Process

To support this, we set up the process we call 'source to CD everyday' as shown Figure 1. We took the automated processes we had for creating the product CDs, and set them up to run after the nightly build. In this way, each morning we had an installable CD image we could use to load the current baseline software for testing. Getting to a CD image every morning was not as easy as it may sound, since we were now dealing with software development at three sites, one of which had a slow connection to the rest of the company. To accommodate the site with a slow connection, we ended up picking up their files as best we could each night, and using a previous copy if we couldn't get the latest files. This requires that the bill of materials (list of files and installation locations they are copied to) be maintained as development progresses. While this sometimes causes problems immediately after file organization changes are made, the benefit is that these problems are uncovered while the changes are still fresh in the developer's mind.

Because licensing and installing our software takes only five to twenty minutes, depending on the configuration chosen, testers are able to uninstall and reinstall our products pretty much every day prior to beginning that day's testing. This greatly helps us find problems with missing files, interface incompatibilities, etc. the day after they occur.

When we decided to go to small teams working in parallel, we were concerned about how to continue this 'source to CD everyday' process. The first thing that had to happen to support multiple small teams working in parallel on the same product was to set up separate branches for each small team's work. Our configuration management team supports this by establishing the new branches when they are needed, and starting up a separate nightly build process for each branch. Since the work of the small teams would be released as a web update, each small team would need its own web update building process. One of our developers took the tools our installation team used to create web updates and created a template web update and instructions for starting a new one. With this template, a newly formed small team spends about four hours during project startup putting their build process together. They then have an automated nightly build process that generates a web update and places it on an internal website to download for testing.

So now we have a 'source to web update everyday' process running for each small team. This allows the tester for that small team to do an uninstall and reinstall each morning, so that they are testing the latest version of their team's software.

Clean OSes by Drive Imaging

Once we had started on the source to CD everyday process, we realized that we needed to have clean systems to support installation and configuration testing. Our definition of a 'clean' system is one which has not had our software installed on it. We had been depending on the uninstall process for our products to clean off any files, directories, registry entries, etc. We knew that the uninstall process was not perfect at cleaning everything up. The best way to ensure that we were not overlooking anything would be to start a testing session with a PC that had freshly formatted hard drives and a newly installed OS. This is what we call a 'clean OS' system. We would do this periodically, but it would typically take four to eight hours for the tester to complete, so we were reluctant to spend the time more than once every few months.

In looking for a solution, we talked to our IT department. They use a utility program for setting up new PCs with our standard software configuration. Basically, it is a backup/restore utility that makes a copy (called image file) of the physical layout of the hard drive and allows you to restore the entire hard drive later. When IT gets a new set of PCs, they format and install the OS and other applications on one of the new PCs. Then they use the utility to make an image of the hard drive on that PC. For the rest of the PCs, they simply restore that image and the hard drive of each PC is an exact match of the first PC.

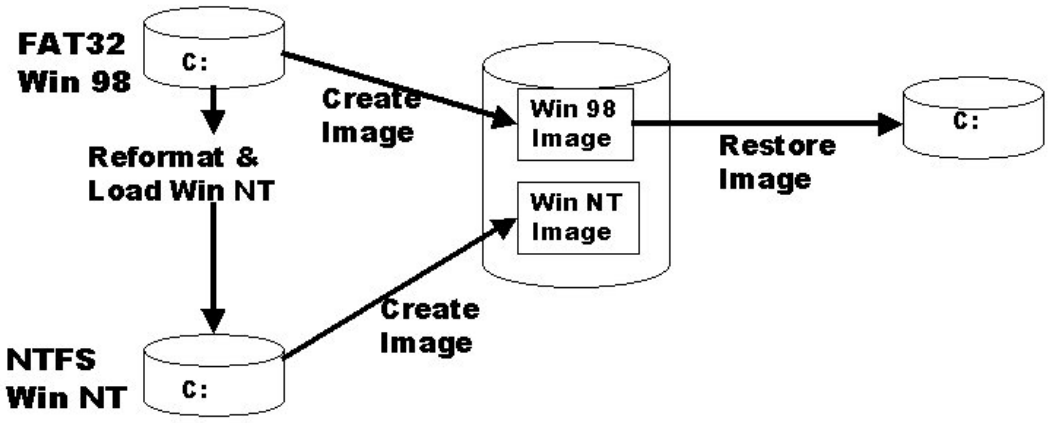


Figure 2--Drive Imaging Process

In testing, we realized that this utility might provide an easy way for us to rapidly restore a testing PC to a known state. Figure 2 shows the process we use for creating and restoring these clean OS images. It only takes about five minutes to restore an image file. When a tester receives a new PC they format and partition the hard drives, and load one of the OSes they want to test with, such as Windows 98. They make an image of this 'clean OS' and save it. To create another OS configuration for the same PC, they reformat the hard drives and install another OS, such as Windows NT. In this way, a tester can have as many different clean OS configurations as they desire. A nice feature of drive imaging is that it works at the physical level, so you don't have to worry about the current contents of the hard disk or its file system format. For example, if you are

changing from Windows 98 where the hard drives are formatted FAT32, to Windows NT where the hard drives are formatted NTFS, you simply restore the Windows NT image and the file system is changed along with the disk contents. In this way the tester can simply load the image for the OS they want to use next and they are ready to finish setting up the configuration.

This process works well for providing a known clean starting point for testing and for easily switching from one OS configuration to another. Because the testing group has a mix of different PCs that have been acquired over time, we are not able to make one set of images that all testers will use. This is due to the different drivers and hardware specific configuration information needed for the different PC types. So it takes a tester about four hours to set up each new image when they get a new PC. Fortunately, once the images are set up, they require little day-to-day maintenance. Since Orcad has been acquired by Cadence, we have run into a problem with how Cadence administers its networks. Something happens on an every couple of months basis that makes stored Windows NT and Windows 2000 images stop logging into the network. When this happens, we have to have a system administrator delete and re-add the system on the network, and then have to make a new image. This takes less than one-half hour, but it can take a few hours to get the administrator's help.

With the use of clean OS images, we have been able to find a number of installation and configuration problems with our products that we otherwise might have missed. These are typically wrong versions of files, missing files, or missing or wrong registry entries. And even if you need to work around the problem today by manually copying a file or editing the registry, the next time you restore a clean OS image, you can determine if the problem has been corrected.

Planning and Managing Configurations

We now had an installation CD image every day from the latest software build, and a way for testers to easily set up a clean OS environment. However, we found that we weren't getting the coverage of different product installation and licensing configurations we wanted. It seems to be human nature that testers fall into routines with preferred configurations they use for testing. We also found that after a week or two of testing, it was difficult for a tester to provide a record of the exact details of each configuration they had used. So it was hard to know exactly which configurations had been covered and which remained to be tested.

To overcome this, we added a table of test configurations to the test plans we write. In the table the columns list the different configuration variables to track. In the rows, we select specific combinations to try and assign a tester to each configuration number. The table also gives us a shorthand way to record configurations in our testing notes. See Table 1 for an example configuration table from a past CD release. The testers on the small teams use the same type of a configuration table.

We choose combinations of configuration variables by trying to cover most pair-wise combinations. However, we skip some pairings to reduce the total number of configurations. We do this by dropping configurations that would repeat the use of configuration variables our customers are less likely to use. For example, the majority of

our customers are now on Windows NT and starting to move to Windows 2000. So we drop some combinations that would use Windows 95 and select Windows NT instead. For the small team projects which are released as web updates, our concern is more with configurations having earlier web updates installed and using different web browsers, because these factors can directly affect the success of web update installation.

We have found that because of the number of variables involved in the typical configuration, it can take several hours to fully set up a configuration that uses less common options. Because of this, we have scaled back from testing a new configuration every day to testing two or three configurations per week per tester. On a small team with a single tester, we have found that ten to twelve configurations can be covered during the testing part of an eight-week project.

CD Testing Configurations

	Locking				Server/Client			OS				Browser		Products to Install										
CFG	NIC	Hasp	Rainbow	Inst	Svr	Cl	Oth	95	98	NT	2K	Netsc	IE	Cap	eCap	Demo	Cap+AP	CIS	PSAD	PSP	PSADB	Opt	LayEE	Lay+	Lay
1	x										x		x	x									x		
2				x							x	x		x											
3		x								x		x	x				x			x					
4	x									x			x					x	x			x			x
5		x								x		x		x						x			x		
6			x			x	x	x					x					x			x				
7	x								x			x						x			x	x			
8		x			x						x	x					x		x			x		x	
9			x							x			x					x		x				x	
10				x						x			x	x									x		
11		x									x		x					x							
12			x								x		x					x	x			x	x		
13	x									x		x	x					x	x			x	x		
14			x				x				x	x				x									
15	x												x					x	x						x
16		x				x		x				x			x										
17			x		x				x				x				x			x		x		x	
18	x										x		x	x					x			x	x		
19		x									x	x			x										
20	x									x		x					x			x	x			x	

CFG – Configuration to test using all the options specified in that row.
Locking – What method is used to license the installed software.
Server/Client – Install the software in a stand-alone (no 'x' in the column) or in a client/server combination.
OS – Which OS to use.
Browser – Which Internet browser to use.
Products to Install – Install this combination of our products and updates.

Table 1: Example configurations table for testing a CD Release

Results and Issues

We have found the idea of providing some test coverage on a larger number of configurations, rather than more test coverage on a fewer configurations, to be effective. We went from testing on three or four basic configurations to testing as many as twenty important combinations of configuration variables. And we were able to accomplish this configuration coverage using only one or two test engineers during a two-month development cycle, while completing our other testing tasks.

Over the year and a half that we were doing small team projects and handling configuration testing for this product, we shipped seven web updates and three CD releases. During the same time period, we needed to coordinate our testing with five web updates and three CD releases for the two products that closely integrated with our product. This configuration testing gave us a product that installed and performed properly in the wide range of environments that our customers use.

During this period, we found many installation and configuration related issues during our internal testing. These were typically missing files, wrong versions of files (a file has been updated but not made it into the build or installation process), or occasionally conflicts between installations of two different products.

I am aware of only two configuration-related problems that escaped from our testing. One was due to a last minute bug fix. It required adding a file to the file set that made up the web update. This change happened after the majority of configurations had been tested, so we missed an installation ordering dependency. We were able to address this by listing the required installation order on the web site containing the web update download files. The other problem was a configuration management issue where files added to one web update did not get checked in correctly and were left out of the subsequent web update. Our web updates were cumulative, so as part of the testing for each web update we would verify that the functionality added in any previous web updates was present. These two web updates happened almost simultaneously and we didn't verify the second web update had all the correct contents until it had been released. Once the missing files were discovered, these files were added to the second web update and it was re-released on the software downloads web site.

The Future

After Orcad was purchased by Cadence Design Systems, the product we had worked on for the past several years was transferred to another group for maintenance. The majority of our project team has been merged into a new product development group that is in the early stages of producing a new product. This new development group is geographically dispersed and is still working to stabilize its software development practices. Until the development practices stabilize, we will not be able to implement this type of configuration testing.

The team that took over our product follows the software development practices of Cadence, where daily builds of the complete product and regularly producing customer installable CD images have not been an objective. Most Cadence products are installed at the customer site by a Cadence application engineer, rather than by the customer. Cadence products also have a closely controlled set of configurations they support.

Therefore the group now maintaining our product have not adopted our configuration testing processes.

It is interesting to note that two of Cadence's engineering objectives for the coming year are to get to a nightly build for all products, and a weekly customer installable release. These changes are planned to allow testing products in an environment that more closely matches the customer-installed environment to ensure that all the parts of the products work together. These planned changes give us hope that our practices of mixing configuration testing in with other testing work will be adopted in the future.

Recommendations

The process we developed is made up of a few basic steps that are good software development and testing practices:

- Plan the configurations you need to test, track your progress, and adjust your plans as needed.
- Establish a fast and efficient way to set up clean testing environments.
- Create a system that gives nightly builds of your software under development, with the results coming out in a form that allows you to install it as a customer would.

I would recommend this process of testing across as many user configurations as you can for anyone dealing with software installed in the end user’s environment. This configuration testing can be done in a way that does not add appreciable overhead to your testing process. I believe that this will help you find more configuration-related problems and find them earlier in your testing cycle, resulting in happier customers and possibly shorter release cycles.

Testing a Bluetooth Product With Web and Embedded Software

Rick Clements
cle@cypress.com
Cypress Semiconductors
9125 SW Gemini Dr., Suite 200
Beaverton, OR 97008

Abstract

Until recently, testing of embedded software has involved different tools and concerns than software for the Windows and Internet environments. Bluetooth is one technology that is facilitating the creation of products that cross these traditional boundaries. This paper looks at the test planning involved for an example Bluetooth based product that has Internet, Windows and embedded components.

This paper works though the steps of initially scoping the QA efforts and testing efforts, creating a QA plan and creating a test plan. This is done with an example product. The appendices contain the documents that are the result of the analysis done in the paper.

The description of the example product isn't as complete as needed for a real project, but it's sufficient to identify the areas that need to be addressed. In a real project, much effort is required to resolve the open issues. For this paper, it's sufficient that we understand the issues exist.

Biography

Rick Clements is a senior software quality assurance engineer at Cypress Semiconductor. His duties include managing the release of a single CD with software from multiple divisions and testing of software for programming CPLDs (complex programmable logic devices). He has 20 years of software experience including built in self-test, embedded software design, embedded software test and process improvement.

Glossary

API	Application Programming Interface
Bluetooth	A standard for short-range wireless networks.
CPLD	Complex Programmable Logic Device
GUI	Graphical User Interface
MP3	A format for recording music that incorporates a large amount of compression.
QA Plan	A plan describing the steps necessary to produce a quality product.
Test Plan	A plan describing the steps necessary to validate the product against its requirements.

The Product Under Test

The Leisure Time 2000 is an imaginary product with enough features to illustrate the issues that need to be considered when products that execute in the Internet, Windows and embedded environments. As it's described in Appendix 1, it allows the downloading and playing of e-books and MP3 music. (The appendix isn't intended to be a complete requirements specification or functional specification. It is intended to provide enough information for the example in this paper. Like a real project, the process of developing tests will show missing detail the requirements.) It is a complex system with many requirements to be tested.

The unit its self allows playing of music while reading or downloading. Playing the music must happen in real time. Changing pages and moving to the next chapter must be performed quickly but they are more tolerant of delays than the music. Downloads must be tolerant of noise and short interruptions. The download must be gracefully exited if the download can't be completed.

The desktop program needs to provide a list of music and books. It needs to provide a conduit for the user to access the Internet. It can be connected to the Internet with a physical connection or a wireless connection. It's connected to the Leisure Time 2000 via a wireless Bluetooth connection. This brings in the question of compatibility testing with different browsers and web sites.

The Bluetooth Interface means the user may want to download from multiple computers. Since there are no wires, the download needs to be able to continue as if uninterrupted if the user walks back into range after being out of range of the connection.

The product includes a physical device. Someone has to be responsible for the electrical and mechanical testing. In some companies the software test group is separate from the electrical and mechanical test groups. In other companies there is a common test group. This paper will assume all of the testing is being managed by a single testing lead. This adds issues about drop testing, EMI testing and UL certification. If these issues are being managed by a single testing lead, there must be people with expertise in these areas on the team. These issues will be mentioned in this paper. It is assumed other team members provide this detail.

Scoping the Effort

Appendix 1 lists a data sheet for the Leisure Time 2000. The purpose of the description isn't do define a viable, marketable product. The purpose is to define a product that illustrates the issues faced when technologies bring different development environments together.

The list below shows some of the differences between the web environment and the embedded environment. These differences affect not only the design philosophy, but also the testing approaches as well. The difference in types of software creates many QA and testing issues. However, this product is not only a software project. It contains electrical and hardware components. This complicates matters even more.

Internet and Windows Environments	**Embedded Environment**
Compatibility with different web servers running different operating systems	Runs on a single device
Web servers may be busy or unavailable	Playing music requires real-time time limits of milliseconds or microseconds
Large amount of memory locally, storage locally and storage on servers	A limited amount of memory and storage
Tools for testing memory leaks, measuring test coverage and automating tests exist	Many standard tools can't be used because of the lack of memory or because they affect the real-time performance of the system under test.

Just the difference in types of software creates many QA and testing issues. However, this product is not only a software project. It contains electrical and hardware components. In some companies, the software, electrical and

mechanical designs are handled by separate teams. In other companies, an integrated team handles the entire design. Since system tests will cover features with software, electrical and mechanical aspects, this paper will assume a single team. Our team has the responsibility for all QA during development and all testing at the end of the development cycle.

We need to provide initial estimates of time and resources needed for testing before the requirements are complete. To do that we will organize the information in the worksheet found in Appendix 2. The estimates given at this time will be very rough since all of the requirements haven't been defined. The estimates will be refined when the requirements are complete and again when the design is done. When the design is done, we will know exactly what interfaces we will have and identify high-risk areas that will require additional testing resources.

The product is defined as having a web component, a hand held component and a PC providing the interface. The software testing will be divided between into two sections. The first section is the web and Windows related software. The second section is the driver on the PC and the hand held unit, and the embedded software. For electrical, mechanical and testing for regulatory agencies, we will have engineers that specialize in these tests. They will provide information to the design engineers to help them meet the EMI tests for the FCC & the tests for the UL.

QA Plan

This section works though identifying the QA issues for the project. The result of this analysis is a test report that appears in Appendix 3.

For a real project, we would have a requirements document before we start this step. The requirements specification would take the statement in the data sheet and give testable requirements. For example, where the data sheet says, "runs on Windows", the requirements specification would say, "runs on Windows 98, Windows NT, Windows 2000 and Windows ME." Since this is an illustration, we will work from the requirements in the data sheet.

Configuration Management

Configuration management is important in a any project. It becomes vitally important for a multi-discipline project. If during design, the requirement for displaying the power remaining in the battery is discovered, many aspects of the project are affected. The battery capacity needs to be displayed in the device's GUI. A gas gauge chip needs to be added to monitor the charging and demand on the battery. It must be electrically part of the circuit. It must be physically placed into the device. Software needs to interface to the API (Application Programming Interface) of the gas gauge chip. A test needs to be created to verify the correct information is displayed to the user. Configuration management must assure that the software design, software code, software tests, electrical design, mechanical design and prototype boards all reflect the change.

An added difficulty is that the electrical and mechanical engineers can see configuration control as a "software thing". If the device contains a CPLD (complex programmable logic device), that CPLD has a programmable circuit. The line between describing the circuit and programming is very blurry. VHDL is one language used to program CPLDs. It is a HDL (hardware description language) and shares a number of features with software programming languages. The code its self can easily be put under source code control. Many tools provide a graphical programming environment. This present the same challenges as the programming environments that are common for software being developed on Windows.

While the discreet electronics design is less like software, changes to it must also be tracked. They changes often require matching changes in the software. For example, if the electrical scaling of an A/D device is changed, the software must be changed to match. If configuration is handled statically, the version of the software and hardware must be identifiable so a technician can tell the software and hardware match. If the software handles configuration dynamically, the hardware components must be able to report their version to the software. If the reporting of version information is going to be used, it is a feature that needs to be added to the requirements documents. This issue should be added to the problem reporting system and closed out when the requirements document is updated.

Problem Reporting

A common tracking system allows problems to be reclassified. A problem may be originally classified as a software problem. For example, "the software isn't driving the display bright enough." After investigating, a software engineer may discover a value reported is twice the correct value. The problem is reclassified as an electrical problem. When an electrical engineer investigates, she discovers a resister network is wrong. However, it's too much effort to change the resisters that are stuffed into the board, so the decision is made to fix it in software. At that time, it's moved back to the software category.

Test Plan

Functional Tests

The major functions are:

- playing music
- reading a book
- downloading music to the device
- downloading a book to the device
- downloading music to the host PC
- downloading a book to the host PC

For the host testing, We have the advantage of a number of commercial test tools. For testing the device the choices are limited. Qronus and National Instruments have tools. Qronus builds a PC test system for you. For this example, we have decided it's easier for us to build that system because we understand our interface. The National Instruments tools strength is in automated control. For this example, we would still have to create DLLs to interface with our device and our group doesn't have experience with this tool. So, we have decided it is easier to create the testing tool ourselves with Perl for the scripting language and C++ code providing the device interface. Creating this tool is a real software development project. It needs to be treated like a real software project. If we hack a tool together, it won't be reliable, people won't believe the results and it will create more work than it eliminates.

To automate the device testing, we need to be able to send commands to it from the PC and have a copy of the commands sent to us. This requires hooks into the product. This interface could be implemented as internal RS232 connection. However, manufacturing can make use of this interface for their testing. They need access when the unit has its covers on. A command can be sent from the host to enable the extra information over the Bluetooth port. This is an added requirement of the design of the product. It needs to be identified and negotiated with the designers in the beginning of the project.

The host related tests need to be tested on all of the supported operating systems. The initial release of the product is limited to the PC running Windows. Which versions of Windows need to be tested? This is in the product definition. Which version and patch levels need to be tested. We decide to limit our tests to the latest version and patch level. We still have to test on Windows 98, ME, NT and 2000. This assumes engineering was able to talk marketing out of supporting Windows 95. Depending on the time frame, Windows XP may be an additional platform that requires testing.

The host-related tests need to test the ability to download from a server, manage the files on the host and the related GUI. We are building on top of the network access in the operating system, so we don't need to worry about creating an isolated net to protect the world form bad packets we might generate.

Verifying that the books are displayed properly and the music is played properly is best done in a manual test. It is possible to build test fixtures to automate this testing. However, it would be the 10% of testing that required 90% of the effort.

Host / Device Protocol Testing

Data, commands and status are passed between the host and the device. Some of the interface is defined by the Bluetooth standard. There are data formats that we define. Even though the project team defines these protocols, they still have to be right. These tests are important when initially integrating the host and device. These tests are also important regression tests when the host or device is upgraded. These tests need to be automated.

The Bluetooth interface chips will handle some error correction, data retries and changing channels. However when the signal is too weak or there is too much interference, the system will have to gracefully handle the errors. These tests will require both a RF signal generator and a lab large enough to separate the host and device.

Stress Tests

The functional tests verify downloading, playing music and displaying text. However, can the device do all of these things at the same time? Downloading can happen in the background. It isn't time critical and it shouldn't hog the system. Scrolling though the book should happen quickly. Much more than half a second will be noticeable. The music is the most demanding requirement on the system. It must have real time response or the music will be choppy. The stress test will perform all of these functions at the same time.

In the functional tests, we listed our automation interface as using the Bluetooth interface for its physical layer. This means automation status and commands will compete with the download for bandwidth. Regardless of the physical layer, the software will pause while the data is being transmitted. Both of the characteristics will change the system timing. For these reasons, the stress tests will be manual tests.

Compatibility

Compatibility tests are a challenge. It's not possible to verify compatibility with everything in the world. Even if that was possible, the list would be obsolete the day after the product was released. Two areas that need to be verified are the web sites providing content and Bluetooth cards in the host PCs. Marketing must come up with a list of sites and a list of cards that are important. The challenge will be to limit the list to a manageable size. Marketing will not want to eliminate any possibilities.

To increase the number of sites that can be covered, the sites are divided into the important (silver) and very important (gold) sites and devices. The silver lists will get light testing. The gold sites will get more extensive testing. Marketing will often want every site in the world to be gold. It is often useful to tell them what level of testing they will get with a list of sites that long. This will be some level below silver. They often see the benefit of giving some sites more testing.

The other part of compatibility testing is what happens when there are changes to the to the sites or Bluetooth cards? Management has to make the decision to test changes or not. They have to balance the cost of retesting against the Product's image over time. If these decisions weren't in the original project plan, the plan needs to be amended. For this example, management has decided that the web sites must be tested on an on going basis. An agreement with the site owners will have to be made to test the device with their site or notify our company of major changes. Management has decided that major changes to the hardware cards will happen slowly enough not to warrant testing of changes.

Reviews

The project reviews are defined as part of the QA process. The design reviews and code reviews are a testing function. It's important to show management that the time spent in these reviews will result in larger savings during the testing phase of the project. Some time because the number of tests can be reduced. More time because of reduced rework time. If an error can be caught before a large amount of software, electronics or mechanical enclosures are built around it; it's much less expensive to fix. If you can get the amount of time saved from projects within your own company, the estimate will be more accurate and management will give it more weight. If not, industry data can be used.

Testing the ability to handle multiple languages and host platforms would require an effort at least as large as implementing them. These requirements are better verified by design review.

Areas of high risk need to be identified and reviewed. Interfaces are traditionally error prone. One Mars landers demonstrated the worst case for an incomplete interface. In that case, it was one side of the interface using metric units and the other side using US units. Most projects use one measure system. However, scaling differences often occur. For example one module is using milliseconds and another component using tenths of a second.

These interfaces include the PC to handheld device and electrical to software. Part of the interface between the PC and handheld device is defined by the Bluetooth standard. Even though the project defines the rest of the details, the details have to be defined and used consistently.

Mechanical Reliability Tests

The UL tests cover some of the reliability testing. They test things like does it catch fire when it's dropped. For the UL tests, there needs to be an interface with the external UL lab. The test group will also provide experience in helping the mechanical design engineers perform their equivalent of unit and integration testing. The tests aren't inexpensive and worse the lead-time to resubmit the product if it fails can be long. The chance of failure must be low.

An area the UL tests don't cover is if the product is dropped, will it still work? The requirements specification must specify how high of a drop it must withstand and how many drops it must withstand. Drop testing will destroy prototypes. Prototypes must be allocated with the knowledge that these prototypes are consumables.

The unit must be shipped to the distributor and the customer. The packaging must protect the unit from greater mechanical stress seen during shipping.

Heat Related Tests

The requirements document will list the necessary operating and storage temperature ranges. This product has no moving parts that may become stiff when cold. Commercial parts will have a low probability of failure at the lower limits. These tests need to be done, but there isn't a large risk that they will fail. The product overheating at the high end of the temperature range is a bigger risk.

The LCD is temperature sensitive. Changing the LCD late in the project could affect both the electronics and mechanical package. Buy the time the product gets to test, it's too late in the process to be changing the LCD. Testing needs to ensure the selection process tests a sample LCD over the temperature range.

These tests require a thermal chamber so the ambient temperature can be accurately controlled. These will be long tests because the chamber needs to stabilize and time is required for the ambient temperature to affect the internal temperature.

While drop tests will break units, the thermal tests will stress the systems. This can make prototypes unreliable. It can't be assumed that all of these prototypes can be given to software test at the end these tests. The resource plan must account for the units that aren't reliable enough for continued use.

Power Tests

In response to user request, the device needs to display the amount of battery capacity. It must display a warning message soon enough the user has a reasonable amount of time to change the batteries. To do this, system needs to be run from fully charged until it runs out of power. This test can be automated. The test fixture can record the power level that's being reported and when the warning message is displayed. When it loses connection with the device, it can log the shutdown of the system.

The device can lose music and books stored on it if it loses power for too long. Data such as its serial number and number of hours of operation (used to determine if the device is under warranty) must be maintained. When the unit is repowered, it must reinitialize its self properly.

Another test is needed to verify that volatile memory is preserved long enough when batteries are being changed.

EMI Tests

The FCC requires the device be tested for EMI emissions at a certified lab. Since the lab will be able to create signals at specific frequencies and power levels, the susceptibility tests can be done at the same time. In addition to general system tolerance, there is interference in the Blue Tooth band. The most notable of these are 802.11b networks, Bluetooth devices not part of your network and microwave ovens.

What Else Is Required?

The QA and testing schedules were specified to be in the project management plan. The schedule still needs to be created and maintained. This is done the same as the embedded or host based products you have worked on before. I find a WBS (work breakdown structure) an important tool for creating a reasonable schedule. The WBS describes what each task is, what's tasks must precede it, what equipment is needed, what signifies the task is complete, the estimated duration of the task and who will work on it. This is information needed to create the schedule. It's also valuable if tasks need to be re-assigned because of project surprises.

With all of the schedules and plans in place, you need to execute them and managed any surprises.

This paper has illustrated the types of issues that must be dealt with during the testing and QA of a multi-discipline project. More important than knowing the questions is the types of people you will need on the project. Even if you are only in charge of the software, the Windows and Internet software is very different from the embedded software. Having a good team is essential.

Appendix 1 - Product Data Sheet

Leisure Time 2000

The Leisure Time 2000 allows people to take a book and reading environment to any location.

Features

- A rugged hand held device
- Bluetooth interface to the host computers
- Download entire e-books, a range of chapters or mp3 music
- Replace selected books or chapters
- Select chapter and book to read
- Create a play list from downloaded music and listen to the music though headphones
- Download from computer
- Download from the internet though a computer
- Runs on Windows
- Macintosh and Linux hosts to be added later
- Allows for multiple languages to be added later
- Long battery life

[These aren't acceptable requirements to create tests (or design software) from. As noted in the paper, part of the testing effort is to help turn alleged requirements into real, testable requirements.]

Marketing Analysis

Lots of units will be sold. The volume will ramp up quickly.

Appendix 2 - Testing Strategy Worksheet

Application Name: Leisure Time 2000	**Author:** Bob Smith **Date**: 1 Jan. 2000
Type of Software: C++ code running in an embedded system. Java code running on a computer	**Development Methodology:** Waterfall
Scope of Testing: Software, electrical, mechanical, thermal, EMI & UL	
Critical Success Factors: Music must play smoothly Must download music and books	**Tradeoffs:** Schedule - yes Scope - no Cost - yes Performance - no Quality - no
Testers: Electrical, mechanical & software debug - designers Embedded software - 1 software test engineer Windows & web software- 1 software test engineer Internal mechanical & electrical - 1 mechanical test engineer EMI & UL tests - environmental test engineer	**Timelines:** Requirements - Jan. 31 Design - Mar.. 28 Test cases - Mar. 28 Coding complete & prototypes - June 30 Debug complete - July 30 Test procedures complete- July 30 Trade Show - Aug. 15 Testing complete - Sept. 15
Risks: Bluetooth is a new technology Software includes web, Windows and embedded software The product will be shown at a trade show before testing is complete	
Constraints:	**Assumptions**:
Test Approach: Electrical simulation, mechanical simulation, software debugging and unit testing will occur before formal testing. Formal testing will include both integration testing and system testing. EMI testing and UL certification will be done by certified labs. Drop tests and temperature tests will be handled internally. Expandability to cover future requirements will be verified by inspection.	
Tools: Windows and Internet features will be tested with a commercial test tool. The embedded test tools will be developed internally. Time at EMI and UL facilities will be rented.	**Deliverables:** Test plan Test cases & procedures Bug list Test Reports

Appendix 3 - QA Plan

1. Identifier

Leisure Time 2000 QA Plan

2. Introduction

3. Standards & plans

This product must meet the following standards:

- FCC EMI standards
- UL standards
- Bluetooth standards

The following plans will be part of this product:

- Data sheet specifying the initial concept (complete)
- Requirements document
- Project management plan (living document that includes schedule and open issues)
- QA plan
- Test plan
- Manufacturing procedures and tests

4. Reviews and audits

Project reviews are held at the end of the requirements, architecture, design and release to manufacturing project phases. Design reviews are held for interfaces between components developed by engineers and areas determined to be a high risk. Changes are reviewed prior to implementing.

Formal audits aren't planned for this project. Project management is responsible for maintaining an atmosphere of professional procedures. Specifications, reviews and configuration management artifacts will be check on a random basis.

5. Configuration management

Item	**Approve Changes**	**Notify of Changes**
Requirements	Cross functional team	Entire team
Specifications	Engineering project management	Entire team
PC Software	Software developers	Test team
Embedded Firmware	Software developers	Test team and electrical developers
CPLD Firmware	Electrical developers	Software and test team
Electrical Circuits	Electrical developers	Software and test team
Mechanical Parts	Mechanical developers	Mechanical developers, manufacturing engineers

All software, embedded firmware, CPLD logic and circuit boards report their version to allow for dynamic configuration.

The design team maintains a compatibility matrix including all released software, CPLD logic and circuit boards. The test team verifies the compatibility though regression tests.

6. Problem reporting

All mechanical, electrical, software and requirements issues will be tracked via the bug tracking system. Each discipline has its own category. Each category will have the same fields to avoid data loss when issues are reclassified.

7. Records

Specifications, plans, minutes of all reviews and test reports are maintained on the project web site. These documents will be archived at the end of the project.

Appendix 4 - Test Plan

1. Identifier

Leisure Time 2000 Test Plan

2. Introduction

2.1 Purpose

This plan covers the system and integration testing done on the Leisure Time 2000 project.

2.2 Scope

This plan covers the system, software, electrical and mechanical testing done after debug and prototype tests, and before the release to manufacturing.

3. Test Items

- The software on the host computer
- The software in the handheld device
- The electronics in the handheld device
- The mechanical package for the device
- The shipping container for the device

4. Features to be Tested

- Downloading text and music to the host
- Downloading of text and music to the device
- Playing Music
- Displaying of text
- Multiple concurrent functions
- Compatibility with web content providers
- Compatibility with host Bluetooth cards
- Drop test
- Shipping packaging
- Cold temperature storage and operation
- Warm temperature operation
- Power management
- Data retention
- UL compliance
- EMI emissions and susceptibility

5. Features Not to be Tested

- Compatibility with a limited number of web sites will be tested.
- Compatibility with a limited number of Bluetooth devices for interfacing with the host computer will be tested.

6. Approach

6.1 Design and Code Reviews

Design and code reviews will be help to eliminate defects before the testing phase and defects that can't be detected by test. The reviews focus on:

- Requirements for future expandability in languages and host platforms
- Areas identified as high risk
- Requirements for future expandability

6.2 Download Text and Music to Host Functional Test

Automated test cases will download text and music from a test server. These tests will cover the downloading of files, deleting existing files and the related host GUI. The GUI test includes cases for handling invalid user input. A commercial tool will be used for the automation. These tests will be run on Windows 98, ME, NT and 2000.

6.3 Download Text and Music to Device Functional Test

Automated test cases will download text and music from the host. These tests will cover the selecting from multiple hosts, host supporting multiple devices, downloading of files, deleting existing files and related device GUI. The GUI test includes cases for handling invalid user input. An internally developed tool will be used for the automation.

6.4 Play Music Test

A manual test will verify the selection, playing of music and the related device GUI. The GUI test includes cases for handling invalid user input.

6.5 Display Text Test

A manual test will verify the selection, displaying of text and the related device GUI. The GUI test includes cases for handling invalid user input.

6.6 Stress Test

A manual test will display text while playing music with a download occurring in the background.

6.7 Compatibility with Web Content Provides Test

Only the sites identified by marketing as important will be tested. This list will be divided into gold and silver sites. For the silver sites, one music and one text file will be downloaded. For the gold sites, each set of music and text will be downloaded. An agreement with each of the gold sites will be made for retesting when changes are made to their site. Because web sites are expected to change frequently, these will be manual tests.

6.8 Compatibility with Host Bluetooth Cards Test

Only the cards identified by marketing as important will be tested. This list will be divided into gold and silver cards. For the silver cards, one music and one text file will be downloaded. For the gold cards, all user accessible functionality will be tested. Manufacturing features will only be tested with the selected card. There will be no testing of for changes in the Bluetooth card. This will be automated with an internally developed tool.

6.9 Drop Test

The unit will be dropped three times on each corner and each side onto a hard surface from three feet.

6.10 Packaging Test

The unit will be dropped three times on each corner and each side onto a hard surface from six feet when in its individual shipping container. The unit when packed in its container and a case will be subjected to a weight equal to being placed at the bottom of a stack of cases.

6.11 Cold Temperature Test

The device will be store at its minimum storage temperature. It will be warmed to its minimum operating temperature. It will be operated at its minimum temperature.

6.12 Warm Temperature Test

The device will be operated for four hours at its maximum operating ambient temperature.

6.13 Power Test

The power test is semi-automated. After new batteries are installed, an automated test script will record the power level displayed, when the replace battery test is displayed and when the system quits functioning. After the batteries are replaced, another automated test script will verify the nonvolatile data. This will be automated with an internally developed tool.

6.14 Data Retention Test

A manual test will verify the volatile data is retained for a sufficient amount of time when the system is unpowered.

6.15 UL Tests

For UL certification, the device must pass the same tests as a handheld computer. These tests will be done at a UL approved lab.

6.16 EMI Emission and Susceptibility Tests

Testing of EMI emissions must be done at an FCC certified lab. Susceptibility testing will be done at the same labs.

6.17 Bluetooth Compliance Tests

The device is tested with existing Bluetooth devices and purchased test equipment for compliance to the Bluetooth standard.

7. Item pass/fail criteria

Each test has defined pass/fail criteria. External standards must be completely met or waiver must be aquired. Internal requirements are evaluated for customer impact by a team that includes Engineering, Marketing, Quality Assurance and Management.

8. Suspension criteria and Resumption requirements

Testing is suspended if the failures that have occurred prevent additional, meaningful data to be collected. Testing resumes when the defects have been corrected and a version has been released to the test group.

9. Test Deliverables

- Test plan
- Test cases

- Test procedures
- Test scripts for the host tests
- Test scripts for the device tests
- Test tool for the device tests
- Test logs
- Defect list
- Test report

10. Testing Tasks

- Create test cases and procedures listed in the approach
- Create a tool for automating embedded tests
- Create test scripts for both host and embedded tests
- Perform the tests
- Reserve external test labs and schedule necessary equipment
- Report results in the tracking system and test reports
- Test lead (This person will also handle QA duties.)

11. Equipment Needs

- Server for text and music
- PC's with all the supported Bluetooth Cards
- Commercial test automation tool
- Internally developed test automation tool
- Rent time and certified EMI and UL labs
- Thermal chamber
- RF signal generator
- Lab large enough to separate host and device

12. Schedule

The schedule is part of the project management plan.

13. Risks / Contingencies

Risk	Contingency
The project includes both contains both Windows and embedded software. The host software must interact with web sites.	We will have a test engineer with web & Windows experience, and a test engineer with embedded experience.
The project must meet FCC and UL standards.	A test engineer with FCC and UL experience will be part of the project. This engineer will work with electrical and mechanical engineers in the design process.
The product will be shown at a trade show before testing is complete.	Tests will be prioritized so tests important to do showing the prototype are complete. The test team will brief marketing regarding features to avoid showing at the show.
Bluetooth is a new standard.	Test equipment specifically for Bluetooth will be used in development and testing. Existing Bluetooth products will be used for compatibility testing.

QA Labs

Quality Assurance Consulting & Testing

Quick Wins: CM in Web Time

Configuration Management for Testers Working in Web Development Environments

Andrea MacIntosh

Senior and Founding Partner

Abstract

Configuration Management, or "CM", long a staple activity for large "traditional" software engineering projects, has been markedly absent from most Web Development projects. CM principles are applicable to software, hardware, documents and other project artifacts, even entire divisions or corporations. Most web application development environments move at such a fast pace that adopting and implementing CM functions over the life cycle of a project is extremely difficult.

This paper (which accompanies a related presentation) will give a brief overview into Configuration Management from a Quality Assurance Consultant's perspective. We'll look at why CM is important, the costs and benefits of implementing CM, the roles and responsibilities associated with CM tasks, as well as some of the reasons behind the lack of CM in many Web Development environments. However, the focus of the presentation will be demonstrating some quick and simple ways testers can add CM to their Web Development environment.

About the Author

Andrea MacIntosh is one of the founding partners of QA Labs Inc., where she focuses on Quality processes, training and consulting. Her areas of technical expertise include Java, XML, web applications, Macintosh operating systems, file formats, and ISO 9000 series implementation. Previously she was the Director of Quality Assurance for Paradigm Development Corporation, which worked on projects for Microsoft, Adobe, Corel, Javasoft (Sun Microsystems), and Inso. Ms. MacIntosh's formal education is in the field of Physics (UBC) where she had the opportunity to perform research in the area of non-linear dynamics of water turbulence and human cardiac patterns.

#200 – 1224 Hamilton Street

Vancouver, BC, Canada V6B 2S8

Table of Contents

Introduction

What is "CM"?

"CM" is short hand for the term "Configuration Management". The SEI lists the following as a definition for configuration management:

> "Configuration Management (CM) is a collection of policies and tools to monitor and assist the development and maintenance of a software system."

ISO 9004 Part 7 gives a similar definition:

> "a management discipline that is applied over the life cycle of a product to provide visibility and control of the product's functional and physical characteristics."

The goal of implementing CM practices is to reduce risk on projects by (1) improving communication between project members, (2) increasing visibility of project elements, and (3) increasing control over project elements. It sounds simple: "let's make sure we're all using the same set of requirements." Yet in reality, I have seen projects that used **four** different sets of requirements, yielding four functional components that had distinctly different user interfaces and a combined set of functional requirements that none of the four components met.

CM is for DOD/Mission Critical Projects, Right?

Nope. CM is for *every* project. The question becomes how much CM do you implement on your project? This question is just as difficult to answer as CM can be to implement. That's why it's important to pick one or more areas in which to implement CM practices, where you believe the greatest improvements can be made.

CM principles are applicable to software, hardware, documents and other project artifacts, even entire divisions or corporations. Most web application development environments move at such a fast pace that adopting and implementing CM functions over the life cycle of a project is extremely difficult. So I'm going to try and focus on the areas that I have found CM activities to be the most useful for web development environments.

Why Do I Need CM?

OK, so you don't think you need formalized CM, you can always ship software without it. On some level, you are right: you can still ship software without having formalized CM processes. But there are huge benefits and cost savings to be had by implementing some of the simplest CM activities and principles into your software development lifecycle.

Benefits of Implementing CM

CM activities can provide many benefits to a project team. The majority of these benefits can be measured and equated directly or indirectly to a savings in effort expenditures, particularly in effort spent doing rework. Here are some of the key benefits that CM can bring to your project:

- Increased visibility into projects

Remember when you were young (or maybe you still do this, I know I do) and you went to the beach. You could spend hours turning over rocks and seeing what kinds of creatures you could find under those rocks. Using CM in your projects is kind of like this. You are looking for ways to see under the rocks, into the caves and crevices of the project, where all the strange creatures go to hide away from prying eyes. Understanding what is going on in your project will allow you to better communicate an accurate picture of the current state of the project.

- Increased control over projects

Just like turning over the rocks on the beach, once you've found the little creatures under there, you can pick them up and hold them in your hand (or, for the squeamish folk, prod it with a stick from a distance). You had pretty absolute control over this thing you had exposed. The same thing applies to your software project. How can you control your budget if you have no idea how much money you are spending? How can you assess the quality of your software if you aren't tracking your testing efforts and results? Once you can see what is happening, how resources are used, how time and money is spent, you can apply some form of control to them.

- Improved communications within project teams and with non-project personnel

So, now that you can clearly see what's under that rock, you can tell your little brother or sister who is down the beach looking at other things just what it is that you've found. This is analogous to telling your other team members and members of other teams what is happening (or not) in your project. Sharing project information can be tricky when your project team is broken up into groups in many different locations, such as a contractor. You want to ensure that communications with project groups in other locations including contractors are clear and controlled, that all parties are receiving the same information at (more or less) the same time. This can be critical for web development environments, where the time to release pressures are huge. Development staff need to convey accurate information to testing staff, and both need to convey accurate information to management and marketing staff. Part of ensuring accuracy is ensuring that all personnel are speaking the same language. CM can help you standardize your project so that all personnel are speaking the same one language.

- Stop losing things that are key to your project

This is always one area that baffles me. In most software development projects, things get "forgotten" or left by the wayside for more pressing matters. It's a case of "the squeaky wheel gets the grease", and it happens all the time. Going back to our rock-flipping at the beach analogy, this would be a situation where you find more than one neat looking creature as you flip a rock over. However, one is perhaps faster moving, and might get away, so you grab it first. While you examine your new prize, the other creature could have slipped away, back under an adjacent rock. Sometimes, it's OK to lose some things, they are of relatively little importance compared to other items. However, there are some things that are costly to lose, such as time lost to unneeded rework or lost defect fixes. CM process improvements can help you keep track of your artifacts and items, so you don't lose them anymore. This can have tremendous cost savings, as lost items usually spells "r-e-w-o-r-k", which costs money and time.

CM's goal is to **IMPROVE QUALITY** by improving control over various aspects of your projects, your departments, and your resources. As you increase the visibility into your projects and your processes, they are easier to identify, to quantify, to assess, and thus to control. However, that being said (or, more correctly, written), there are costs and difficulties with CM process implementations.

Costs of Implementing CM

- People Costs

To implement CM, you need someone who does this stuff for a living, someone who is experienced in process improvement **OR** a well-rounded team. I prefer the team approach, because you can divide the CM labor amongst the team members, and accomplish quite a lot in a short period of time. As well, the greater number of voices you have repeating the same mantra will slowly but surely have an influence on the rest of the project team.

You will also need a high-level/upper management champion, a person who buys into SQA in general and CM in specific. This person can help with introducing new processes by leading the way within the upper management group. This person will help get you the budget and resources you need to implement any process improvements.

Regardless of the route you choose, there will be an associated cost in terms of what percentage of your efforts and/or your team members' efforts are spent on CM activities. If you spend 10% of your workable hours for CM, then approximately 10% of the real cost of your employment will have been spent by your

organization for CM implementation work by you (note that the real cost of employment of a resource is usually double the full-time salary cost of that resource).

- Tool Costs

One of the first things you'll find you need for implementing CM is a version control tool, and possibly a document control/management tool. Whatever tool or tools you end up using may have a price tag that includes the license, associated resources and documentation, and a support contract. You may also be able to receive training in the use of the tool from the vendor.

- Training Costs

People need training time and follow-up time, both in the use of tools and techniques, and in learning and understanding the role and importance of CM as it relates to software engineering. You and your team may need training courses, materials, and possibly additional supporting resources (such as a consultant). While spending time being trained, resources are not actively performing project work. So any training time may be equated to an indirect cost in terms of lost productivity.

- Space and Equipment Costs

To implement some CM processes, you may have to make a substantial investment in hardware and software, from adding hard drive space on common servers to replicating a complete deployment environment. Of course you'll need to have physical space for any additional machines, and for any additional team members. This can be a substantial cost for many web development environments, as some of the hardware and software required for these CM implementations are quite costly.

- Quality Costs

Remember, many different companies define quality in many ways, but to the end user, quality always means, "Am I satisfied?" If we define quality as perceived value to the user over cost to the user, then web applications, which have no real cost to the user, should move towards infinitely greater quality. In fact, as the cost goes down and the marketplace becomes saturated with more and more sites that offer similar transaction functionality, and as these sites become increasingly complex, we see the opposite occur. Decreasing quality of software, for both desktop applications and web applications, is something that has been occurring for quite some time now, and the sad truth is that end users have been tolerating (if not encouraging) this trend.

I believe that market forces will cause and already have begun this trend to reverse. As the Internet becomes swamped with dot.coms competing against each other, the limited attention span of users will reward only those sites that do not disappoint the user. This is the key argument for increasing pressure towards better testing of web applications: with traditional software a user has spent a certain amount of money and hence feels motivated to get the best utility from his or her investment. The alternative would only be to buy a different solution (which implies more expenditure, a new learning curve, incompatibilities etc.). A web user does not have such a difficult choice. If she uses an airline reservation system that does not behave according to expectations, she can click through to another service provider and see if she likes that environment better. The switching costs for most web applications are so low, that users will simply browse away from sites that exhibit poor usability. (Refer to Jakob Nielsen's and Donald A. Norman's article for *InformationWeek Online* entitled "Usability On The Web Isn't A Luxury" at http://www.informationweek.com/773/web.htm, or refer to Dr. Neilsen's website, http://www.useit.com, for more information on usability in general.)

Difficulties in Implementing CM in Web Development Environments

- The iterative nature of Web development

Releasing an initial version then adding pieces or revamping the UI over time (the "whale-dolphin" release approach) adds a new dimension to adopting CM for web development environments. a whale/dolphin model. The whale is the initial hump of effort to release the first version of the web app and the dolphins are subsequent cycles of functionality increments. These subsequent cycles continue throughout the life of the web application (see Figure 1 below). Because releases are small and regular, it should be easy to convince yourself that any process changes you implement should fit well with this model. Process improvement activities that work well with the whale-dolphin model are usually small changes that can be measured over a short period of time. With small changes in each release cycle, you can evaluate the effectiveness of the process change in a short time (without having to wait for a long cycle to be completed).

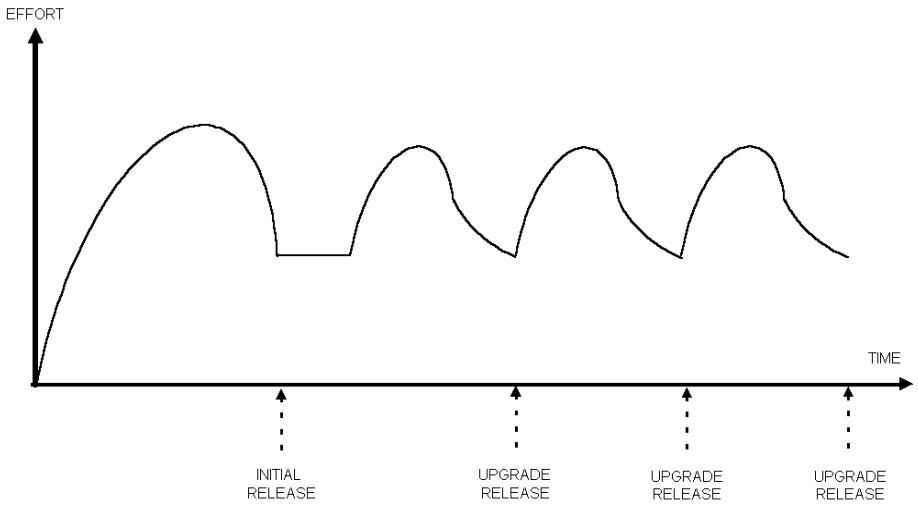


Figure 1: The "whale-dolphin" release cycle of web development projects

- The Time-To-Market pressures.

Short schedules do not allow for implementation, training, audits and follow-up to implementation of CM initiatives. CM is seen as "stuff that slows down the development process", makes it harder for people to "go live" or meet investor deadlines.

- Not the right people.

Many dot.coms and Internet development companies lack the right resource to take on these kinds of tasks, and they also often lack of senior management support for such initiatives. They are often seen as a line item in a budget that can be easily cut. After all, you don't need CM to produce code, right?

- Bad habits

We all know how hard it can be to break bad habits. Web development in general is full of many bad habits, such as not using version control for source code. So if web development environments are full of many bad habits, and it will be hard to break a bad habit, then it will be very hard to break many bad habits.

What CM Won't Give You

CM will not help give you a way to change developer attitudes about testing and QA, in general. Because it is more likely than not that QA resources will be the resources doing the CM activities, it is important that you make an effort to show the distinction between QA and CM. Although they are very interrelated, they are distinct activities and processes. Most project staff are somewhat mystified by what QA is, and what QA resources do on a project (aside from testing), so to further muddy the waters with CM can be misleading.

As well, as there are many development project personnel who do not believe that independent QA and testing is a good thing (as opposed to simply having the development staff test their own work), implementing any CM processes is not going to help change that kind of a mind set. CM will also not give you a means to hide staffing problems such as weak or under-performing resources, or the need for major amounts of rework due to poor or shoddy work.

What CM Supports

CM activities help support certain kinds of software engineering activities. In many cases, without CM processes, many of the following kinds of activities become increasingly difficult, if not logistically impossible to do:

- Parallel Development allowing for developing/maintaining several versions of a product in parallel.
- Distributed Engineering across LANs and WANs including data replication and shadowing facilities.
- Building and releasing product versions to customers as well as re-creation of earlier product versions.
- Change Management to ensure that products and their elements have adequate measure to cover planning, tracking, authorizing and controlling of changes.
- Configuration and Workspace Management, including specifying configurations, such as testing and support environments (e.g. compilers, debuggers).
- Process Management which addresses defining the process lifecycles for the product elements, setting up appropriate change controls, and authorization policies for product element modifications.
- Object and Repository Management to facilitate the storage of product elements with version management and branching and their meta data information.

For Web Development environments, some of the above activities are not critical. For example, object and repository management is probably not a huge concern, but you can bet that distributed engineering, building and releasing, and workspace and configuration management sure are key activities. To ignore the importance of these activities will surely lead to peril (or significant rework and schedule slippage – which in the Internet market – certainly is perilous).

One particular example is that of distributed engineering. Many companies will outsource some of their development, particularly when a schedule is tight. Web development projects have several components that may be outsourced, including art and design, development and coding, testing, and deployment and hosting. CM helps organizations that outsource part of their projects track precisely what was delivered from contractors and outsourcers, when those deliverables were delivered, as well as storing those deliverables in an appropriate manner.

Another example that affects web development projects perhaps more distinctly than traditional shrink-wrap or client-server development projects is the dependencies upon third party software such as plug-ins, platforms, and clients. Most web applications today make heavy use of licensing and wrapping 3rd party functionality, such as application server software. This is usually done to reduce time to market and overall development costs. However, because of this heavy dependency on third party products, web development organizations must keep close track of product versions and defects or vulnerabilities in those product versions. CM will help organizations do this.

CM Focus Areas

There are four key areas of CM: Identification, Documentation, Control, and Auditing.

Identification

A core activity of CM is identification of artifacts. You need to develop the means and ability to quickly, consistently, uniquely and easily identify artifacts. Common identification techniques include filename conventions, version numbering conventions, and standardized meta-information ("information about information").

Documentation

Part of CM documentation activities involves documenting your processes, even the informal ones, such as "Bob always does the builds on Monday morning" or "we store our builds on server X". Some examination of your current software development practices will reveal that you and your team are probably doing some CM activities without calling them what they are. I like to call these kinds of informal processes "project lore" – they are the unwritten "rules" of the project.

The other part of this activity involves documenting the artifacts that have been uniquely identified (above). This kind of information, or "meta-information" (information about information) can help project personnel find what they need more easily, as it allows them to determine for themselves if they are reading the correct or applicable document, or have a copy of the most recent requirements, for example.

Control

Another core activity area is that of control. You need to control all the project artifacts, from documents to test records, from defects to requirements, from versions of third party software to hardware components for your deployment environment. Controlling processes are usually difficult to implement effectively in any software development environment, but are more difficult to implement in a web development environment. Common control mechanisms include source version control tools and defect tracking tools.

Audit

Auditing is the final piece of the CM puzzle. Once you have some controls in place, you will want to measure the effectiveness of those measures. The best way to do that is by use of audits. Audits can be simple or complex, small or large, high-level or very detailed, but all can be used to pinpoint specific problems (or issues that can become more serious problems down the road) while providing concrete data or evidence of the scope and magnitude of these problems.

What CM Should Look Like

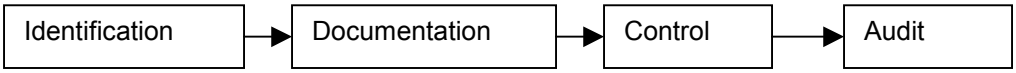


Figure 2: The four aspects of CM for traditional development projects

You first begin by identifying things or "artifacts", like documents, source code, bugs, anything you use on a project, in some unique manner. That is, each item will be uniquely identified from other items, perhaps with an ID number. Once you have identified these items, you can then document their existence, referring to these items by their unique identifiers. After having documented the items, you will have enough information about the items themselves to begin to apply basic forms of controls over the items and changes to the items. Finally, to ensure that the items are being identified, documented, and controlled according to your standards, you can audit the items against the standards you have created.

What CM for Web Development Environments Often Looks Like

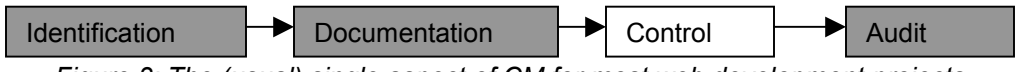


Figure 3: The (usual) single aspect of CM for most web development projects

Because most web development environments work with little to no documentation and with even less time, the majority of efforts are spent trying to control the environment and the situation. However, the

team often doesn't know what to control, or if the attempts are in any way effective. Since items have not been identified, they cannot be effectively documented, and you cannot perform audits to verify that the controls you have put in place are in fact working, or how effectively they are working.

What is a "Quick Win"?

A "Quick Win" is a small process improvement, something that is small in terms of size of effort required for implementation. The idea behind Quick Wins is that not all teams can swallow large process changes, but almost everyone can adapt to small changes that make their jobs slightly easier.

The key points when thinking about Quick Wins are:

- Don't try and solve all your problems at once. Start with one or two improvements and review them at the end of the project. When compiled together, can effect huge improvements.

- Remember that Quick Wins should be easy to implement. Small cost- and time-effective changes are going to be easier than large changes, such as requiring an entire team to adopt a new methodology or new suite of tools.

- You want a Quick Win to be something that could be done by a project resource as part of their regular project work. Less than 10% of a resource's time should be spent on Quick Wins.

- Most Web Application projects are short in duration (1-6 months), so Quick Wins need to be of the same time-frame. Quick Wins allow you and your team to gradually improve your processes over the duration of the project, but you also want to be able to measure the effectiveness of the Quick Win at the end of the project.

When I begin a consulting stint at any organization, I try to examine the current processes in place with the idea of finding and implementing one or more Quick Wins almost immediately. I have found that because I am an external person who has worked in many different development environments, I can often see possible solutions to problems that team members who are close to the problem cannot.

Identification

File Name Conventions

One project that a contract development company was creating for a client was delivered with all source file names pre-pended with the project code name (something geek-chic, like "ScoobyDoo" or "Tardis"). Once delivered to the client and the flaw exposed, the files had to all be renamed, as well as all the references and packaging. This was a terribly messy situation, which caused lots of unnecessary rework, and the contract development company ended losing money on that engagement.

A similar case occurred when the company changed the web company name, oh, four times in as many months. This caused a major amount of rework each time the name was changed. But laziness paid off: the programmers didn't make the changes required for the third name change, suspecting another one was coming soon. And sure enough, the fourth one came up a week later! They saved at least a person-week that would have been otherwise wasted.

The Quick Win Approach:

NEVER use your project's name (or code name) as a part of your file name. This is especially important for source code – makes reuse and/or reconfiguration a messy task. You can do it for project documentation, but you need to be consistent. Make sure that the spaces or no spaces issue and the case sensitivity issue are resolved (whether or not spaces are allowed in filenames, whether or not filenames are case sensitive or not).

Try to avoid putting version number information in the file name as well. This kind of information should be inside the document – or handled by a document version control tool. Omitting version number information from the filename makes intranet linking far far simpler. Updating the documents and the links are handled by the repository, not by changing the links themselves.

Version Numbering

Having worked on a project where the testing team was testing against outdated requirements, simply because the requirements document did not have any information as to the version number (or that the document had even been updated!). The test team logged lots of defects against their copy of the requirements, the development team kept closing the defects as "Not Repro" or "Not a bug", and the test team kept re-opening them as "Oh, yes, this is a bug alright!".

This was incredibly frustrating for the entire team. This went on for a solid week before one tester and one developer were discussing a specific defect. The tester referenced the specific requirement, to which the developer replied "that's not a requirement anymore". The two examined their copies and found that there were significant differences between the two, and that the development team was using the latest, up to date requirements.

Because there was no version information within the document (or at least within the filename) and the document was not under version control, all of the team members were using different variations of the real requirements. I can attest that the additional cost of re-testing against the updated requirements – once they were distributed to the testing team – was very high. This particular project ended up going live two weeks late, simply because of the confusion and subsequent retesting required.

The Quick Win Approach:

Documents should be given appropriate version numbers. I suggest trying this scheme:

"<version>.<revision>DRAFT"

where **version** runs from 0 to n, and **revision** from 1 to [1]0. So a first draft of a document would be "0.1DRAFT". All incremental revisions of that draft document triggers an increment in the revision number. Once a document is "accepted" or "signed off" (no longer a work in progress), it graduates to a "1.0" version. After that point, it should be strictly controlled. Minor changes trigger a revision number increase, major changes trigger a version number increase. Major changes should be approved by stakeholders. But that's getting into change requests/change control boards, a much larger topic for another day…

If you think that might be too difficult to implement, just stick with a **"<version>"** numbering scheme, where version increments from 0 to n. It's always better to start with small process improvements, small changes that your team will accept, and slowly change the process over time than to try and force a more drastic change all at once.

Documentation

Ever come onto a team half-way through the project?

Remember how difficult it was to find information about how the project team did their work?

- Where builds were stored on the server?
- Who was responsible for what?
- What kinds of coding standards were in place?

Whatever CM quick wins you decide to implement for your project, they should be written down. Make sure these "policies" are somewhere accessible by all project members. Aside from these project policies, finding project documentation (especially up to date documentation) can be incredibly difficult.

The Quick Win Approach:

Set-up a Project Intranet/Company Intranet/Department Intranet. Using a version control tool, create a separate repository (or root directory) for project documents. Documents should always be kept separate from source (refer to the section on controls below for more details on this statement). Then shadow these documents to a central web server. By linking them all together via a simple (one page) index web page, you will allow people to find new and updated information themselves, and add visibility into the project (this is great for new people).

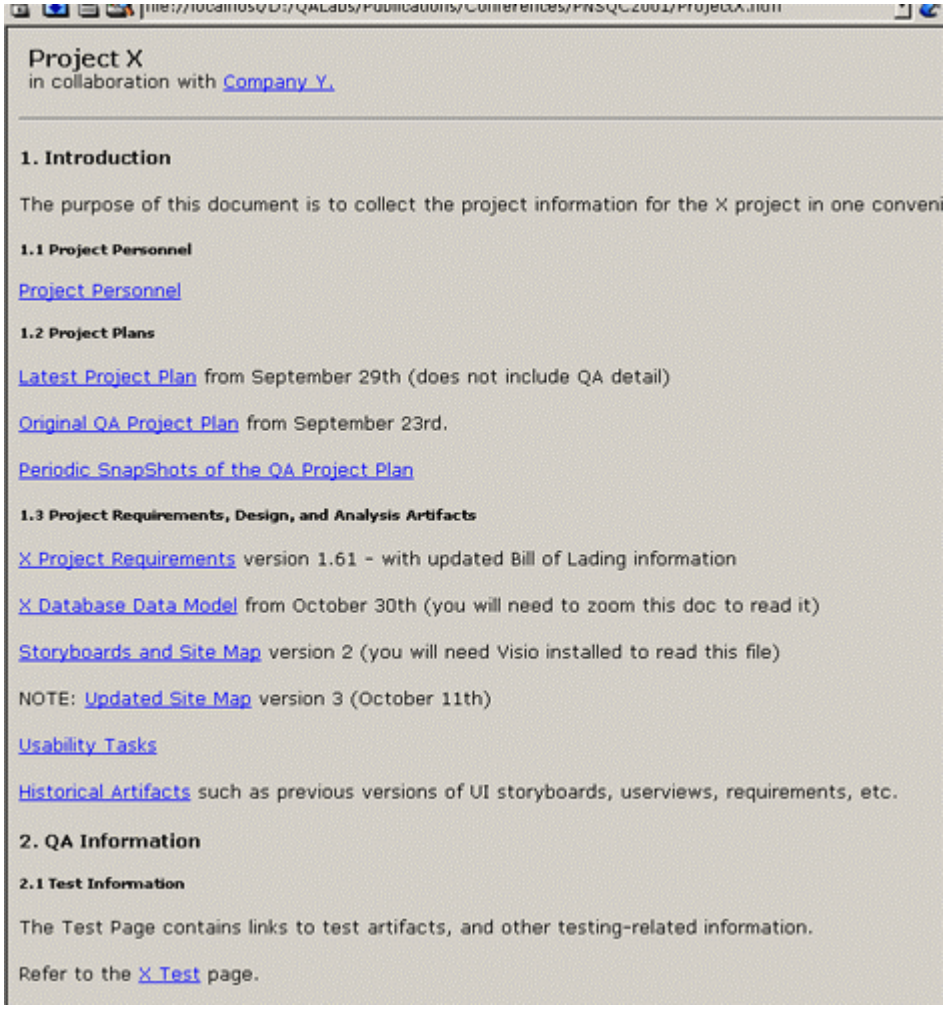


Figure 4: Sample Intranet Index Page.

This is a very easy file to create and maintain. Because you have placed all of your documents in a repository, and have shadowed them to a central web server (even a file server will suffice), you merely have to update this single file, adding, removing and editing links as required. If you can begin this page at the beginning of the project, then you will find that maintenance of it is not overly time consuming (you would spend no more than 5% of your weekly effort updating this sort of a file).

Even if the majority of your project team doesn't seem open to having a project intranet, it can still be valuable for the QA and test team for the project to have their own. Sometimes, the best way to get people to try new things is to show them how easy it can be.

I recall one project where our QA team for this project set-up our own document intranet at the beginning, solely for QA to use, and didn't tell the development staff about it, thinking that they wouldn't be interested (as we really only had linked up the QA Plan, test plans, test cases, and the defect database). Well, the development lead saw it on my screen one day, asked about it, and immediately started contributing to it! He began to link in development-centric documents, such as coding standards, functional specifications, and architecture diagrams. Before long, all of the development staff were linking in documents, and were using it as their reference point for the entire project.

Control

Document Control

My all time favorite horror story involved document control, or more precisely, the lack of document control. One project team was tracking all of their defects using a custom Outlook form and public folders, all stored on a central mail server. This public folder was set-up so that the project manager and the development lead could delete posted items, which they decided was a good way to "remove" closed and fixed defects from the list. They wanted to keep the list as accurate as possible *in displaying open or resolved defects only.* As defects were fixed or addressed, retested, then closed, the project manager then deleted the defect record from the folder. If you are a tester/QA person, you are probably cringing right about now. You can imagine how effective the test team's attempt at regression testing was, and how much useful information about bug counts, bug severities and so forth was gathered from the defect list throughout the project.

So controlling documents is important because we need to have a history of what we have done on our project to date, what decisions were made and why, as well as what plans we hope to implement in order to achieve specific goals. If these documents are not controlled, and get "lost", deleted, corrupted, become unusable or unreadable in any way, then the project team cannot work cohesively as a team towards common goals in a common way.

The Quick Win Approach:

You should create a separate repository for project documents; documents should always be kept separate from source. Why?

- Maybe the back-up schedules are different (and they probably should be) – source code is usually backed up more often than documents.

- If one goes down or becomes corrupted, you can restore it from the last back-up. It is always easier to restore part of a repository (and indeed part of a project) instead of the whole ball of wax. Documents are usually in binary format (such as Word) and these binary document files can be hard to recover from a previous version (or at least it can be significantly more difficult to isolate the differences between two versions of a binary document file).

Some kinds of documents that should be controlled include:

- Project Plans

- Requirements

- Design or Architecture Documents

- Test Records

- Test Assets
- User Documentation

One example is that of test records, which are the pieces of anecdotal evidence that prove that testing was performed and that the results were captured. Test records include:

- Defect reports
- Test case execution reports
- Defect status reports/metrics

In the case of defect reports, defects **must** be kept in a database that is accessible by all project personnel. Some sort of tracking tool has to be implemented. Email, Excel spreadsheets, post-it notes, Exchange public folders will not suffice. Why? Defects occur over time, and your codebase will change over time. It is incredibly difficult to make an assessment of the current state of your web application if you cannot determine what defects are currently active in the current build as well as defects that have already been addressed. As well, you will want to track productivity of both the testing group and the development group. Using a database will allow you to collect measurable productivity data. Use a FREE or low-cost defect tracking tool that has web access. Check the "CM Yellow Pages" website for a good listing of these tools.

You should have one person responsible for reviewing and managing the defects. This entails reviewing new defects, ensuring defect reports are complete and unique (non-duplicates), and initial assignment of the defect to a resource. I would suggest that the QA Lead or the Project Manager for the project does this. You may want to have an another person responsible for prioritizing the defects, someone who talks to all the stakeholders. This may be done by QA, but usually by someone who has a good relationship with the client or the end-user.

Source Version Control

Ever have one of your developer's hard drives fail and lose weeks of work? Ever tried to replicate exactly what code was on the live server prior to a hard disk failure? Ever tried to roll back a change that was buggy? With web development environments, rolling back changes is very hard unless you are using a version control tool. In most cases when changes are made, there is no going back only forward. So if a serious defect is found in new code, rolling back a production environment to a previous version is rarely done. Instead, the project team scrambles to fix the current problem, quickly test and upload the new version.

Source version control gives you a means for controlling releases to live servers, a means for controlling integration (or removal) of new code and defect fixes, a place for all programmers to put their code and a place from which QA can do builds.

The Quick Win Approach:

Go Cheap and Easy. Use whatever tool(s) you own and whatever your people know how to use:

- VSS
- CVS
- RMS

However, don't go cheap and hard, meaning that think long and hard before you sacrifice the cost of a version control tool for the additional simplicity and usability of a perhaps more expensive tool. Make certain any remote personnel can access it, but with some restrictions in place (administrator privileges and super-user commands should be off-limits to most of the team).

You should get into the habit of archiving your source (and any compiled binaries) onto a separate server on a regular basis, daily if you can. Why?

- This allows recovery if your repository becomes corrupted or if a back-up of the repository was unsuccessful.

- This allows QA to quickly restore to a previous build, and continue testing, if a new build is "broken" or largely untestable.

- This allows you to quickly re-deploy a solidly tested release in case you run into problems with either your deployed version or the configurations themselves.

Testing Environment

I recall one project where the development and the testing team had only one environment for both groups to share. So developers were uploading updated files "on the fly" throughout the day, and the testers were watching as defects appeared, then disappeared, then re-appeared but slightly different, and so forth. The developers were constantly closing the tester's defects as "Not Repro", the testers were finding defects that would suddenly disappear or they were finding that the testability of the web application itself was so variable and unreliable ("hey, where's the login screen gone?") as to make the entire web application untestable.

Yet, this is a common scenario in the web development community. And while making testing virtually impossible, it is not yet dangerous to the end user, the customer, until the application goes live. Once it is live, and actual users can access the application, this style of migrating changes is highly likely to cause serious defects.

Your test team shouldn't crash your site -- they should crash a separate test site. Testing a live site does not allow for "bad user" testing or nasty security/reliability testing. Your test environment should mimic the planned deployment environment as closely as possible. However, this is often difficult because of hardware and software costs.

The Quick Win Approach:

I recommend a minimum set-up of three (3) environments:

- A set of development servers used for the development team.

This environment is the "sandbox" for developers to check their code prior to check-in.

- A set of staging servers that are periodically updated with a new release, used as the test platform.

Deployments to this environment should be controlled.

- A set of production servers that hosts the live site.

Notice that only the second environment is the real testing environment, but it is part of the deployment path. These three environments should be as similar as possible. Of course, this isn't always possible, simply due to the amazing cost of some of the hardware and software required for final production/deployment environments. But you should try to mimic as closely as possible the final deployment environment. One exception is the testing environment, where you will often have testing tools and debuggers installed to assist with testing.

The environments themselves must be controlled as well. That means no weird stuff being installed on machines, no changing Java versions, for example, without everyone knowing and agreeing to it beforehand. QA should know exactly what is installed on the testing and production environments, including versions of third party products and plug-ins.

Builds and Deployments

I have been called into projects that have had no control over what was being tested and subsequently released. No one knew exactly what version of what files were being tested, or were being hosted on the production servers. Migrating changes to either the testing or the production environments was a nerve wrecking, nail biting experience, as with each file upload, no one on the team was certain of what the results would be. This is particularly dangerous for live, production environments as uploading an incorrect version of one single file can causes the entire web application to stop functioning correctly (or at all). Many instances of this exact situation occurred, so much so that the deployment team began deploying new code in the middle of the night knowing that was when traffic to their site was at the lowest point (and thus less likely to affect a large number of users).

Doing releases at 3 am does not make sense. There has to be a better, lower-stress way to do this, right? If we set up the three environments listed above (development, testing and production environments), how do we then migrate releases between these environments? In particular how do we deal with migrating to the live production environment?

The Quick Win Approach:

Migration of code between environments should appear to be One-Way only. This means that source code should appear to be migrated from a development environment to a testing environment to a final deployment or production environment. However, the code itself is migrated from the code library to the specific environment, not from one environment directly to another. You shouldn't move actual files from your testing environment directly to your production environment, but from your source control repository (more specifically, from a tag or label within your code repository) to the production environment. This allows you to always recover the exact code base you placed on the environment at any time.

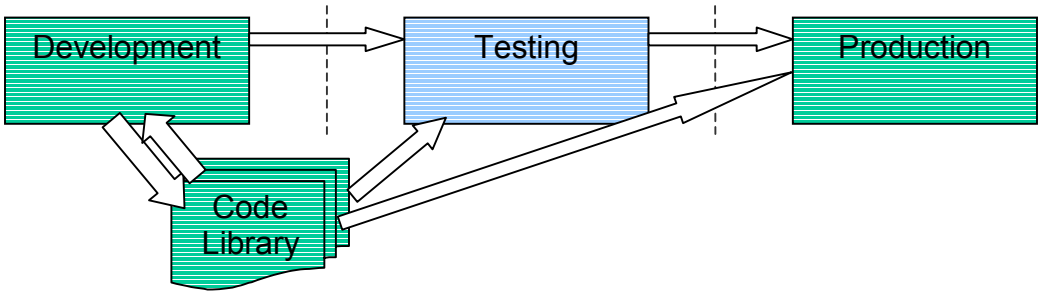


Figure 5: Migration Paths for Web Development Environments

While this model has been used in traditional software development for some time, one key difference when applying this model to web development is the amount of feedback between environments. With more "traditional" software development projects, there may be changes made to source code during a site deployment. These final changes made on-site need to be folded back into the code repository. However with deployment to a controlled environment (controlled in the sense that the end user doesn't control the environment, but your organization maintains control of the environment and access to it), such as for a web application, you can migrate changes far more easily. It doesn't cost you anything (or very little) to release updated code to your production environment while re-releasing traditional software (such as "shrink wrap" software) is very expensive.

Releasing to the web, your controlled environment, is far more forgiving than releasing traditional shrink wrap software because of the ability to quickly correct the problem. As well, migrating any changes from the production environment back to the code repository is very simple. I will list one warning here: because it is so easy to migrate changes and to fix defects on the fly, we have a tendency to get "sloppy" in controlling changes to production environments. Because the possible costs of making an error in deployment and the subsequent fix are small when compared to the same scenario for traditional software projects, it is very common to take a lackadaisical attitude towards builds and releases.

Unfortunately, if your users find a serious error, because of the nature of the web, they will more likely than not seek out an alternate service – your competition!

Migration between environments is should be controlled, in general deployments from the Development environment to the Testing environment should be controlled by Development or by QA. Deployments from the Testing environment to the Production environment should be controlled by QA and/or by project management. Deployments should always be done from a label or tag within the source version control repository. I have also found it helpful to embed the build or release (version) number somewhere in the software. This helps everyone (including external users) quickly identify what version of the software is currently deployed, and thus what version contains specific defects.

Audits and Other Processes

Audits

Have you ever been on a project where everyone says everything's moving along smoothly, there's no slippage, and no one has any questions or any issues? And you find out later (when it's really too late) that there have been serious problems all along? How is it that we allow people to slip on their tasks without taking into account the repercussions?

Case in point: a company had a programmer whose code was so unreadable by humans (undecipherable) that once he quit the company, the company was forced to re-write most of his code (no one knew what it did or how it did it). How was this programmer allowed to code so shoddily for so long?

Case in point: if you create software that may have high liability issues (say, medical or other personal information), you need audits to prove that you made every best effort to not produce buggy or sub-standard software. How can you prove that you have done what was expected of you and your team? How can you ensure that your team is doing what was agreed upon, and in the manner that was selected?

My favorite example of why audits are a good thing is the project where daily tape back-ups were supposed to be made of the source. These tapes were supposed to be taken off-site to ensure that if anything went seriously wrong at the company's site, they would lose no more than one day of project work. The person charged with making the back-ups made the back-ups as directed, once a day. He faithfully took the tapes home at night and brought them back the next day. However, when the code repository server's hard disk failed, and the team tried to restore the previous day's version from the back-up tapes, it was determined that the back-up had failed. It turned out that the person making the back-ups never tested them, never verified that the back-up had completed successfully. And because the team had only one set of tapes that were re-used on a daily basis, he had overwritten the last back-up (which may or may not have been successful as well). The team had to cobble together a new source repository by pulling latest versions of specific files from the team members development and testing machines. They lost over a week's worth of work and had at least another week or rework required to fix the problems caused by this disaster.

If your gut feeling is that something is wrong, it is very likely that something is. Auditing will help you find out what is going sideways, hopefully with enough time to address the problem before it becomes a costly disaster.

Auditing will help verify that:

- Procedures are correct and appropriate
- Test results are being recorded and retained
- Documentation is complete and up to date

- Software meets requirements

- User documentation is complete and accurate

Most software development teams already do a form of auditing: they do one at the end of a project cycle and call it a “Post Mortem”. This is where the team gets together and reviews what worked well on the project, what didn’t, and what needs improvement for the next project cycle. So expanding this idea shouldn’t be too foreign a concept to your team members.

The Quick Win Approach:

First off, don’t call an audit an “audit”. People are afraid of audits, because they think that an audit is designed to catch their mistakes, to try and hunt out bad or under-performing workers. Come up with some other phrase, maybe just calling it a “checklist” will be sufficient. Often if you can implement audit processes in a stealthy manner, you won’t even be noticed. By “stealthy manner” I mean not walking around the office telling everyone about how your checklists are going to show management all the problems on this project, or sending a broadcast e-mail to the entire team asking for feedback on your comprehensive audit program (you should be selective about who you ask for feedback)

Create checklists early on in the project. I have always found it useful to create them before the project gets well underway, because you won’t have time later on. It allows other project personnel a chance to see what will be coming, what QA’s expectations are regarding specific processes, and allow them to give you feedback on your checklists.

You will also want to determine your audit frequency: how often will you check the items on your checklist? I have found it particularly useful to tie audits to milestones, such as phase completion. Milestones or phase completion points are the natural pausing point in a project lifecycle, when all project personnel stop for a second to look up and around. So if you were to do some detective work, some snooping around and asking some questions related to “where are we now?” and “have we done what we said we would, and how we would do it?”, you usually won’t stir up too many people’s ire.

Make sure your checklists have simple “Yes” or “No” type-questions:

- “Was document X stored correctly?”

- “Are defects all stored in the defect tracking system?”

- “Are you using the latest version of the requirements?”

You don’t want to have a checklist that asks vague questions of the auditor. The auditor should be able to easily execute the audit without having to ask for help with each item. This will help you gather consistent results, consistent data that has meaning. It will remove the subjective-ness that can be unknowingly imposed by the auditor (this can occur when different auditors perform the same audit if the audit is loosely defined or constrained).

Spend no more than one hour performing the audit. Test drive your audit one evening after everyone has gone home to see how long it will take to complete your audit. If the audit takes more than one hour, then either 1) break it into two or more audits or 2) reduce the number of items on your checklist. I would strongly recommend you choose option number 2. Why? Because audits can be hard to do in a correct, impartial, and consistent manner. You want to be able to collect meaningful information from these audits, and if you make them too hard to do, you will increase the likelihood that the audit will not be done completely or correctly. Plus, your project managers don’t want you spending any more time doing audits than necessary. If you are spending too much time doing these audits instead of testing new code for release, then you can bet that any budget or schedule for audit activities will be cut.

Have a place in your document repository to store the results. You need to keep track of the results because you want to track these things over time, and over projects. Each time you perform a certain

audit, you should compare your results to the previous time you performed the same (or similar) audit. This will allow you to look for trends, both good and bad ones.

Issues raised or discovered during audits must be followed-up. This may result in considerable re-work or unplanned effort and this unplanned effort needs to be folded into the updated project plan and schedule. Remember: the audit doesn't create the new work/re-work; it merely uncovers the need for it.

Conclusion

- **Keep it quick.** Spend no more than 10% of your time on CM implementation(s).
- **Keep it small.** Make the change easy for everyone to adopt and follow and deal with one problem at a time – don't get distracted by other issues or you'll drop the ball.
- **Keep it focused.** Have a plan and try to stick to it.
- **Iterate, Iterate, Iterate**. Work quick wins into each cycle and make sure at the end of a phase or a project, that you review the CM implementation as part of your post mortem.
- **You are not going to get it (completely) right the first time.** You may not get it all fixed the first time out, but any improvement is better than none.

References

General Configuration Management References

1. Cagan, M. and A. Wright, "Requirements for a Modern Software Configuration Management System," Continuous White Paper, January 1992, pp. 17.
2. Dart, S., "Concepts in Configuration Management Systems," Proceedings of Third International Conference on SCM, Trondheim, Norway, June 12-14, 1991, pp. 18
3. IEEE Std 828-1990, *IEEE Standard for Software Configuration Management Plans*, American National Standards Institute, 1990.
4. IEEE Std 1042-1987, *IEEE Guide to Software Configuration Management*, American National Standards Institute, 1987.
5. ISO 10007:1995 (E), "Quality Management - Guidelines for Configuration Management."
6. Rigg, W., C. Burrows, and P. Ingram*, Ovum Evaluates: Configuration Management Tools*, Ovum Limited, 1995
7. Configuration Management Yellow Pages http://www.cs.colorado.edu/~andre/configuration_management.html
8. Not All Tools Are Created Equal http://www.adtmag.com/pub/oct96/fe1002.htm
9. Software Configuration Management by MIL-STD-498 http://stscols.hill.af.mil/CrossTalk/1996/jun/CMbyMIL.html
10. Branching Patterns for Parallel Software Development http://www.enteract.com/~bradapp/acme/branching/streamed-lines.html
11.

Other Related References

Software Engineering Institute (SEI) SEI's CM pages http://www.sei.cmu.edu/legacy/scm/"The Joel Test", a quick set of 12 questions to ask yourself about your software development processes http://joel.editthispage.com/stories/storyReader$180CM Today, which as links to free CM and defect tracking tools http://www.cmtoday.com/yp/configuration_management.html

A Proven Work Breakdown Structure for Process Improvement Projects

By Tim Avilla and Heather Falk

Process improvement projects must be planned and managed just as any other project. This paper outlines a practical work breakdown structure for a process improvement project and offers a strategy for implementing the process in the organization. The paper defines the products for a process improvement project; outlines the phases, activities, tasks and deliverables for a project; and offers strategies for implementing a new process.

The Authors

Tim Avilla has more than 22 years of experience with the Oregon Department of Transportation (ODOT) and is currently the program manager for ODOT's Process Improvement Program for Information Technology. Tim is an experienced project manager and has led many highly visible projects, including technical implementation of the Oregon digital photo driver license.

Heather Falk provides software configuration management for Oregon Secretary of State with 16 years of experience in project management, software engineering and process improvement. Special interests include software configuration management, process architecture and information systems project management.

Tim and Heather would like to acknowledge the contributions of Julie (Stewart) Mallord for her work and concepts in the initial development of the work breakdown structure for process improvement projects. Julie is currently the manager of the Project Management Office at the Oregon Department of Human Services and chapter president for the Willamette Valley Chapter of the Project Management Institute.

Tim can be contacted at
Oregon Dept of Transportation
ODOT Mill Creek Building
555 13th Street NE, Ste 1
Salem OR 97301-4166
Timothy.C.Avilla@state.or.us
Phone (503) 986-3231
Fax (503) 986-4072

Heather can be contacted at
Oregon Secretary of State
Public Service Building
255 Capitol St NE
Salem, OR 97310
Heather.L.Myers-Falk@state.or.us
Phone (503) 986-2248
Fax (503) 986-2249

Introduction and Background

The Information Systems Section of the Oregon Department of Transportation (ODOT-IS) established the Process Improvement Program in 1998 with the mission of facilitating process improvement initiatives for system development with a long-term goal of obtaining the benefits of a Capability Maturity Model (CMM) Level 3 organization.

This team established a governance structure, completed a formal CMM assessment in December 1999 and outlined a plan of action to improve performance in key processes identified as lacking. Before embarking on a series of process improvement projects, the team recognized the need to establish an infrastructure that could be used, repeated and improved in completing PI projects. That need resulted in the concepts described in this paper.

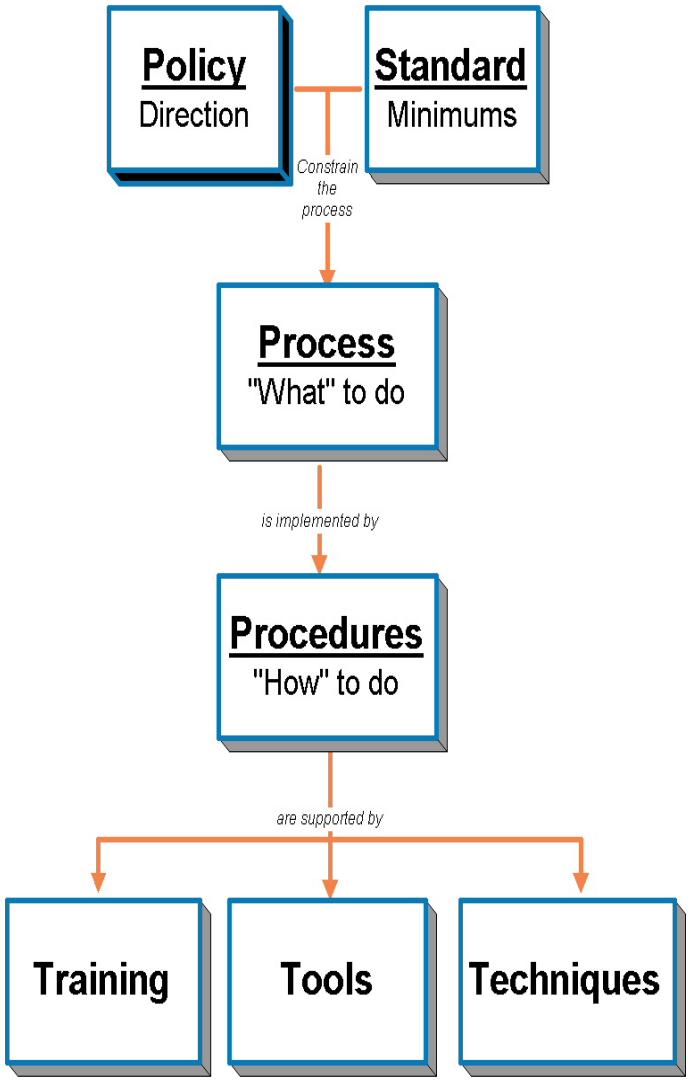
This paper defines the products for a process improvement project, outlines a practical work breakdown structure for a process improvement project to deliver these products, and offers a strategy for implementing the process in the organization. The paper also presents insights into the difficulties of bringing about organizational change.

1 PRODUCTS OF A PROCESS IMPROVEMENT PROJECT

1.1 Operational Framework and its Relationship to CMM Common Features

The deliverables that characterize the processes must demonstrate a commitment to change as defined in the operational framework. The operational framework is used to implement the common features of CMM key process areas (KPAs) as defined in ***How to Use the Software Process Framework***[i]**.** The framework is useful whether implementing CMM key process areas or another process. The elements that make up the operational framework and their purposes are listed below:

Figure 1 - Operational Framework



Policy — provides the "law" or regulations that govern, guide, or constrain operations.

Standard — provides the operational definitions or acceptance criteria for final or interim products or processes.

Process — describes what happens over time within the organization to build products that meet the standards in accordance with organizational policies.

Procedures — describe step-by-step instructions that implement a process.

Training — provides people with necessary knowledge and skills, including training on organizational policies, standards, processes, procedures and tools.

Tools — provide the needed support for organizational policies, standards, processes, procedures and training.

Techniques — are the methods or means used to execute the procedure steps.

1.2 Benefits of the Operational Framework

Using the operational framework to organize process documentation will provide these benefits:

- **Each process product has a specific purpose with no redundancy**. A common tendency is to gather too much disparate information in a single document. For example, if procedural instructions are included in the policy, or training materials are found in the procedures, each document becomes less usable and loses its focus. With an operational framework, this is no longer a problem.

- **Common understanding and definition of the process products are achieved**. With any change, communication is probably the most important factor, and a common language is the most important element of communication. The operational framework provides one basis for communication.

- **Information is easier to find**. Once the organization understands the framework, members know where to quickly find what they need.

- **Maintenance is more manageable** because stable elements of a process are kept in different products, separate from the more volatile documentation that must adapt quickly to changes. Policies, standards and processes are high-level guidance documents that would not be expected to change often. Procedures and training will change more often than policies, but not as frequently as tools. The organization should allow and encourage frequent adjustments to tools and techniques in order to adapt to changes.

2 PHASES, ACTIVITIES, TASKS AND DELIVERABLES OF A PROCESS IMPROVEMENT PROJECT

When the products and outcomes of the process improvement effort have been defined, the plan of action for the project must be defined. The scope of the project is portrayed in a work breakdown structure that ultimately will be used to define the activities, schedule, and deliverables for the project.

2.1 What is a Work Breakdown Structure (WBS)?

The phrase "work breakdown structure" has been used to refer to several types of breakdown structures including product breakdown and organization breakdown. These structures are based on a top-down hierarchy of definition from general to very specific.

A product breakdown structure reflects what is being built, frequently referred to as a product description. For example, the operational framework provides the product breakdown structure in a process improvement project.

A work breakdown structure is all the work required to deliver the product, including the planning and managing.

An organization breakdown structure describes the organization and is referred to as the organization chart.

The relationships between the three are; the organization chart (organization breakdown structure) helps identify who in the organization will perform the work (work breakdown structure) to deliver the defined product or service (product breakdown structure).

For the purposes of this paper, the term, "work breakdown structure" refers to the structure that defines the work required to produce the product or service. It is the definition of the work scope for a project, and the intent is to define all the work of the project.

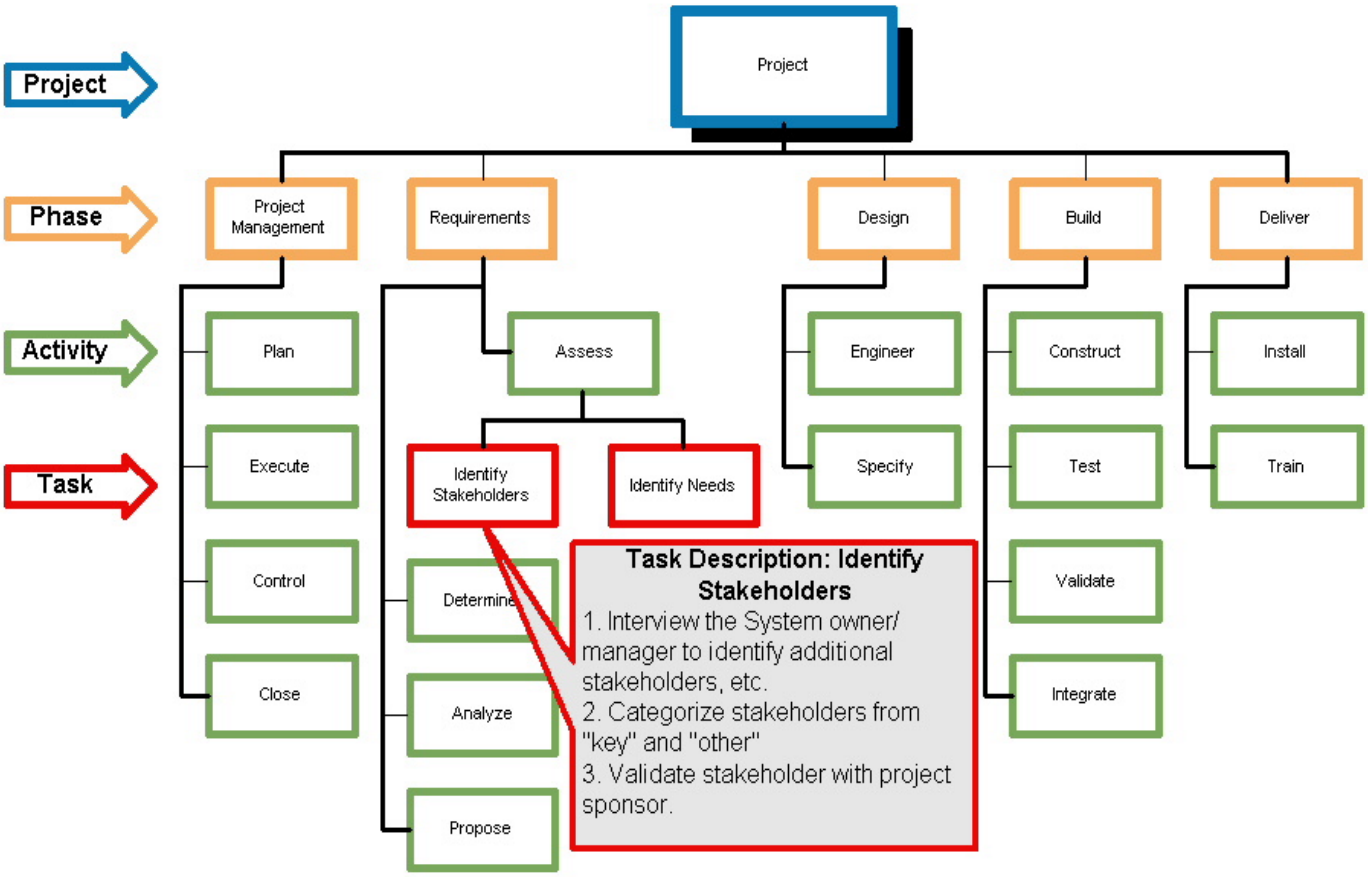
2.2 A Common WBS for System Development Projects

The hierarchy of a work breakdown structure can be described this way:

- **Phase** – major stage of project or life cycle methodology
- **Activity** – logical grouping of tasks
- **Task** – defined as the effort to produce a deliverable, approximately 40 – 80 hours
- **Task description** – detailed definition of the task for common understanding

An example of a generic WBS follows:

Figure 2 - Common WBS for System Development Project



Several benefits can be realized by an organization that uses a generic WBS tool as a starting point for the development of a project-specific work breakdown structure:

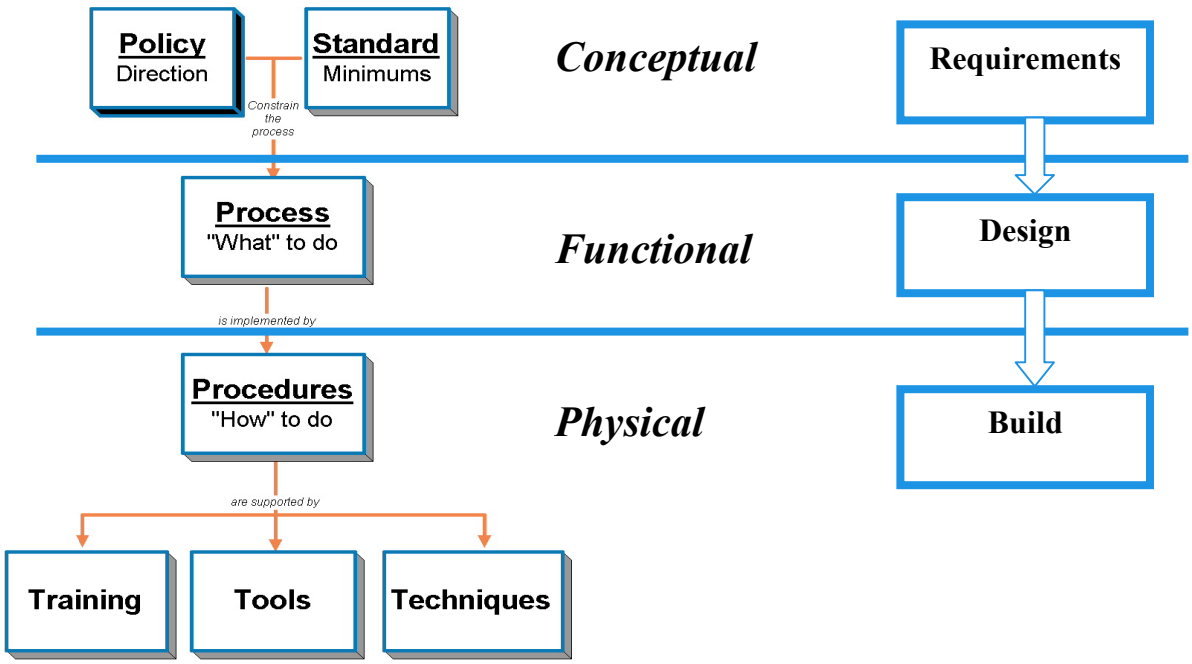
- Reinforcement of a standard methodology: phases, activities and deliverables
- Common description of deliverables
- Communication tool for identifying customer expectations

2.3 Relating a System Development WBS to a Process Improvement Project

In developing our WBS for a process improvement project ODOT IS's PIP team first examined the products that made up the process; the operational framework. Then the team looked to system development methodology for a guide to establishing the phases and activities to develop the products of the operational framework.

The team determined that the conceptual products of the operational framework represented the policy and the policy standards. The functional level equated to the process product, and the physical level represented the Procedures and all the supporting products; tools, techniques and training.

Figure 3 - Relating System Development WBS to the Operational Framework



The next step was to identify which activities are required to develop the process products for each phase of the life cycle, matching the conceptual, functional and physical.

ODOT IS's efforts realized a work breakdown structure for process development and process installation.

Customers then could choose tasks and activities that address the characteristics of their process improvement projects. Project characteristics include the complexity of the process being implemented, acceptance level of the user community, and the technology to be introduced in the process solution.

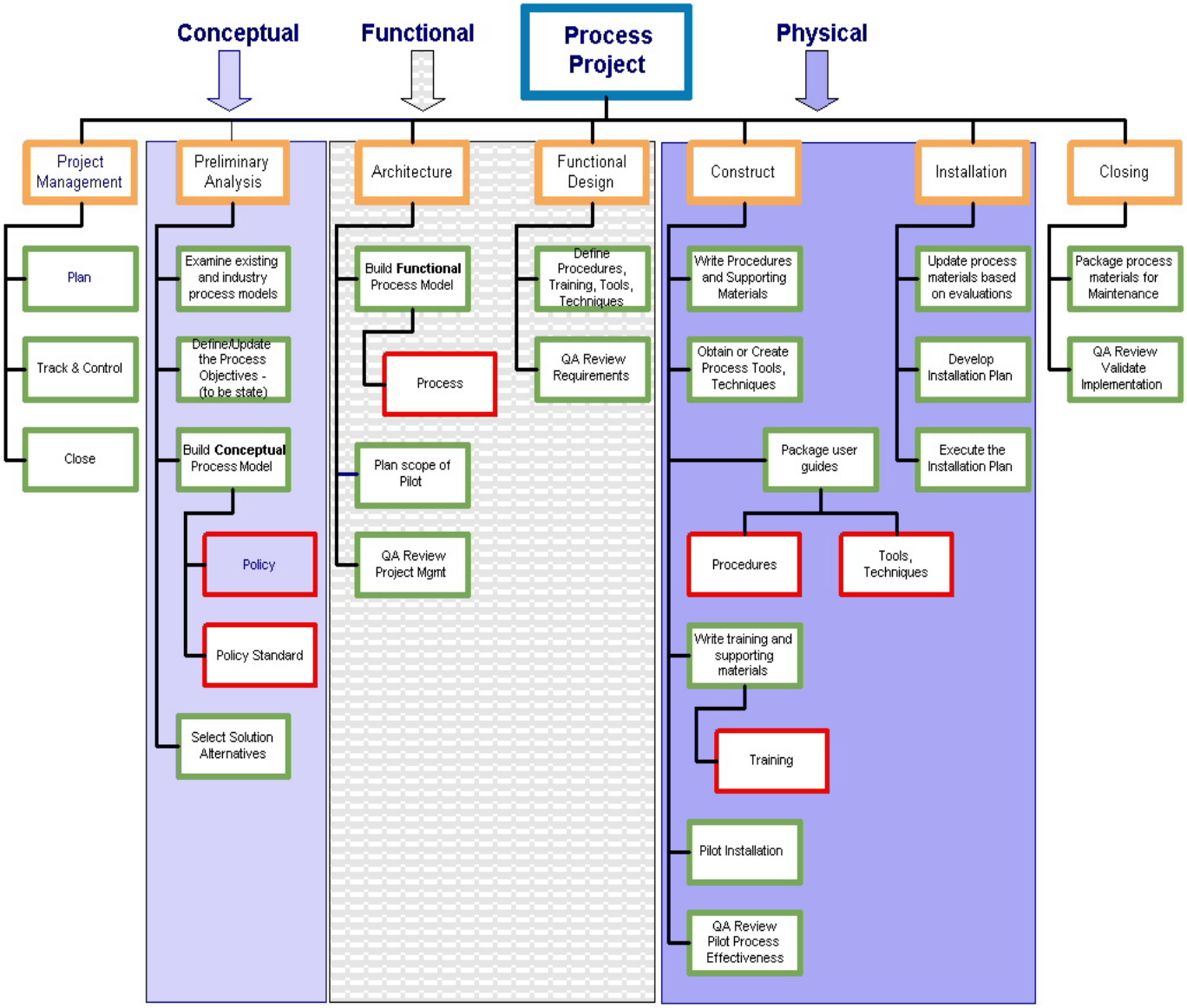
There were significant differences between a system development project and a process improvement project. The most notable are identified below:

- The steps of acceptance and validation were more important and time-consuming factors in process development than in systems development.
- Resident process knowledge was lacking for user requirements/validation of a new process.
- The procedure-level user guide and training are the real "end product" for a process project.

2.4 Our WBS for Process Improvement

Below is the WBS ODOT IS developed for process improvement projects. The generic phases have been replaced with ODOT's standard system development methodology

Figure 4 - ODOT's WBS for Process Improvement



To complement this graphical representation of ODOT IS's process WBS, the team developed a WBS dictionary that provides a list of all the tasks for each activity represented above.

3 TAILORING THE WBS FOR THE SELECTED STRATEGY

There are many different ways to implement process improvements in an organization. The strategy must be sensitive to the organizational hierarchy and the type of process being implemented. The purposes of defining the implementation strategy are to recommend the best scenario for proceeding and to define how the process improvement will be delivered across the organization.

3.1 Determine Implementation Strategy

A new process may be implemented in one release or several release(s), where there is a need to introduce change in the organization progressively. The availability of human and material resources and technologies also must be considered, due to the potential impact it might have on development costs and system performance. The strategies also define whether each release should be implemented simultaneously in all organizational units, for all client groups and at all locations, etc. If the change is to replace an existing process, the strategy indicates whether there will be parallel operation of the two systems during implementation.

The strategy also must consider the workload and time required to:

- develop and test processes;
- train staff;
- install the human resources and infrastructure for ongoing process use, operation, maintenance and support.

The strategy might need to define alternative scenarios outlining the proposed implementation strategy[ii]. Each scenario groups all the process releases and suggests a release schedule. The releases generally result in a progressive implementation of the process. The implementation strategy may contain a description of the advantages and disadvantages of each scenario in terms of the time required to attain the objectives, the capacity of the organization to absorb change, the impact of the transitional periods between releases, and the risks inherent in each scenario.

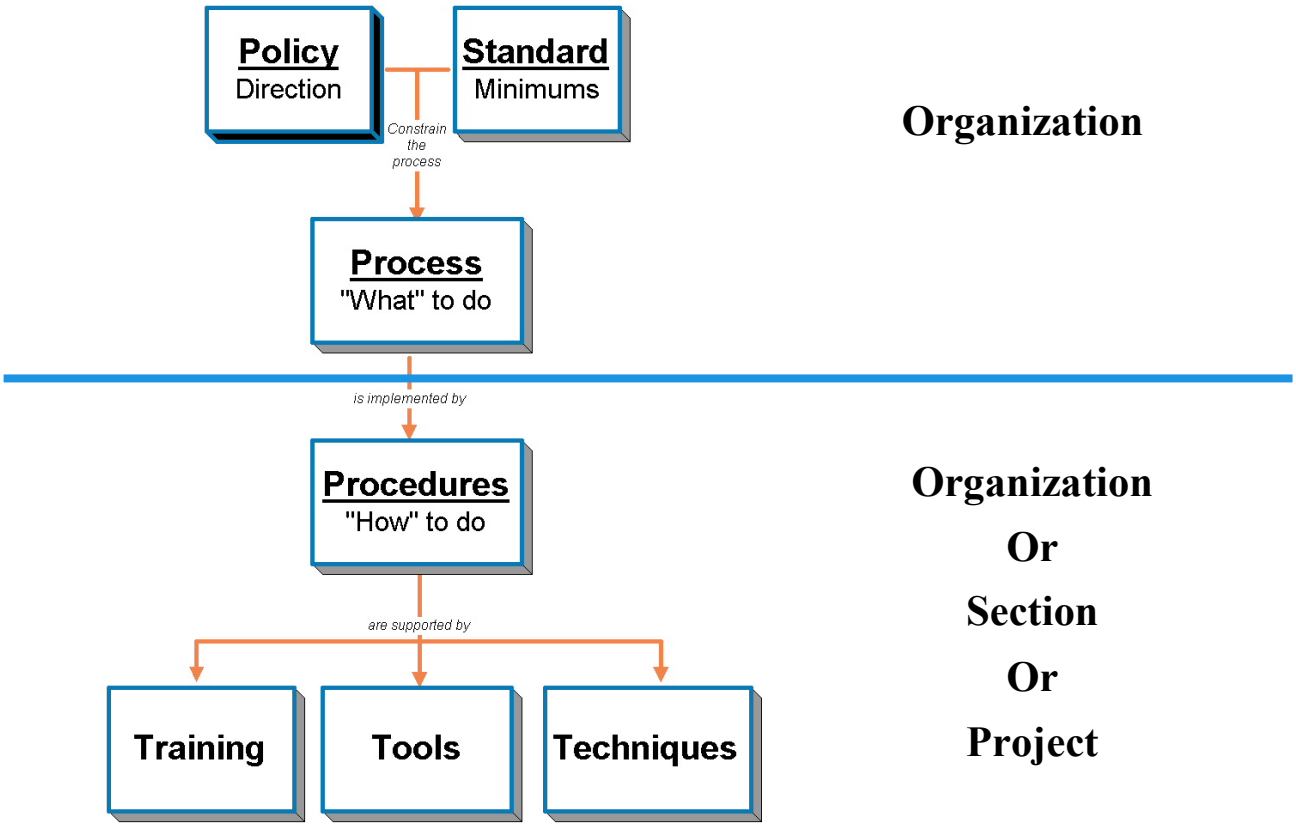
3.2 Impacts of Implementation Strategy on the WBS

The implementation strategy will impact the WBS in terms of who does what and how many times or iterations.

For example, in ODOT IS there are three separate application development sections supported by a centralized technology management group and the Office of Information Technology group that includes; process improvements, security, position management, contract management, and the IS library.

Because ODOT IS has a centralized process improvement function, the organization has elected to establish organization-level products within the operational framework and optional section level products as follows:

Figure 5 - Implementing the Operational Framework



This approach supports goals and mission of obtaining the benefits of a CMM Level 3 organization by standardizing the conceptual and functional levels of all processes developed while maintaining the flexibility to customize procedures specific to the application development sections' needs.

ODOT IS has used several different strategies to implement the peer review, quality assurance, project planning tracking & oversight, software configuration management and requirements management KPAs.

3.2.1 Enterprise-Wide

ODOT IS used an enterprise-wide approach to implement a quality assurance process. This is a CMM Level 3 tactic that assumes a certain level of consistency across the organization is required. This strategy was driven by the fact that quality assurance services were being outsourced under a single contract for the entire IS organization. The process action team was augmented with representatives from each application development section. The section representatives were chosen for their influence within the application development sections. The core project team developed the quality assurance operational framework and used the section representatives to validate its content and market the products within the section.

The advantage to this approach is that only one set of documents (policy, standard, process, procedures, tools, techniques and training) must be developed, and the set can be rolled out to the organization in a single implementation effort. This commonality across the organization is an important step toward Level 3 benefits. The disadvantage in this approach is that it is more difficult to overcome cultural differences and gain acceptance of the operational products.

3.2.2 Section Level

A section-based strategy was used to implement a peer review process. The team described this as a Level 2.5 approach. In this implementation a host section was identified to be the first to implement the process and provide project management for the process improvement project. The host section, working with representatives from across the organization, developed an enterprise-wide policy, standard and process. Once those products were approved, the host section developed procedures, tools, techniques, and training that would work in that section. When the host section implementation was complete, the other sections could implement the process by adjusting and customizing the procedures to meet their needs, as long as they followed the policy, standard and process. This approach was taken because of the sections' need for the procedures to fit within the culture and operations in their individual sections.

The advantages in the section-level approach are that each organizational section can define the procedures in the terms that its is most familiar with and that the approach allows for the possibility of developing parallel procedures, tools and training for many sections. Disadvantages are that the section-level approach will take more effort and that potentially multiple sub-projects will be required to develop procedures and training for each section. In addition, in some process areas, this approach might not take the entire organization to Level 3 benefits.

3.2.3 Project-by-Project

The project-by-project approach was used to implement selected activities of the project tracking and oversight process. This is a Level 2 approach. Each project's team determined its own internal procedures for activities, such as issue management, change management, and document control. Procedures were established by tailoring generic procedures by the project manager. The project team followed these procedures during the course of the project and abandoned them when the project was completed. Generic procedures were based on organizational policy, policy standard, and process.

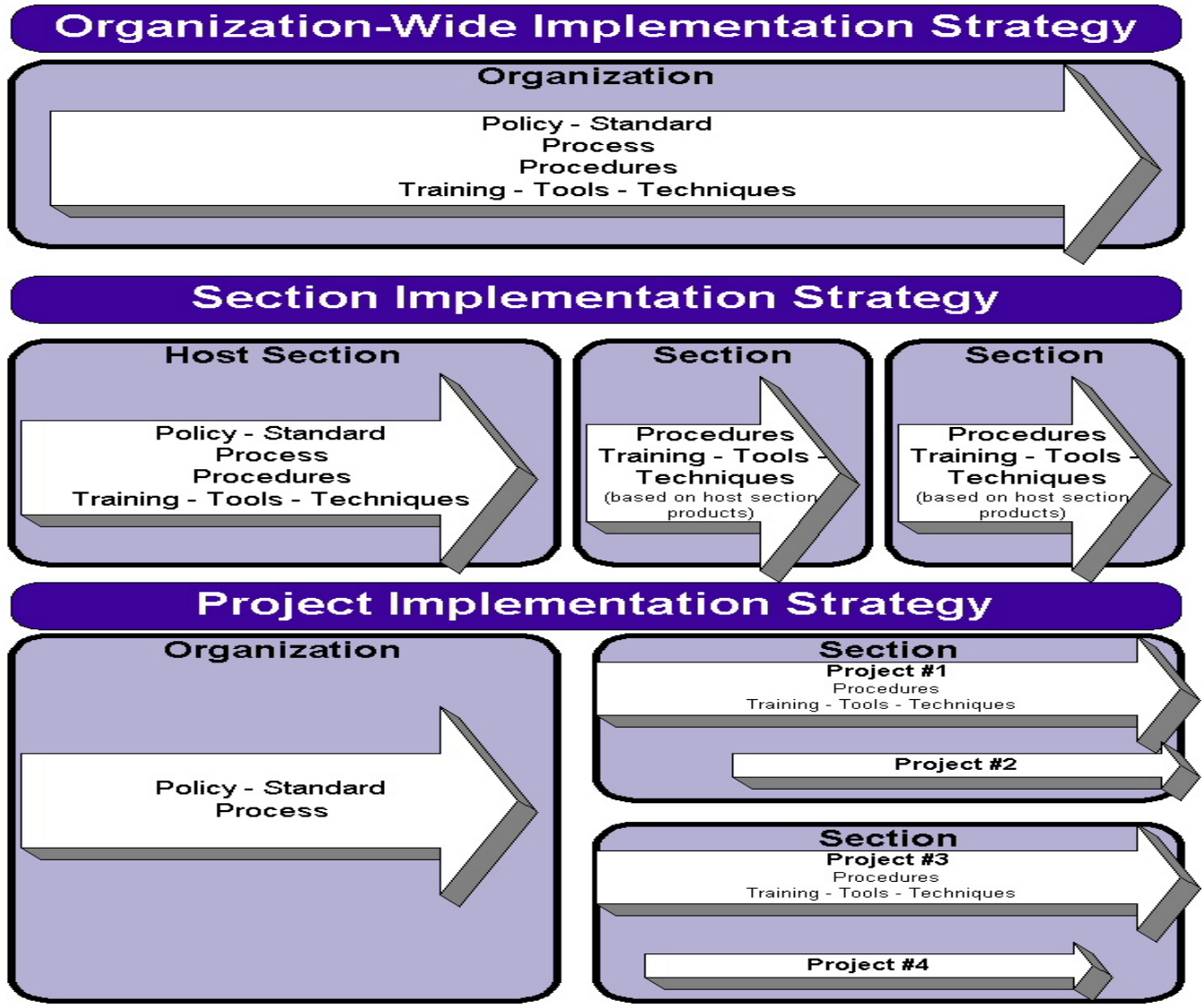
The advantage of the project-by-project strategy is that it can be implemented quickly by developing procedure templates and then allowing each project team to tailor them to meet its needs. However, if tailoring becomes extensive, each project team might not reinvent the wheel at the start of each project, but it might spend unnecessary time designing a new hubcap for their project. The success of this approach is highly dependent upon the skills of the project managers involved.

3.3 What to Tailor within the WBS

The generic process WBS includes all the components to develop and install a process. The different strategies tend to affect how many times the implementation or installation effort takes place, rather than how the process products are developed.

For example, the policy, standard, and process products are only developed once for the entire organization. The physical level or procedures, training, tools, and techniques may be created once or several times reflecting an organizational, section, or project-by-project implementation. The diagram below reflects how the different strategies affect the installation efforts

Figure 6 - Implementation Strategies

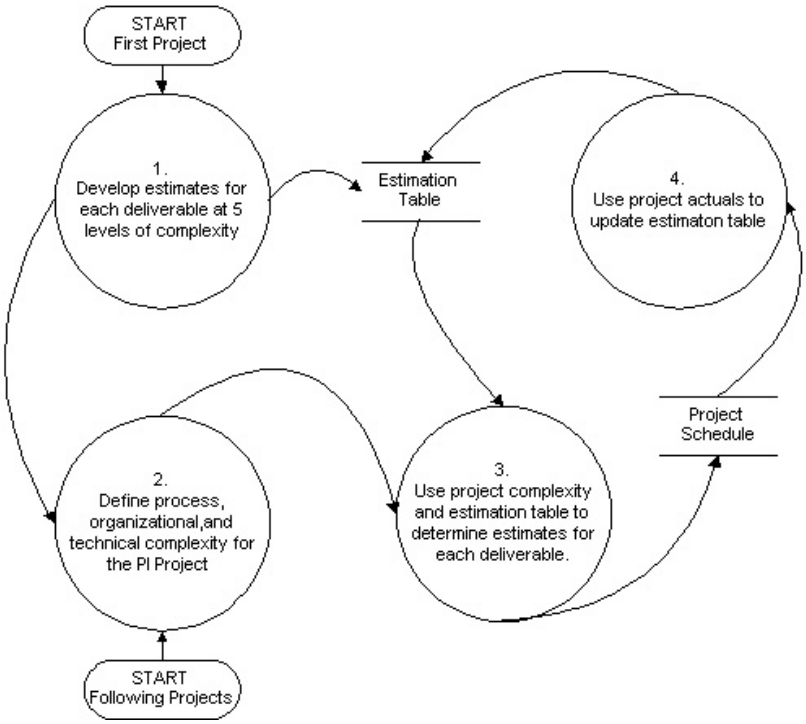


4 ESTIMATING THE EFFORT OF A PROCESS IMPROVEMENT PROJECT

With the generic work breakdown structure for process in hand, the next effort was to estimate the process improvement project. Because ODOT IS had no history of process development and installation, the team investigated conventional system development estimating methods and found that system development estimating models did not apply to estimating a process project. For example, lines of code and function points estimating didn't apply. A single one-page policy and two-page procedure can be very simple to implement or take days of training and coercion. ODOT IS developed the model to gauge the size and effort involved in the process improvement projects with the intention of improving estimation on each project. Initially the effort estimates were used for strategic planning purposes only. Detailed project plans were developed for each process improvement and installation project at the time the project was initiated.

The estimation model is illustrated at a high level in the following diagram. Following is a description of each step in the process.

Figure 7 - Estimation Process



1. **Develop Estimation Table**. The purpose of the estimation table is to capture the estimated work effort required to produce each deliverable identified in the WBS. The hourly effort for each deliverable was estimated at one of five levels depending on the complexity of the project. The first cut of the estimation table was simply the best guess based on prior experience with application development projects or experiences in previous process improvements. The team made and documented a number of assumptions in order to complete the estimation table. For example, it was assumed that project management deliverables and activities were a percentage of the total effort for the entire project and that

the percentage increased as project complexity increased. A simplified estimation table is shown below.

	Level of Complexity				
	1	2	3	4	5
Deliverable	**Hours of Effort**				
Policy	10	10	15	25	40
Policy standards	30	35	45	55	60
Process definition	20	25	30	35	40
Procedure specifications	20	35	50	75	100
Training specifications	20	35	50	75	100
Tools specifications	20	35	50	75	100
Technique specifications	20	35	50	75	100
Procedure *n* (repeat for each procedure)	20	35	50	65	80
Tools and templates	40	50	60	70	80
Process user guide	40	50	60	70	80
Training package	85	180	330	560	800
Implementation strategy	20	40	60	70	80
Installation plan	40	75	110	145	180
Training delivery	60	90	120	160	200
Coaching delivery	40	100	180	280	400
Estimated effort for PM is a % of total effort	**% of Total Effort**				
Project planning, project management and change control	10%	13%	15%	20%	25%

2. **Complete Complexity Worksheet**. In order to determine the complexity of a specific process, ODOT IS developed and used a complexity worksheet. The complexity worksheet poses questions about the planned process improvement projects and rates the responses from 1 to 5, with 1 being the simplest and 5 the most complex. The worksheet partitions the process improvement project complexity into three impact areas:

   Process complexity is based on the number and type of new policies, processes, activities and techniques that are within the scope of the process improvement project. It also considers the how much of the process is already in place in the organization (process readiness) and how aware the organization is of the process in general (process knowledge).

   Organizational complexity is based on the number of organizational sections involved, impact on sections, and expected resistance to change in both the IS and business areas.

   Technical complexity considers the tools and technical components needed for project success. It must take into account the number, type and cost of new tools needed, the number and type of tool users, programming required, resistance to the tool, and technology required.

   The ODOT IS team used the results to quantify a complexity rating for each impact area. The complexity ratings in each impact area were weighted and averaged to produce an overall complexity rating for the project. A sample complexity worksheet is shown in the appendix.

3. **Derive Hourly Estimates.** The complexity ratings then were mapped against the estimation table to produce the estimated effort, in hours, of each deliverable. The overall complexity was the main consideration, but some deliverables were determined using only the technical

complexity. Other deliverables' estimates were multiplied based on the number of processes and procedures that would be needed for the particular KPA.

4. **Update Estimation Table with Project Results.** At the close of each process improvement project, the estimation table is reviewed in light of actual hours needed to complete the deliverables. If warranted, the estimation table is updated for use in future projects.

5 CONSIDERATIONS FOR PROJECT SUCCESS

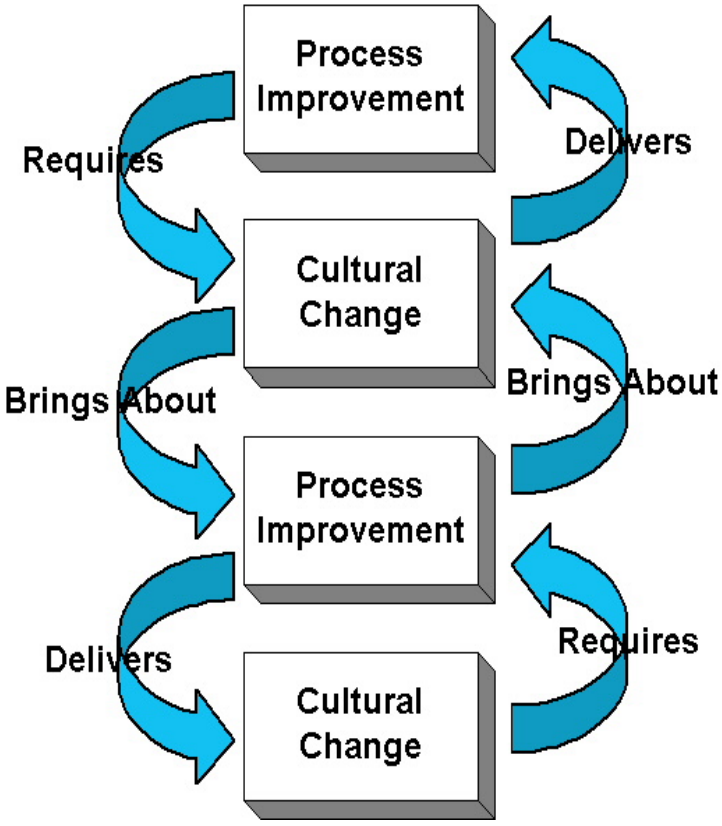
Human factors and organizational dynamics are as important, if not more important, in process improvement effort as they are in systems delivery. Getting people to use a new process is not as simple as inserting the installation disk and following instructions. This section contains some insights into key considerations that will pave the way for project success.

5.1 Organization Culture and Readiness

In her book, *CMM Implementation Guide*[iii], Kim Caputo outlines the difficulties in bringing about process improvement and organizational change. The pattern of shared basic assumptions about solving problems and working together defines the organization's culture. A common situation for ODOT is crisis intervention and fire fighting. The underlying shared assumptions are that:

- If the best people work on solving problems, they will exert heroic effort to save the project
- An organization has no control over problems. All it can do is react when problems occur.
- Organizations don't have time to plan ahead. They must finish a project as soon as possible.
- Hard work and heroics in the face of crisis are rewarded and reinforced.

Figure 8 - Catch 22s



Process improvement and cultural change are often caught in a number of Catch-22s, as illustrated in this graphic.

- Process improvement requires culture change; and cultural change requires process improvement.
- Process improvement brings about culture change; and cultural change brings about process improvement.
- Process improvement provides the foundation for cultural change, and cultural change provides the foundation for process improvement.

In order to interject a change into this cycle there are a number of

prerequisites that must be in place. Each must be present in alignment before people's behavior will change. ODOT IS found the following to be the keys to enable change:

- Vision: Vision makes the goals real. Without vision people will be confused.
- Skills: People need to know how to do what is being asked.
- Incentives: They need a reason to change.
- Resources: They need the time to do it.
- Plan: They need to know the steps required for change.

The degree that an organization's culture is receptive to defined and documented processes will define the degree and effort required to bring about process improvements. ODOT IS's insights, of a rather resistant culture, have identified the following:

- Mid-level managers are more impacted by organizational change than executive management. Mid-level manager may not be as supportive as executive management and may act as barriers against change in an organization. Emphasis on communication and definition of expectations has helped mitigate risks with this group.

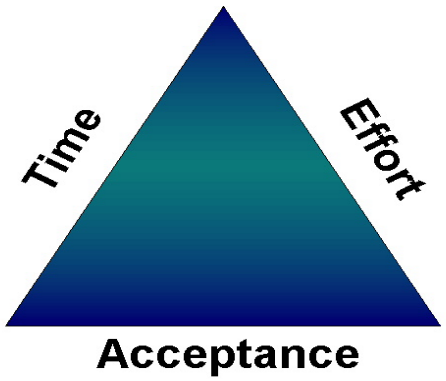
- Government agencies tend to have long-term employees, with little threat of firing or means of providing compensation benefits for incentive of process improvement efforts. More emphasis is on the consensus model to build acceptance.

- Even when groups are similar, they won't admit it or recognize it -- they don't trust that a process developed outside their group could work for them.

5.2 The "Project Triangle"

Projects always face tradeoffs among project objectives. Performance in one area can only be enhanced by sacrificing performance in another area. While the specific tradeoffs will vary from organization to organization and from project to project, identifying and adjusting for these constraints are essential to the success of the project. Triple constraints often are depicted as a triangle where the sides represent the constraints to be managed in the project.[iv] Typically, the three major constraints of a project are time to implement cost and scope of the effort. In ODOT IS's process improvement efforts, the team found that time, effort, and acceptance were the constraints to be managed.

Figure 9 - The PI Project Triangle



1. **Where the project priority is reduced time**, the project would limit the scope or the effort and acceptance. The product development time is minimal compared to the time it takes to gain acceptance. This is particularly true when the affected groups or individuals have not been involved in the definition or development of the process products.

2. **Where the project priority is reduced effort (resources),** the project would encourage a centralized approach to development and installation. Additionally, other factors that could affect this approach would include acceptance or institutionalization of the process.

3. **Where the project priority is increased acceptance,** the project would include more participants in the development and installation aspects of the project. This could be reflected in a large and timely development effort of the process and/or several pilots and refinement efforts.

5.3 Process Area

Some key process areas within the CMM are better suited to a top-down, rather than bottom-up, approach for process development and installation. The Level 2 KPAs within CMM are project centric, while Level 3 KPAs are organizational in nature and are better suited to an organizational approach.

ODOT IS has applied a centralized approach to quality assurance, project planning, tracking and oversight, and the training KPAs with much success. The organizational approach also is influenced by the cost and efficiency usage of tools to support these KPAs — For example, the project management tools for project scheduling and tracking.

KPAs that involve the system development or software engineering staff as primary roles tend to favor a section-level implementation. The acceptance factor is significantly improved when these stakeholders are able to design procedures, tools, techniques and training to their specific needs and requirements. What ODOT IS has found is that the procedure-level products are not very dissimilar from section to section, but that acceptance was in the process of development, rather than the content of the process products. The following illustration provides a breakdown of KPAs for Levels 2 and 3, and the primary roles affected:

Figure 10 - Level 2/3 KPAs by Role

Level 2	Level 3
Performed by - Project Management Staff	
Project Planning	Integrated software management
Project Tracking/Oversight	-
Performed by - Product Development Staff	
Requirements Management	Product engineering (life cycle)
Configuration Management	Inter-group coordination
Subcontract Management	Peer Review
Performed by - Organizational Staff	
Quality Assurance	Organizational Process Definition
-	Organizational Process Focus
-	Training Program

Diagram is from Software Process Improvement with CMM by Joseph Raynus[v]

6 CONCLUSION

ODOT Information Systems' experience has found that process improvement is best accomplished and organized around a PI project that is defined, managed and run using solid and proven project management principles. More casual, less organized approaches to process improvement can produce results, but results will lack focus and will not disperse adequately across the organization. ODOT has tried a casual approach, and results were random and short-lived. In addition to improving processes in the organization, ODOT is attempting to change the organization's culture. Historically, good processes were developed and used by focussed individuals striving to do their jobs better and more consistently. These processes spread like folklore across the organization. They were shared casually and often faded away upon the sage's retirement. ODOT's Process Improvement Program is striving to collect this process information, create a process-knowledge repository, make ongoing improvements and promote use across the organization.

Process improvement would be easy if it were just a matter of documenting what needs to happen and installing the instructions among those who do the work — like inserting a roll into a player piano. The keys will press and release in the same sequence every time. Installing a new process into those who do the work is a much more complex, difficult and dynamic course of action. The process users are each unique, thinking and feeling individuals who are part of a culture with certain expectations about "how things work around here." A successful PI project is one that changes behaviors and, produces "standard operating procedures" that everyone in the organization can use everyday to get work done. These changes only occur if the humans involved in the process see the rationale and value of the new process. People will challenge things that don't make sense and will not follow procedures for which there is poor rationale (unless there is a looming penalty, e.g. IRS audit, pink slip, etc.). This aspect of human nature is a good thing[vi] because it enables processes to improve over time, provided all participants agree on a starting line from which to begin making changes.

7 APPENDICES

7.1 Process Complexity Worksheet

The process complexity worksheet is used to estimate a project's complexity along three dimensions: process complexity, organizational complexity, and technical complexity. The worksheet presents questions about the planned PI projects. Evaluators then rate the responses 1 to 5, with 1 being the simplest and 5 being the most complex. The evaluation group reviews the response for each question and reaches consensus about the complexity level for each dimension and the overall project complexity. The intention of the worksheet is to facilitate a focussed discussion, rather than to presents a mathematical model for determining process complexity.

	Process Complexity Level				
	1	2	3	4	5
Number of new policies	0		1		More than 1
Type of new policy	Simple		Moderate		Complex
Number of processes	1		More than 1		More than 3
Type of process	Simple		Moderate		Complex
Number of activities	1 to 3		3 to 9		10 or More
Type of activities	Simple		Moderate		Complex
Number of new techniques	1 to 3		3 to 9		10 or More
Type of new techniques	Simple		Moderate		Complex
Process readiness	Many processes in place		Some processes in place		Very few processes in place
Process knowledge	Many knowledgeable staff		Some knowledgeable staff		Very few knowledgeable staff
	Organization Complexity Level				
	1	2	3	4	5
No. of IS sections involved	1 to 3		3 to 5		More than 5
Type of IS changes	Minor		Moderate		Major
IS resistance to changes	Minor		Moderate		Major
No. of bus. sections involved	1		2		More than 2
Type of bus. changes	Minor		Moderate		Major
Bus. Resistance to changes	Minor		Moderate		Major
	Technical Complexity Level				
	1	2	3	4	5
Number of new tools needed	0		2		3 or More
Type of tool	Minor		Moderate		Major
Number of tool users	Few		Some		Many
Cost of tools	Less than $1000		Between $1K & 5K		More than $5000
Type of tool users	IS users only		IS and some business		IS and business users
Programming required	User developed		Installation and configuration		Custom built
Resistance to tool	Minor		Moderate		Major
Technology required	Exists everywhere in ODOT		Exists somewhere in ODOT		New to ODOT

REFERENCES

i SEI Special Report CMU/SEI-97-SR-009, October 1997, Linda Parker Gates

ii P270G - Implementation Strategy - DMR ProductivityCentre DMR MacroscopeTM 3.1 DMR Macroscope Workplace Version 3.1.0 © 1998, DMR Consulting Group Inc.

[iii] CMM Implementation Guide-Choreographing Software Process Improvement ©1998, Kim Caputo

[iv] A Guide to the Project Management Body of Knowledge (PMBOK ® Guide) 2000 Edition, Project Management Institute, Inc.

[v] Software Process Improvement With the CMM, ©1998 Joseph Raynus

[vi] Martha Stewart Living Magazine, Martha Stewart

From Requirements to Release Criteria:

Specifying, Demonstrating, and Monitoring Product Quality

Erik Simmons
Intel Corporation
JF1-46
2111 NE 25th Ave.
Hillsboro, OR 97214-5961
erik.simmons@intel.com

Version 1.1, 08/17/01

Prepared for the Pacific Northwest Software Quality Conference, 2001

Key Words: Product Quality; Quality Requirements; Quantified Quality; Release Criteria

Author Biography

Erik Simmons has 15 years experience in multiple aspects of software and quality engineering. Erik currently works as Platform Quality Engineer in the Platform Quality Methods group, part of the Corporate Quality Network at Intel Corporation. He is responsible for Requirements Engineering practices at Intel, and lends support to several other corporate software and product quality initiatives. Erik is a member of the Pacific Northwest Software Quality Conference Board of Directors and the Steering Committee of the Rose City SPIN. He holds a Masters degree in mathematical modeling and a Bachelors degree in applied mathematics from Humboldt State University in California, and was appointed to the Clinical Faculty of Oregon Health Sciences University in 1991.

Abstract

Without good planning and monitoring, product quality is often a matter of chance. Specifying quality requirements is an excellent first step to predictable, managed product quality, but it is just the beginning. Quality specification, quality planning, data generation, quality monitoring, and quality reporting all work together to ensure that quality levels are known throughout the project. Quality release criteria are a good way to ensure that product quality drives interim milestone releases and the final product release. The breadth of quality dimensions can force teams to learn new concepts and skills in order to demonstrate the qualities associated with the release criteria. Despite this challenge, quality specification and quality monitoring are excellent practices that help monitor end product quality throughout the lifecycle.

Introduction

If you ask a typical product or project manager what his biggest challenge is, you will often get answers involving cost and schedule. Only occasionally in the commercial product development arena will you hear product managers mention product quality as a significant concern. There are several reasons for this:

- Recent market trends have placed tremendous pressure on companies to be the first to market with a product
- Venture capital funds expect to see a product quickly as a sign of progress
- Products are more complex than ever before, lengthening development schedules

There is some rationale behind a manager's focus on attributes other than quality at first. When bringing a new product to market, the innovators that make up the first wave of customers are much more concerned with features and the newness of the product than with price or quality. Later, when the early majority begins to buy, quality and price become greater concerns [Rogers95].

Despite this reasoning, without good planning and monitoring, product quality is a matter of chance. Numerous teams have launched products only to find out that required qualities were missing or inadequate, leading to expensive and embarrassing retrofits, product recall, or outright product failure in the marketplace. In order to prevent such mishaps, a product development team must perform several activities:

- Quality specification
- Quality planning
- Data generation
- Quality monitoring
- Quality reporting

These activities are not strictly sequential, but can overlap or occur in parallel. Each activity is introduced below and is then illustrated in the example that follows.

Quality Specification

Quality specification is the process of establishing and documenting the product's quality requirements[1].

Quality requirements are often overlooked when specifying the requirements for a product. This could be because the requirements team does not know what qualities should be included, does not know how to write quality requirements well, or is pressured by existing culture and practices to focus solely on functional requirements.

Quality has many dimensions. A holistic view of quality must consider all aspects of quality for all stakeholders, rather than focusing narrowly on defect detection and removal. By taking this approach, known in some circles as "big Q" rather than "little q", the resulting quality specification will thoroughly cover product quality.

[1] Quality requirements are called "non-functional requirements" in many sources, but this seems to be a poor term for what are essentially qualities representing how well (fast, reliable, etc.) the product's functions must deliver what they provide. In fact, many students laugh out loud when the term is introduced during requirements classes, joking that it must refer to the features in the product that don't work. The same term is also applied to some design constraints, further clouding its meaning.

Although there is no standard categorization of quality requirements, the set of categories shown in Figure 1 is typical. Organizations typically customize such lists and add details based on the type of product they build and areas in which they have faced problems in the past. For example, software performance can be segmented into 'time' (speed) and 'space' (resource usage). Security contains many facets, including integrity, availability, confidentiality, and accuracy [Chung00]. Availability can be separated into reliability, maintainability, and integrity [Gilb88]. One exhaustive list of quality requirement categories has161 elements [Chung00]. Overlaps within definitions are extremely common within the different sources. The best advice is to establish a well-defined list of categories and maintain it over time, based on project experience and need.

- Performance
- Availability
- Usability
- Supportability
- Security
- Maintainability
- Manufacturability
- Testability
- Extensibility
- Scalability

Figure 1. Example Quality Requirement Categories.

Quantifying Quality Requirements

To be meaningful, quality requirements must be verifiable and measurable. Stating quality requirements precisely in natural language is difficult. Authors often use weak words such as *fast, easy, simple, minimal, as needed, adequate, normal,* and a host of others [Wilson96]. These qualitative terms leave the meaning of a requirement open to different interpretation by each reader. Such ambiguity leads to missed expectations in the final product, and prevents accurate testing.

Planguage is a technique to quantify qualitative statements that works very well on quality requirements. Planguage is a keyword-driven specification language that improves communication about complex ideas. Easy to learn and use, Planguage also provides many benefits [Simmons01, Gilb01, Gilb97a, Gilb97b]. A basic description of Planguage is given in Appendix A. Planguage is used to document the quality requirements in the example later in this paper.

Setting Target Values

Before we can quantify quality requirements, the levels of achievement for each quality requirement must be defined through target setting. Target setting is most often based on analysis, benchmarking, or standards. Each type of quality requirement (performance, reliability, scalability, etc.) can require a specific approach to target setting in order to obtain accurate, realistic, and meaningful targets. Setting quantitative targets for these attributes is challenging at first, forcing a team to think in new ways about the product it is building. However, most of the methods are not difficult and the results more than justify the effort. The toughest part of the process is often finding the necessary data or applicable standards.

Quality Planning

If quality only made it as far as a requirements specification, it would be of little value. The additional activities of quality planning, data generation, quality monitoring, and quality reporting are the keys to obtaining real value from quality requirements.

Many existing sources discuss the need to use quality requirements to guide the subsequent product design, and for traceabililty between requirements and design as a form of verification. But this is not enough. Quality must permeate beyond the specification and design to become one of the guiding principles for interim milestone reviews and the final release of the product.

Capturing the important product qualities from the requirements as quality release criteria helps ensure this happens.

Quality Release Criteria
Quality release criteria (QRC) are used to assess and monitor the risk to interim milestone releases and the final product release. Quality release criteria cover a diverse set of quality attributes for a product. Some of the criteria are related to product quality requirements. Others are related to SQA, test, software configuration management, or project management data.

The QRC are grouped within assessments, which makes quality reporting and quality monitoring simpler. Assessments similar to those used at Intel are shown in Table 1 [Gatlin00]. Some assessments only apply to certain situations or categories of products. The assessments are always used as a point of departure on a project. Each project must customize and adapt the assessments and underlying quality release criteria to its needs.

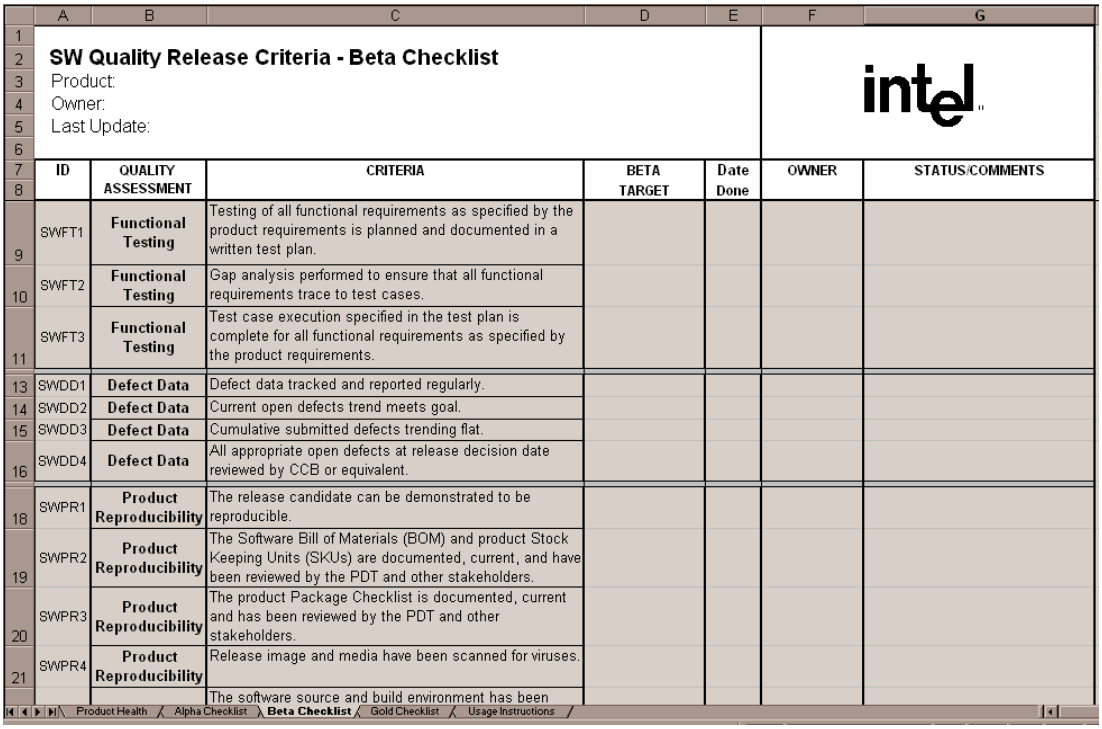
Table 1. Assessments Containing Quality Release Criteria.

Assessment	Description
Functional Testing	Evaluation of the testing of features and functions to determine whether the testing is adequate to ensure quality.
Product Reproducibility	Evaluation of the ability to archive, reproduce, and rebuild the software product, and that the consistency of the product package with the contents of the released product has been verified.
Legal Compliance	Evaluation of product compliance with legal requirements.
Continuous Operation	A method for detecting time-dependant product defects such as memory leaks and race conditions. It is also one indicator of product reliability over time.
Defect Data	Evaluation of product defect information and verification of correct product operation.
Ease of Use	Evaluation of the ease of use testing to ensure it covers at least the product's user interface and all features related to ease of product use.
Install Testing	Evaluation of install and uninstall testing to verify correct, reliable install and uninstall of product is possible.
Customer Documentation	Evaluation of product user documentation for accuracy and consistency with product operation.
Customer Acceptance	Evaluation of the testing performed to determine whether the product meets its customer acceptance criteria.
Standards Compliance	Evaluation of product compliance with product standards including Intel engineering or other standards. Process standards are specifically excluded.
Compatibility Testing	Evaluation of the product under test to perform its required functions while sharing the same environment with one or more systems or components.
Performance Measurement	Evaluation of the performance attributes of the software under test to determine whether the validation effort is adequate for ensuring that performance goals are met.
Regression Testing	Evaluation of the testing of the product that determines whether changes have caused unintended effects and whether the changed product still meets its requirements.

Assessments can contain different quality release criteria and associated targets for the alpha, beta, and gold milestones. For example, at Intel the Defect Data assessment (named SWDD) is defined to contain the following four criteria at the beta and gold milestones, but only SWDD1 at the alpha milestone:

SWDD1. Defect data tracked and reported regularly.
SWDD2. Current open defects trend meets goal.
SWDD3. All open defects at release decision date reviewed by the Change Authority.
SWDD4. Defects waived by the Change Authority for release are documented in Release Notes.

This system of definition is quite natural to implement in the form of a spreadsheet, with separate worksheets for the alpha, beta, and gold milestones. An example subset of the release criteria for the beta milestone is given in Figure 2.



SW Quality Release Criteria - Beta Checklist

Product:
Owner:
Last Update:

ID	QUALITY ASSESSMENT	CRITERIA	BETA TARGET	Date Done	OWNER	STATUS/COMMENTS
SWFT1	Functional Testing	Testing of all functional requirements as specified by the product requirements is planned and documented in a written test plan.				
SWFT2	Functional Testing	Gap analysis performed to ensure that all functional requirements trace to test cases.				
SWFT3	Functional Testing	Test case execution specified in the test plan is complete for all functional requirements as specified by the product requirements.				
SWDD1	Defect Data	Defect data tracked and reported regularly.				
SWDD2	Defect Data	Current open defects trend meets goal.				
SWDD3	Defect Data	Cumulative submitted defects trending flat.				
SWDD4	Defect Data	All appropriate open defects at release decision date reviewed by CCB or equivalent.				
SWPR1	Product Reproducibility	The release candidate can be demonstrated to be reproducible.				
SWPR2	Product Reproducibility	The Software Bill of Materials (BOM) and product Stock Keeping Units (SKUs) are documented, current, and have been reviewed by the PDT and other stakeholders.				
SWPR3	Product Reproducibility	The product Package Checklist is documented, current and has been reviewed by the PDT and other stakeholders.				
SWPR4	Product Reproducibility	Release image and media have been scanned for viruses.				
		The software source and build environment has been				

Product Health | Alpha Checklist | Beta Checklist | Gold Checklist | Usage Instructions

Figure 2. Example QRC Spreadsheet, Beta Milestone.

Data Generation

Because the quality release criteria cover an array of topics, the raw data to evaluate the QRC come from many sources. Functional testing, inspections, audits, and project metrics are all contributors to at least some of the criteria. Initially, the breadth of the quality release criteria places pressure on the data generation capabilities of most organizations, since evaluating all the necessary criteria requires many new skills, techniques, and tools. For example, relatively few software test teams have experience in demonstrating software reliability through a continuous operation demonstration.

When improved requirements specification techniques are adopted along with a broad set of quality release criteria, an organization must also consider how to train and equip personnel to support the required demonstrations. Without this, the investment in new practices within product requirements and quality release criteria will be frustrating, unsuccessful, and a waste of project resources.

Quality Reporting

Organizations must consider how, how often, and to whom to report the QRC data. The data should be reported to a group chartered with managing the product development effort and monitoring risks to the alpha, beta, and gold milestones. This might be a project office, a project manager, risk manager, product planning council, or another similar organization. Typically, a cross-section of stakeholder groups (such as product quality, program/product management, corporate finance, and product marketing) should receive the release criteria risk data to make sound project-related decisions.

Quality reporting is often a new practice for an organization. Because of this newness and the nature of quality data, quality reporting can be a sensitive topic at first, so be sure that all parties understand the data reported. Quality data is prone to misinterpretation by those unfamiliar with data gathering practices and limitations. Misinterpretation or misuse of the quality data[2] can have lasting negative consequences for an organization. Educating the participants about the definition, collection practices, reporting formats, and allowed uses of the data can help overcome initial reluctance. Sometimes, placing allowed and prohibited uses of quality data into a company policy is also helpful, so long as compliance is monitored and there are consequences for violating the policy.

Quality Monitoring

The quality release criteria enable quality monitoring to begin as soon as the data can be generated. Exactly when this is depends on the nature of each criterion. For example, the product reproducibility criterion cannot be effectively measured until the development team generates a complete system build.

Because of the large number of quality release criteria, a quality dashboard made up of the assessment categories is useful during project review meetings to get a quick but complete picture of quality assessment status (Figure 3). Program managers and others with responsibilities related to product or risk management can monitor the progress towards alpha, beta, and gold milestone releases at a glance, while the details behind the dashboard are available on the sheets for each milestone if needed.

	A	B	C	D	E	F	G
1				Risk Legend:			
2	**SW Quality Release Criteria Scorecard**			1: Showstopper			
3	Product:			2: High Risk			
4	Owner:			3: Medium Risk			
5	Last Update:			4: Low Risk			
6	Current Software Milestone Objective:			5: Negligible Risk			
7	ID	QUALITY ASSESSMENT	DESCRIPTION	RISK		OWNER	NOTES/COMMENTS
8				CURRENT	PREVIOUS		
9		Summary	Overall product health.				
10	SWFT	Functional Testing	Evaluation of the testing of features and functions to determine whether the testing is adequate for ensuring quality.				
11	SWDD	Defect Data	Evaluation of product defect information/verification of correct product operation.				
12	SWPR	Product Reproducibility	Evaluation of the ability to archive, accurately and consistently reproduce/rebuild the software product, and ensure the consistency of the product package with the content expectations for the released product package.				
13	SWIT	Install Testing	Evaluation of install/uninstall testing to verify correct install and uninstall of product.				
14	SWCD	Customer Documentation	Evaluation of product user documentation for accuracy and consistency with product operation.				
15	SWCT	Compatibility Testing	Evaluation of the product under test to perform its required functions while sharing the same environment with one or more systems or components. This includes the categories of binary sharing, data sharing, HW/OS/Environments, and backward compatibility.				
16	SWLC	Legal Compliance	Evaluation of product compliance with legal requirements.				
17	SWCO	Continuous Operation	A method for detecting time-dependant product defects such as memory leaks and race conditions. It is also one indicator of product reliability over				
18	SWRT	Regression Testing	Evaluation of the testing of the product to determine that changes have not caused unintended effects and the product still complies with requirements.				
		Customer	Evaluation of the testing performed to determine whether the product meets				

Product Health | Alpha Checklist | Beta Checklist | Gold Checklist | Usage Instructions

Figure 3. Example Quality Dashboard.

[2] Improper use includes employee performance appraisals, compensation reviews, or promotion evaluations. Misinterpretation includes incorrectly reading graphs, misinterpreting statistical results such as confidence intervals or significance levels, overlooking sampling problems, confusing association with causality, or violating distributional assumptions.

Putting It Together

Near the beginning of a project, a team starts with the standard set of QRC assessments and the quality requirement categories found in the organization's requirements document template. From these, the team can form a complete framework of product quality based on the particular characteristics of the product they must produce.

The quality requirements both influence and are influenced by the quality release criteria (Figure 4). Quality requirements provide target values used to demonstrate the status of the QRC, but the QRC also influence the categories of quality requirements chosen by a team for specification. For example, the continuous operation release criterion influences the product's requirements specification, forcing a team to specify product reliability as one of the requirements for the product. At the same time, the reliability requirement establishes the target values for the continuous operation demonstration that feeds data to the quality release criterion.

The concepts and practices presented above all work in concert to help ensure that product quality is not a matter of chance. By taking a holistic view, product quality will not be a matter of chance, but rather of conscious decisions based on data. This risk-driven, data-driven approach will get problems out in the open early, while there is still time to correct them. The result will be a reduction in rework, improved schedule predictability, shortened time to market, and improved product quality.

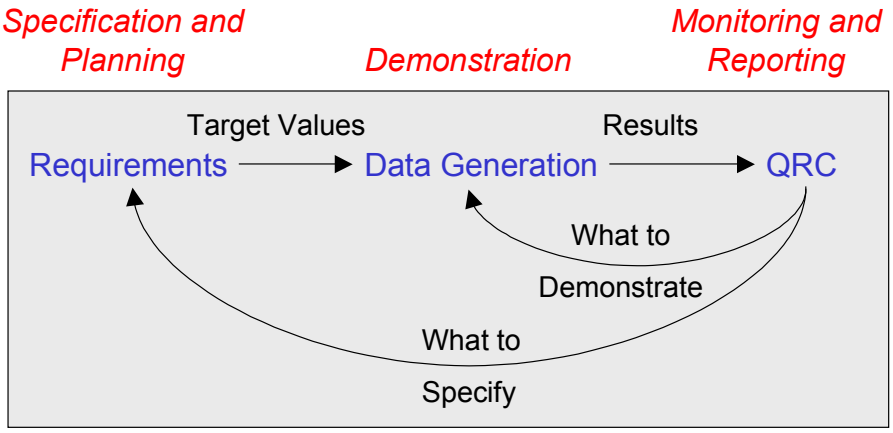


Figure 4. Relationship Between Requirements and Quality Release Criteria.

Example

Suppose that you are the Quality Manager for a new product. The product is a device similar to a television set-top box in appearance, and its primary function is to capture movies broadcast on sidebands of standard TV broadcasts for possible later viewing. The device chooses to capture a movie for the user based on a proprietary algorithm that combines several dimensions, including end user preferences for movie type, the viewer's age, and actors. When a user decides to watch a captured movie, a charge is incurred via a back-channel telephone link to a billing service.

The potential market for the product is large, since it does not rely on proprietary networks or equipment for transmission or billing. The user base has a low tolerance for failure based on experience with television and household appliances. Competition exists, so prices for viewing a

movie will lead to small margins. The product will use the "shaver and blades" model, in which the product itself will be inexpensive and most of the revenue will be derived from movie viewing fees.

The project team's risk analysis shows that the large potential user base, low margins, and user intolerance for failure lead to a significant risk to profitability: support calls. The team estimates that recouping the cost of a single user support call will require a full year of average product usage. A defect in any portion of the product, its documentation, or its usability could have disastrous financial effects.

As quality manager, you want to help reduce this risk during design and development of the product. Several factors might lead to a support call, including installation and setup, daily use, product reliability, and fault tolerance and recovery. By ensuring that the product is simple to install, quick to set up, easy to use, reliable, and self-correcting for as many errors as possible, support calls will be minimized. These issues trace into the QRC assessments as shown in Table 2.

Table 2. Support Call Issues Mapped to QRC Assessments.

Issue Leading to Support Call	QRC Assessments
Ease of Use	Ease of Use
Ease of Installation	Install Testing Customer Documentation
Reliability	Continuous Operation
Fault Tolerance and Recoverability	Functional Testing Defect Data

Let's examine two of the concerns listed above in more detail: reliability and ease of installation.

Reliability

Several factors govern reliability target for the example product. First, there are many types of failure, each possibly having a different target value. The reason for this is that a user base has differing tolerances for different types of failure, based on characteristics such as the effects of the failure, the user's innovativeness, and his experience with technology [Rogers95]. For the class of failures defined as *software defects leading to media playback failure* but no damage to hardware, suppose that market research shows the user base to have a tolerance of one failure per 100 uses. Given a projected average usage duration of 2.5 hours, this translates to a target of 250 hours of failure-free operation (or, alternatively, 100 consecutive uses). Building in an engineering margin of 100 hours to the plan, a traditional reliability requirement might state that the system Mean Time to Failure (MTTF) be at least 350 hours, measured at 95% statistical confidence. However, a good reliability requirement is made up of much more than just the target value, and this is where Planguage is so effective. The Planguage requirement looks like this:

TAG: Reliability.Playback
GIST: The reliability of media playback.
SCALE: Mean Time to Failure for playback-related failures only, using a Representative Operational Profile with randomization.
METER: A Probability Ratio Sequential Test with α=10%, β=10%, and a discrimination ratio of 2.0
PLAN [Gold]: At least 350 hours
MUST [Gold]: At least 250 hours
MUST [Beta]: At least 75 hours
Representative Operational Profile: DEFINED: An operational profile that is likely to occur during use of the system after deployment.

This requirement is verifiable and measurable: the measurement method and targets are both specified. Planguage allows the specification of goals for different milestones (beta and gold). It also allows separation of planned levels of success from “must have” minimums needed to avoid financial, political, or other failure. In this case, a minimum reliability of 75 hours is required before the beta milestone is reached and the product is tested by a broad cross-section of the user base. This level is the minimum judged to protect brand equity, allowing earliest possible product use without damaging the reputation of the company. So, if you can’t demonstrate greater than a 75 hour MTTF, you do not pass the beta quality gate.

The continuous operation quality release criterion captures the reliability requirement and its associated targets:

CO1: The product is able to achieve the specified continuous failure free operation target defined in the Product Requirements Document or Software Quality Plan.

Once product development progresses to the point that we can measure reliability, this criterion will help determine whether the product meets the reliability requirement at all three milestones.

Ease of Installation
Targets for ease of installation must be based on some benchmark value that represents what the typical user would expect (and tolerate) as setup time without calling a technical support line for help. By examining devices such as CD or DVD players, VCRs, or value segment PCs, a reasonable target would be less than 30 minutes, with fewer than 10% of users requiring phone support during the process. Building in some margin, and recognizing that an ideal value might be closer to 10 minutes or less in all cases, we can write the requirement for installation time in Planguage as:

> TAG: Setup.Time
> GIST: The time needed for a user to set up the system.
> SCALE: Mean Time from opening the box to completion of the enclosed instruction set.
> METER: Usability testing with at least 25 focus group subjects in each of the top 3 product demographic groups.
> PLAN [Gold]: No more than 20 minutes at least 80% of the time
> MUST [Gold]: No more than 30 minutes at least 80% of the time
> MUST [Beta]: No more than 45 minutes at least 80% of the time
> WISH: No more than 10 minutes 100% of the time

The targets in this requirement allow progress towards increasingly stringent quality goals.

We can write a separate requirement to cover the proportion of the users requiring a support call to complete the installation:

> TAG: Setup.Calls
> GIST: The likelihood that technical support is required to complete a setup.
> SCALE: Proportion of users having to ask for help at least once during setup.
> METER: Usability testing with at least 25 focus group subjects in each of the top 3 product demographic groups.
> PLAN [Gold]: No more than 5%
> MUST [Gold]: No more than 10%
> MUST [Beta]: No more than 20%
> NOTE: Relates to Setup.DocQuality

The need for a call to technical support during installation relates to the quality of the user documentation, so we need an additional requirement for documentation quality:

> TAG: Setup.DocQuality

GIST: The quality of the installation guide.
SCALE: Proportion of test subjects able to locate the solution to Common Problems in the installation guide.
METER: Common Problems tested on at least 20 focus group subjects in the primary product demographic group.
PLAN[Gold]: At least 95%
MUST[Gold]: At least 80%
MUST[Beta]: At least 60%
Common Problems: DEFINED: The top 3 problems located in installation testing for Setup.Calls

The related quality release criteria for these requirements are:

SWIT1: End-user installation documentation or configuration guide exists, includes install and uninstall procedures, and has been verified for accuracy by testing.
SWCD1: Customer documentation is checked for technical accuracy and consistency against the product.

Monitoring the release criteria associated with reliability, install time, support calls, and end user installation documentation as the product development effort progresses provides a risk-based assessment of product health in the areas needed to minimize support costs. More generally, the summarized release criteria constitute a product health dashboard that can be updated and reviewed at regular intervals throughout the project.

References

Chung00	Chung, Lawrence, Nixon, Brian, et al, *Non-Functional Requirements in Software Engineering*, Klewer Academic Publishers, 2000
Gatlin00	Gatlin, Manny, and Hannon, Gregory, *Managing Software Product Quality*, presented at the Pacific Northwest Software Quality Conference, 2000
Gilb01	Gilb, Tom, *Competitive Engineering*, Addison Wesley 2001 (forthcoming)
Gilb97b	Gilb, Tom, *Quantifying the Qualitative*, available at http://www.result-planning.com
Gilb97a	Gilb, Tom, *Requirements-Driven Management: A Planning Language*, Crosstalk, June 1997
Gilb88	Gilb, Tom, *Principles of Software Engineering Management*, Addison Wesley 1988
Rogers95	Rogers, Everett M., *Diffusion of Innovations 4th Ed.*, The Free Press 1995
Simmons01	Simmons, Erik, *Quantifying Quality Requirements Using Planguage*, presented at Quality Week 2001
Wilson96	Wilson, William, Rosenberg, Linda, and Hyatt, Lawrence, *Automated Quality Analysis of Natural Language Requirements Specifications*, available at http://satc.gsfc.nasa.gov/support.

Appendix A: A Brief Description of Planguage

Planguage (PLANning lanGUAGE) was created by Tom Gilb [Gilb01, Gilb97a, Gilb97b]. Planguage is a keyword-driven language that can be used in requirements specifications, design documents, plans, and other places where qualitative statements are common[3]. Its primary benefits are that it quantifies qualitative statements, improves communication about complex ideas, prevents omissions within quality requirements, and is easy to learn.

Planguage has a rich set of keywords. The commonly used keywords are given in Table 3.

Table 3. Planguage Keywords.

TAG	A unique, persistent identifier
GIST	A short, simple description of the concept contained in the Planguage statement
STAKEHOLDER	A party materially affected by the requirement
SCALE	The scale of measure used to quantify the statement
METER	The process or device used to establish location on a SCALE
MUST	The minimum level required to avoid failure
PLAN	The level at which good success can be claimed
STRETCH	A stretch goal if everything goes perfectly
WISH	A desirable level of achievement that may not be attainable through available means
PAST	An expression of previous results for comparison
TREND	An historical range or extrapolation of data
RECORD	The best-known achievement
DEFINED	The official definition of a term
AUTHORITY	The person, group, or level of authorization

Besides keywords, Planguage also offers several convenient and useful sets of symbols:

- Fuzzy concepts requiring more details are marked using angle brackets: <fuzzy concept>
- Qualifiers, which are used to modify other keywords, are contained within square brackets: [when, which, …]
- A collection of objects is indicated by placing the items in braces: {item1, item2, …}
- The source for a statement is indicated by an arrow: Statement ← source

[3] The term Planguage is also used as the name of some programming languages for parallel processors, but that use is not related to its use in this paper.

Gathering Requirements is like pulling teeth!

Debra Schratz

Abstract

The hardest part of building a product is deciding exactly “what” to build. Why is this true? One big reason is that Product Development Teams (PDT) typically do not plan elicitation efforts very well.

This paper is designed to assist development teams plan elicitation efforts by outlining the steps to plan an effective requirements gathering process for a project. Events associated with planning the elicitation effort are discussed. An elicitation plan template, real life examples, and sources for more information are provided.

If gathering requirements has been like pulling teeth, this paper will take the pain out of planning eliciting requirements.

Biographical Sketch

Debra Schratz has 5 years experience in quality engineering. Debra currently works as a Platform Quality Engineer in the Platform Quality Methods group, part of the Corporate Quality Network at Intel Corporation. Since January 2000, Debra has delivered more than 150 training classes on requirements engineering for Intel worldwide. Prior to her work in quality, Debra spent 8 years managing an IT department responsible for a 500+ node network for ADC Telecommunications. Debra is a member of the Rose City SPIN Steering Committee. She holds a Bachelor's of Arts degree in Management with an emphasis on Industrial Relations.

Email address: Debra.Schratz@intel.com

Mailing address: 2111 NE 25th Ave, Hillsboro, OR 97124

Gathering Requirements is like pulling teeth!

The hardest part of building a product is deciding exactly "what" to build [Brooks87]. Why is this true? One reason is that Product Development Teams (PDT's) [1] typically do not plan elicitation efforts very well.

Why don't teams plan their elicitation effort? PDT's are faced with several challenges. For example, growing product complexity, intense schedule pressures, compressed cycle times (the time to get products to market), dispersed development teams, physical distance from the customer, and the general feeling "if you are not coding, you are not doing anything!" All these challenges contribute to pressure to begin developing the product before requirements are collected.

Myths associated with planning a good elicitation effort

In a "perfect" world, teams would ensure feasibility, commit to the project, and consistently would have predictable results. In the "real" world, PDT's get excited about the project, commit to a schedule, try to make it work, and are surprised when they have unpredictable results. Three "myths" contribute to a lack of planning:

Myth #1: We know what we are supposed to build, so why spend the time planning? Without a solid plan to elicit requirements from customers, the resulting product will be an "engineering" centered product that is likely not to meet the needs of the customer. Poor customer identification causes teams to develop products that don't reflect a wide-range of customer needs, so they miss a large piece of the puzzle. If this happens, consumers will go elsewhere to purchase products that have the features they want and need. Only the products that are developed with the customer's needs adequately and accurately defined will succeed.

In other cases, a development team will have a good idea what the customer wants, but lacks vital information from other teams such as test, manufacturing, quality, and customer support. The result is an elicitation effort that does not include all necessary stakeholder groups and ultimately doesn't meet the needs of everyone with a material interest in the product.

Planning an elicitation effort allows a team to uncover what exactly must be built, which will reduce rework and shorten the overall project schedule [Blackburn96].

[1] In this paper, "Product Development Team" (PDT) is a cross-functional team with representatives from many disciplines who are responsible for developing a product from concept through development.

Myth #2: We don't have enough time to create a plan, we are already behind! Many software projects have intense time constraints and limited resources. Compounding this problem, key team members can be pulled away from the elicitation effort to put out "fires" that have flared up on a recently released product. This results in lower requirements quality and schedule delays.

At some point, a PDT will figure out that planning for the elicitation activities would have been a good idea, but it is too late in the project to do anything about it. Milestones and deliverable dates arrive and the teams push forward with what they have, for better or worse.

The right time to begin planning is as soon as the cross-functional requirements team forms. As the team identifies the stakeholders, objectives, strategy, and deliverables associated with gathering the product requirements, time and resource constraints become visible and can be dealt with. The elicitation plan can be used to gain commitment from management on schedule and resources for requirements gathering. When teams feel they don't have enough time to plan, that's when they need it the most!

Myth #3: We have to show management we are "doing" something. PDT's often jump into coding so they can show they are moving the project forward. Management wants to see results, and too often in software projects this means source code**.**

Writing code is much more fun for most programmers than planning elicitation objectives! Most managers don't see the value of planning [Hooks01] and don't allocate the time to do it right.

Many teams use concurrent engineering as a way to decrease the time it takes to get products out to customers. This can be very effective if done properly (keeping the requirements at least a step ahead of development), and can be very dangerous if done improperly (implementation starting before product scope is defined and stable).

Why plan?

A well-planned elicitation effort results in better requirements. This has several major benefits to the project, including:

- ✓ Reduced scope creep
- ✓ Reduced re-work
- ✓ Decreased time to market
- ✓ Improved product quality
- ✓ Reduced risk
- ✓ Improved schedule predictability

An effective elicitation effort recognizes that development is a collection of concurrent activities rather than a rigid 1-2-3 sequence. Planning the elicitation activities allows teams to start development on those items that are relatively stable so the team can minimize rework. Planning also improves synchronization between requirements gathering and other activities such as prototyping and architectural specification.

When teams spend adequate time in the planning phase, they spend less time in the testing and integration phases of the project, reducing time to market [Blackburn96].

An effective elicitation plan includes a summary of the market and business segment, an overview of the problem or opportunity they are trying to solve, a review of the system domain, and a basic understanding of the needs of their stakeholders. The elicitation plan provides the information a team needs to execute the elicitation process. A basic elicitation process includes:

1) Establishing a cross-functional requirements team early in the exploration phase of the project that is responsible for gathering the product requirements. This team will work together to elicit, validate, specify, and verify the requirements
2) Brainstorming the sources of requirements. PDT's should spend time considering where the important product requirements will come from. Common sources include people, previous products, competitive products, company initiatives, research and development, and market data.
3) Identifying stakeholders and techniques to elicit requirements. This information is used to plan and estimate the amount of effort and schedule that will be needed to gather a complete set of requirements.
4) Performing and documenting the elicitation activities

To meet the needs of such a process, an elicitation plan should contain:

I. Introduction

- Problem Statement/Opportunity:

  A concise summary of the problem statement or opportunity for the product. This generally is translated into the product's scope and vision when combined with data from the Market and Business Context.

- Market and Business Context:

  Market and Business Context – A description of the market and how it interacts with the business goals for the product.

- System Domain:

  System Domain – the environment and conditions in which the system operates. For example, if you wish to build a mobile communications product, you must understand the environments and conditions in which it will be used. To build an automotive system, you must understand the car industry, the manufacturing process, and how cars are used

- Definitions:

  Defines terms and acronyms used in this project.

II. Assumptions

Identifies the assumptions made that are outside the influence or control of the development team. Each assumption should translate into a risk in the Risks section.
Since assumptions by nature tend to be 'big deal' items that often are outside the influence or control of the product development team. They are also an explicit admission of missing information – they would not be assumptions if all were known.

III. Elicitation Objectives

Contains the high-level objectives of the elicitation effort. Examples include validation of market data, gathering end user requirements, creating use cases, etc.

IV. Strategy & Process

What strategies and process will you employ during the effort? Customer visits; group interviews, observation, and surveys are some examples.

V. Products & Outputs of the Elicitation Effort

State the expected deliverables of the effort. For example, draft use cases, completed customer surveys, etc.

VI. Stakeholder Description

The Stakeholder Description is best written in an Actor (who) – Goal (why the actor cares) format, using a two-column table or similar format. This description can then be used not only to guide the elicitation effort, but also as input to use cases for documenting the results of the elicitation.

Stakeholder	Goal

VII. Techniques and Format (by stakeholder group)

For each stakeholder group or actor, state the techniques that will be employed during the elicitation and the format used to capture data. Example: End Users will be interviewed, with data captured on audiotape and written notes.

Stakeholder	Technique	Format

VIII. Schedule & Resource Estimates

Estimate the schedule and resources needed for the elicitation effort. Include a measure of uncertainty for your estimates, such as +/- or a range of values.

Item	Estimated Schedule	Resources	Range of uncertainty

IX. Risks

What risks can you identify for the elicitation effort? Assumptions are a good place to start, but focus on schedule risks, resource risks, and risks to the quality of the products of the effort. Identify the type, possible effects, magnitude of risk, likelihood, and a mitigation strategy.

Risk ID	Description	Mitigation/ Management Plan	Time Risk Estimate	Prob. Of Occurrence	Exposure

Common Problems

There are three common problems with elicitation plans:

1) **Going beyond the scope of the elicitation effort**. A good elicitation plan identifies only the activities associated with the elicitation effort. Teams need to constantly evaluate the plan for scope. Ask yourself: Does the plan focus on just the elicitation activities and not the WHOLE project? When teams keep the scope to elicitation only, completing the plan is easier.

2) **Inaccurate resource, time, and effort estimation for elicitation activities.** There are two sources of inaccuracies: including activities outside of the elicitation effort, and omitting necessary activities. Ask yourself: Does the elicitation plan accurately estimate the activities associated with gathering requirements? Don't forget to look at past projects to facilitate and confirm estimates.

3) **Missing important stakeholders**. Most teams identify the customer and end user as stakeholders, but other groups are often overlooked. Ask yourself: Are we missing any important stakeholders? When all stakeholders are identified, it is easier to plan the elicitation activities needed to understand each stakeholder's viewpoint.

How do I get started?

Gathering requirements won't be like pulling teeth if teams start by:

- Creating an elicitation plan for the project that is focused just on the elicitation activities to minimize scope creep
- Using the elicitation plan to gain explicit management commitment of schedule and resources for requirements gathering
- Identifying assumptions, risks, schedule, effort, and resource estimates for the elicitation efforts (not the whole project) to improve product quality and shorten time to market
- Planning for contingencies that may disrupt the elicitation efforts in order to minimize their impact

- Gaining consensus on resource commitments and priorities before the elicitation planning gets underway
- Considering all stakeholder groups when planning the elicitation effort to minimize re-work.

Sources for more information

Below is a list of sources for more information on how to plan for a successful elicitation effort in your organization:

- Leffingwell, Dean and Widrig, Don, *Managing Software Requirements: A Unified Approach*, Addison Wesley, 2000.
- Hooks, Ivy and Farry, Kristin A., *Customer-Centered Products, Creating Successful Products through Smart Requirements Management*, Amacom, 2001.
- Wiegers, Karl E., *Software Requirements*, MS Press, 1999.
- Hoffman, Hubert F. and Lehner, Franz, *Requirements Engineering as a Success Factor in Software Projects,* IEEE Software, July/August 2001.

For an example stakeholder checklist:

- Chastek, Gary, Donohoe, Patrick, Kang, Kyo, and Thiel, Steffen, *Product Line Analysis: A Practical Introduction,* CMU/SEI, and June 2001.

References

Brooks87	Brooks, Frederick P. Jr. *No Silver Bullets - Essence and Accidents of Software Engineering,* Computer, April 1987
Blackburn96	Blackburn, Scudder, et al, *Improving Speed and Productivity of Software Development: A Global Survey of Software Developers*, 1996
Hook01	Hooks, Ivy and Farry, Kristin A. *Customer-Centered Products, Creating Successful Products through Smart Requirements Management*, Amacom, 2001

Appendix A – Example Elicitation Plan

Elicitation Plan

Internet & Multimedia Tablet Project (IMT)

Revision 0.1
01/12/01

Primary Author: Joe Smith

Key Contributors: Billy Smyth, Bill Poole, Eddie Stone, and Stan Smith

1.Introduction

Problem Statement/Opportunity:

Design a seat-back Internet & Multimedia tablet to be deployed in commercial aircraft.

The initial concept for the product includes a variety of functions:

- Displaying safety videos and other push programming on an interrupt basis
- User-selectable movies and special interest programming
- Email
- Stock quotes
- Ticket sales & reservations
- Rental cars, hotel bookings
- Current flight statistics and connecting flight data
- Individual and multi-player games

Market Context:

The target markets are the airlines (Delta, United, TWA, etc.) and aircraft manufacturers (Boeing, Airbus) who have expressed interest in offering email, stock quotes, and multi-media entertainment to their passengers.

System Domain:

It is important we understand the airline industry, FAA regulations, wireless networking, ISP limitations, user needs, other entertainment devices currently offered on flights, and the manufacturing process of commercial aircraft, including the subcontracting relationships between primary manufacturers and suppliers.

Definitions:

Term	Definition
FAA	Federal Aviation Administration
ISP	Internet Service Provider

2.Assumptions

Assumptions made regarding this elicitation project:

- ✓ Access to passengers will be easy
- ✓ Access to the airlines may be difficult

3.Elicitation Objectives

The objectives of the elicitation plan are:

- ✓ Validate function list
- ✓ Gather end-user (passenger) requirements
- ✓ Create Use Cases to capture high-level functional requirements
- ✓ Document, validate, and prioritize the scope of the project

4.Strategy & Process

The overall strategy is to locate one manufacturer and at least one airline that is a customer of that manufacturer to function as representatives for the other stakeholders in those groups, and concentrate our efforts on that subset.

The basic process will be a combination of interviews and observation, with validation occurring as soon as possible after the initial information gathering activity. Validation will consist of written synopses of interviews and observation sessions sent to participants and key stakeholders for corrections and additions.

Once enough information has been validated, use cases will be created for all stakeholder groups, but the focus will be on the end user first.

5.Products & Outputs of the Elicitation Effort

The outputs of this plan are:

- ✓ First draft of high-level functional requirements in the form of Use Cases
- ✓ Completed customer questionnaires/surveys
- ✓ Validated market data
- ✓ Completed end-user requirements
- ✓ Detailed scope of the project

6.Stakeholder Description:

Actor	**Goal**
Passenger (end user)	Check Email

7.Techniques and Format (by stakeholder group)

Stakeholder	Technique	Format
Passenger (end user)	Interview	Written notes/audio tape

8.Schedule & Resource Estimates

The elicitation schedule and resources are estimated below:

Item	Estimated Schedule	Resources	Range of uncertainty
Complete 1st draft Use Cases	WW8	Billy Smyth, Bill Poole, Eddie Stone	+/- 2 weeks
Complete customer surveys	WW12	Stan Smith, Billy Smyth	+/- 4 weeks
Validate market data	WW13	Eddie Stone	+/- 3 weeks
Completed end-user requirements	WW15	Billy Smyth, Bill Poole, Eddie Stone, Stan Smith	+/- 2 weeks
Detailed scope	WW15	Billy Smyth	+/- 4 weeks

9.Risks

The risks associated with the elicitation efforts are estimated below:

Risk ID	Description	Mitigation/ Management Plan	Time Risk Estimate	Prob. Of Occurrence	Exposure
R1	Access to passengers is more difficult than anticipated	Coordinate closely with airlines to determine passenger accessibility. MUST: Conduct 5 passenger focus group studies. PLAN: 10 focus group studies. WISH: 15 focus group studies	4 weeks	50%	3 weeks
R2	Access to airlines is limited	Discuss the goals, objectives, and priority of this project with the airlines and make adjustments to the scope of the product. MUST: 2 airlines participate in ethnographic interviews. PLAN: All airlines participate	3 weeks	33%	2 weeks

Asynchronous Learning Networks for Software Engineers

Richard H. Lytle
Oregon Master of Software Engineering
Lytler@omse.org

ABSTRACT

Asynchronous or anytime/anywhere learning is expanding rapidly in many areas of education and training, but it is still rare in the engineering disciplines. This paper explores the potential of asynchronous learning for software engineering education and training. The paper addresses the following topics: (1) Importance of asynchronous communication for increased access to learning; (2) The experience of asynchronous learning by students and faculty; (3) Evaluation of asynchronous learning.

BIO

Richard Lytle, Director of the Oregon Master of Software Engineering program (OMSE), draws on personal experience and involvement with many asynchronous learning programs supported by the Alfred P. Sloan Foundation. Dick currently is teaching software engineering at OMSE using asynchronous technologies. Previously he was Dean of the College of Information Science and Technology at Drexel University, Philadelphia, and chief information officer at the Smithsonian Institution.

Asynchronous Learning Networks for Software Engineers

Richard H. Lytle
Oregon Master of Software Engineering
Lytler@omse.org

ABSTRACT

Asynchronous or anytime/anywhere learning is expanding rapidly in many areas of education and training, but it is still rare in the engineering disciplines. This paper explores the potential of asynchronous learning for software engineering education and training. The paper addresses the following topics: (1) Importance of asynchronous communication for increased access to learning; (2) The experience of asynchronous learning by students and faculty; (3) Evaluation of asynchronous learning.

Richard Lytle, Director of the Oregon Master of Software Engineering program (OMSE), draws on personal experience and involvement with many asynchronous learning programs supported by the Alfred P. Sloan Foundation. Dick currently is teaching software engineering at OMSE using asynchronous technologies. Previously he was Dean of the College of Information Science and Technology at Drexel University, Philadelphia, and chief information officer at the Smithsonian Institution.

1. What are Asynchronous Learning Networks (ALN's)?

The Alfred P. Sloan Foundation, America's most prestigious private engineering foundation, has sponsored a large number of ALN projects over the past seven years, including engineering and other technical disciplines from major universities. The Oregon Master of Software Engineering, with funding from the Sloan Foundation, has launched ALN software engineering courses that are available now.

The Sloan Foundation's definition of ALN's is:

> **Asynchronous Learning Networks** (**ALN's**) are people networks for anytime - anywhere learning. ALN's combine self-study with substantial, rapid, asynchronous interactivity with others. In ALN's, learners use computer and communications technologies to work with remote learning resources, including coaches and other learners, but without the requirement to be online at the same time. The most common ALN communication tool is the World Wide Web. **(www.sloan-c.org)**

The most important and unexpected part of this definition is that ALN's are *networks of people.* This is not hype! Across many Sloan Foundation learning programs, most ALN students report higher levels of interactivity among themselves and with faculty, compared to face-to-face courses. ALN faculty report similar results. The interactivity derives from the asynchronous threaded discussions, which provide students and faculty more time to reflect on an issue, consult other sources, and contribute with much more deliberation than is possible with synchronous technologies or face-to-face classes. Students and faculty in OMSE ALN courses report these same effects.

In fact, ALN's are not strictly "anytime" learning environments. Rules of the road are necessary to ensure that people in the network create enough asynchronous contribution cycles to support in-depth discussions. For example, a course may require that students contribute to discussions for 30 minutes, on any four days of the week. They can contribute at any time of day on those four days.

Another important advantage of ALN's is that they give students an opportunity to learn when they cannot attend classes at the same time or the same place. To reach these students, ALN courses usually do not require any face-to-face or synchronous online attendance.

Sometimes ALN's are confused with other approaches to learning or with specific technologies. ALN's are not self-paced learning, in which the learner can go through a course at any chosen rate, because the essential core of an ALN is the people-network. On the other hand, ALN's can contain technology-based training tools, simulations, elaborate multimedia productions and the like, as well as textbooks and other readings. But these technologies do not provide the power of ALN's.

Having said all of this, strict asynchronous learning is not a universal solution. Rather, ALN's are an approach to learning that is very widely effective, but may not be effective for every learning need, business activity, or individual learning style. For example, in the ALN software engineering program at Drexel University in Philadelphia, most courses involved group projects. Students found that they needed telephone conferences to get projects started--especially to determine project vision--but that they used asynchronous communication for most of the project work. Project groups were established at the beginning of the course to ensure that each group of students could schedule limited synchronous communications.

Some courses combine ALN's with face-to-face meetings. Although there are applications where this approach works well, requiring face-to-face meetings may exclude many learners who have severe time or place constraints. OMSE never requires students to gather at the same time or the same place for ALN courses.

2. Why are ALN's a good fit for software engineers?

The primary objective of this paper is to alert software engineers to the promise of ALN's. Despite their comfort on the web, many engineers are very conservative in their approach to learning environments. If face-to-face lectures are not available, engineers may regard lectures delivered by synchronous video as the only good substitute. So, for those engineers who find ALN's to be counter-intuitive, we ask you to take a close look.

Software engineers are good prospects for ALN's because:

- Software engineering is an applied field in which engineers are synthesizing best practices from published sources, knowledge of fellow engineers, and their own experience. As described above, ALN's provide the substantial interactivity with other people that engineers need to consult, learn and collaborate.
- Software engineers are time-constrained at work and in their personal lives and thus have difficulty attending face-to-face or synchronous video courses.

3. What is it like to learn via ALN's?

In the presentation version of this paper, we use a video that gives the viewer a good sense of what it is like to participate in an ALN course. Since we cannot use the video in this printed paper, we offer OMSE's "Guidelines for ALN Students," Table 1; and "Excerpts from Discussion Threads," Table 2.

If you contacted OMSE asking for information on ALN courses, you would receive an email with the following content (see Table 1). Please read it, pretending that you are thinking about taking an ALN course.

Table 1
Oregon Master of Software Engineering
Guidelines for ALN Students

- Access to software engineering courses on a more flexible schedule is the obvious advantage of taking OMSE courses via the ALN. But ALN's are different from face-to-face classes, and you will need to recognize and adjust to these differences in order to succeed.
- We will never require you to get together as a class at the same time or the same place. Our delivery technology will provide synchronous chat rooms, but you will not be required to use them in the course.
- Asynchronous discussions have very explicit rules. For example, you may be expected to spend at least 30 minutes on each of four different days--at any time during those days--contributing to asynchronous threaded discussions. This requirement assures that there are enough cycles to support in-depth discussion for all students.
- Our ALN's curriculum, instructors and students have the same quality compared to face-to-face classes.
- Your learning will be equal to face-to-face courses, and the time you spend "in class" and doing homework will be approximately the same as face-to-face courses.
- Our delivery technology and technical support for students will be provided on a 24/7 basis by eCollege (www.ecollege.com).
- Faculty, the OMSE Director, and OMSE staff will be available for course and student support issues. Faculty will establish ground rules for email communications.
- You will receive regular feedback about your progress in the course.
- While ALN courses may use various forms of multimedia, in the main students will use texts and publications, web sites, annotated Powerpoint files and discussion threads.

Newcomers to ALN's will find the course structure and study materials similar to face-to-face courses. At OMSE, ALN courses follow the same weekly learning units as do face-to-face courses, textbooks are the same, and the faculty member usually teaches the same course face-to-face. The two significant differences are the absence of face-to-face lectures and the people-interaction through threaded discussions. For most students and faculty, intensive interaction on the people network more than makes up for the absence of face-to-face lectures.

Faculty members also have a different role in ALN's compared to face-to-face courses. The most important faculty role focuses on the people-centered networking that is the core of ALN's.

- The faculty member formulates the discussion questions to promote and guide discussion. This is a very important function, in which the stage is set for learning.
- Once the course is underway, the faculty member becomes more a facilitator than a "sage on a stage."
- Because OMSE students are experienced software professionals with much practical knowledge to share with each other, the faculty should guide discussions in a way that maximizes student-to-student interaction. The faculty member keeps discussions on target and brings overlooked perspectives into play.
- The faculty must establish a precise schedule for feedback to students on papers and grades, and adhere to that schedule. ALN's need this structure even more than face-to-face classes.
- At a minimum, the faculty member will ensure that the learning via the ALN course is "at least as good" as its face-to-face delivery.

4. An example of ALN threaded discussions

Another way to gain a second-hand view of ALN's is to read a typical discussion thread. Keep in mind that it is entirely different to participate in a discussion than to read a transcript of that discussion. Table 2 contains a snippet from a weekly discussion on the threads for OMSE 500 ALN, Principles of Software Engineering, Winter 2001. In this course, nine students contributed about 100 times each week to the asynchronous discussions.

Table 2 Part 1
Excerpts from OMSE 500 Discussion Thread
Winter 2001

Richard Lytle (Facilitator) | 2.22.2001 3:02:02 pm

QUESTION: How do you know when you are done testing?

RESPOND TO THIS TOPIC | EDIT TOPIC (ADMIN ONLY) | DELETE TOPIC (ADMIN ONLY)

Sam's original post to the question

From my experience, this has always been an inexact science. Not trying to sound flippant, but to some extent its simply when it's time to deliver, or close the environment. I've seen this often determined purely by business reasons. For the types of stuff I do, it's based on a few loose criteria. 1) Confidence in typical workflow capability. Through monitoring of standard regression test suite, and interactive tests there's a confidence that 'conventional' use cases are handled. (2) Incoming defect rate. Some rough measure of a decline in defect rate that is deemed acceptable. 3) Customer reaction. Overall, would they rather have what's there, or have us continue to test/fix. Beyond that in my work experience it's always been as much a political decision as anything. As a program or project manager it's somewhat of a balancing act that includes schedule targets, additional functionality, and outstanding defects. So in a way, this has made things easier from a software engineering perspective.

Table 2 Part 2
Excerpts from OMSE 500 Discussion Thread
Winter 2001

Richard Lytle (Facilitator) | 2.22.2001 3:02:02 pm

QUESTION: How do you know when you are done testing?

RESPOND TO THIS TOPIC | EDIT TOPIC (ADMIN ONLY) | DELETE TOPIC (ADMIN ONLY)

DISCUSSION THREAD CONTINUED

John's reply to Sam

The three conditions you cite are very similar to the ones we use. This seems to be a practical way to deliver working software timely and at a reasonable price. While this may not be an acceptable point to stop testing for software where a failure could lead to the loss of life, it brings the testing process to a close.

Sam's reply to John

I'm glad I'm not responsible for life dependent software. I make enough stress for myself as it is.

Sam's second post to the question

One other factor that plays in to our situation is history. We inherited something that was in such bad shape both functionally and quality wise, that with the appropriate resource, it really wasn't hard to make it better. As counter intuitive as it may seem, we have better quality now than 2-3 years ago and yet we find more bugs. Why? Because we have more functionality (both new and overhauled), in which to find defects, and the expectations of everyone, development, end users, is much higher. As an example, we had a particular tool-path generation operation a couple years ago that really had only one defect against it - 'it doesn't work'. That's all anyone needed to know. No one used it. We rewrote it and now we have a number of reports against specific geometric situations that it doesn't handle well. We've got more bugs now and a better quality product. Unfortunately, it has been difficult to get upper management to see this.

5. How do we assess the quality of ALN-based learning?

How do we approach the issue of quality in any learning method or technology? Very carefully and very critically! We must keep in mind that the quality of any graduate professional program fundamentally depends on the quality of the curriculum, the faculty, and the students. ALN's or any other delivery means is not the primary driver for quality education and training. Since this article is no place for an extensive foray into educational research, three glimpses of the ALN quality issue must suffice:

- A convincing review of educational research by Thomas L. Russell reports "no significant difference" in learning based on the use or non-use of any technology. (The No Significant Difference Phenomenon, 1999, North Carolina State University). The research survey examines 335 studies over 70 years. We can conclude from this extensive review of research that there is no evidence that ALN's or any other delivery technologies have been shown to increase or decrease the quality of learning.
- ALN's supported by the Sloan Foundation have created an extensive community for assessment of ALN providers, including practical ways to ensure quality. The Sloan Foundation's minimum ALN standard is "at least as good" as face-to-face classes. A large proportion of students and faculty in Sloan-supported projects testify to the "at least as good as" standard. (www.aln.org).
- The last source of information about quality is student course evaluations from the Winter 2001 offering of OMSE 500ALN. Student evaluations are an integral part of the OMSE course quality program for face-to-face and asynchronous delivery. Student identity is not revealed to OMSE faculty. See Table 3 for student comments. Readers may request soft copies of the full student assessment by email (lytler@omse.org).

Table 3
Students' Summary Evaluations of
OMSE 500 ALN Winter 2001

The distance learning option makes the classes convenient. I also like the format and believe that I learned as much or possibly considerably more than I would have learned in a classroom environment.

The discussions were extremely valuable and sometimes demanding to provide responses to the other posts by students.

I don't feel I have any similar experience to compare it to, so I can't rate it on a comparison basis. From an expectations perspective I'd probably give it an 8 out of 10.

This is my first (ALN) class and I really enjoyed the class. Rating: 9 of 10. It could have been a 10 if the above suggestions (recorded lectures) are made available!

Many of the discussion questions were very provocative. I enjoyed (the instructor) contrasting differing discussion responses and driving to the reasons for those differences.

(The instructor's) indecisiveness was problematic. He seemed to judge discussion participation by purely number of responses, not quality.

The discussions were guided and moderated quite well. The flexibility offered and

6. Do you want to learn more about ALN's?

The best source of information about ALN's is provided by the Alfred P. Sloan Foundation. At www.sloan-c.org, you will find course offerings by members of the Sloan ALN Consortium, an organization that certifies ALN programs against Sloan quality standards. At www.aln.org, you will find Sloan-sponsored publications that are mainly aimed at universities and other learning organizations, but also deal with issues of importance to corporate and other potential users of ALN's.

Experience With A Process For Competitive Programming

Bart Massey
Computer Science Department
Portland State University
bart@cs.pdx.edu

A lightweight software development process was developed for the unusual programming environment of the Association for Computing Machinery International Collegiate Programming Contest. This process was used by teams from Portland State University over the course of two years, with excellent results: the process was judged to be effective in improving the performance and satisfaction of student programmers. The process modeling techniques and process elements used should be portable to a wide range of industrial domains for which standard methodologies are inappropriate, especially to smaller teams and projects.

Bart Massey received a B.A. in Physics from Reed College, then spent several years as a Software Engineer III in the Television Measurement Systems Division of Tektronix, Inc. Returning to school to study programming languages and artificial intelligence, Bart received his Ph.D. in 1999 from the University of Oregon. He has spent the last few years teaching at Portland State University and in the Oregon Master of Software Engineering program.

1 The ACM ICPC

The Association for Computing Machinery (ACM) has conducted its International Collegiate Programming Contest (ICPC) [1] on an annual basis since 1970. In 2000, more than 2,400 teams of students were involved in this competition at the regional level, making the 2000 ICPC arguably the largest organized student programming experience in history.

Internationally competitive student ICPC teams tend to be composed of the best and brightest programmers at the top-ranked schools in the world. This tendency is enhanced by the unusual structure of the competition, little-changed since the 1970s. In brief, a team of three students (at most one having an undergraduate degree) shares a single terminal over a period of around 8 hours. The team is given 6 or more "brief" requirement specifications in a stylized format at the start of competition. They are then expected to cooperatively analyze the problems, select the most tractable (there is rarely sufficient time for even the top teams to solve all the problems) and to produce programs which will produce correct answers given blind test inputs. Scoring is by number of problems solved, with ties broken using a combination of time-to-solution and number of incorrect submissions.

2 Teamwork and the ICPC

The ICPC contest format is supposed to encourage team-based problem solving, but it often has the opposite effect. In practice a single expert programmer, though monopolizing the terminal and receiving little or no support from teammates, is often more effective than a coherent team. Like professional athletic teams or small industrial software development teams, a dominant player can make the team effort a one-person show. There are several reasons why the ICPC tends to exaggerate rather than minimize this problem [2].

First, the well-known wide gap in productivity between programmers [5] appears to hold in this situation. Since the contest time period is so short and access to the terminal is a fairly effective bottleneck, this productivity gap is amplified: the effectiveness of the programmer at the terminal tends to limit the progress of the team. Thus, on a competent team, an exceptional programmer will dominate programming activity.

Second, the usual problems of effective programming team communication are exacerbated by the contest conditions. While software engineering communications problems have been appreciated at least since the mid-70s [5], the ICPC's short time scale and limited access to computing, display, and typographic resources all aggravate the situation. Currently, students have been encouraged by modern programming environments and practices to think online and share their thoughts via e-mail and the web: this leaves them ill-prepared for the high-pressure isolated pencil-and-paper problem solving environment of the ICPC.

Third, the methods of contest preparation and organization commonly used by the student teams and their advisors tend to encourage individualization rather than teamwork. For starters, it is common for an institution to select participating team members based on their placement in an individual competition under contest conditions. While this method undoubtedly helps select those with a certain skill set, it is likely to select against those who work best cooperatively. Further, there appears to be a strong tendency among coaches to emphasize parallel problem solving during the competition: this typically manifests as a suggestion that the problems be divided up among team members, with each team member working on a problem independently [6]. Again, while the benefits of this approach are obvious, the less obvious downside is that a team member's attempt to elicit help from or offer advice to another team member may be treated as a source of inefficiency and actively discouraged.

3 A Process Model For ICPC Preparation and Participation

In the Fall of 1999 the author was charged with preparing teams from Portland State University (PSU) in Portland, Oregon for the Pacific Northwest (PNW) Regional ICPC. Recent PSU experience with the Regional ICPC competition had not been good: the two 1997 teams had solved one problem each, and in 1998 one team solved two problems while the other failed to solve a single problem. This was particularly disappointing given that at least one problem in a typical year is designed to be especially easily soluble. In addition to a demoralizing past, the time available was short: work began about one month before the competition date.

Finally, the quality of the students was fairly uniform: while all of the contestants were quite competent, there was no single "superstar" programmer. (This is a common situation in industrial software development as well: extraordinary performers are scarce and expensive. Approaches that work well in this setting should thus be useful in the industrial environment as well as the academic one.)

Given these constraints, a new approach to contest preparation and execution was needed. The approach selected was predicated on achieving significant improvement quickly using a uniform talent pool. An informal software development process was sketched out, with the goal of allowing (and indeed enforcing) effective teamwork under contest conditions. The major components of this model were:

- A structured process for evaluation and ordering of problems to be attempted. Teams were instructed to
  - Read through each problem as a group.
  - Discuss any ambiguities, incompleteness, or inconsistency in the problem specification.
  - Discuss the general approaches that might be used to solve the problem.
  - Sketch a high-level architecture for a solution program.
  - Estimate (in an *ad hoc* fashion) time and lines-of-code for each architectural block, and then sum these estimates to achieve an estimate for the entire program.
  - Find whether there was a "champion" for the problem on the team; someone excited about attacking it.

  Using this process, the problems were to be ordered (by team consensus, with the team captain having veto power) and tackled sequentially. Based on estimates by the coach and experience during practice sessions, the eventual rule of thumb was that this portion of the overall process should take roughly 30-60 minutes.

- A structured process for tackling each problem. The three team members were instructed to apportion the roles of *requirements analyst*, *implementer*, and *tester* among them. The requirements analyst was instructed to carefully analyze the requirements and design, and was also charged with proofreading and assisting with coding. The regional contest rules allowed at most two team members at each terminal simultaneously: the third team member was excluded. This third team member, the tester, was charged with developing black-box and gray-box system test cases from the information available, and delivering these at the appropriate time. Only when all test cases passed, and all team members concurred that the program was complete and correct, was the program to be submitted for judging.

An instructional approach to contest preparation was developed concurrently. There were several components deemed important to emphasize:

Problem Classification: ICPC problems tend to fall into a small number of recognizable categories. Some of these categories contain a large percentage of deceptively difficult or time-consuming problems.

Others require specialized knowledge or approaches (that the students may not possess) to solve effectively. Some time was spent training the students to classify problems and re-evaluate their difficulty.

Similar situations arise in anticipating problematic components and techniques in small industrial software development projects. One of the most important contributions of an experienced developer is the ability to recognize and leverage these problem patterns: team training is a key component of this.

Requirements Specification: An ICPC "programming problem" description is invariably an English-language requirements document. The quality of construction and proofreading varies widely: a good rule of thumb is that the problem requirements must always be considered carefully. Some effort was made to teach students to look for ambiguous, incomplete, or contradictory requirements, and to explain techniques for dealing with these defects under contest conditions.

The successful deconstruction of requirements documents is a skill that any good software development team must possess. While the ICPC "requirements document" is somewhat idiosyncratic in structure and form, its principal defects are shared with the majority of requirements specifications encountered in industrial situations, and the approaches to detecting and correcting these defects are quite similar.

Debugging: Incredible as it may seem, in the author's experience even highly effective student programmers generally evince little knowledge of effective debugging techniques and practices. Indeed, it seems plausible that the often-deplored rapid "edit-compile-crash" cycle common among student programmers results less from the ease of access to high-speed editing and compilation than from simple lack of education in effective debugging practice. The students were taught the rudiments of a formal debugging methodology, and observed under contest conditions to evaluate their effectiveness in the use of this methodology.

The lack of methodology and expertise in debugging does not magically disappear once a student graduates. Given the crucial importance of this skill to the implementation and testing phases of software development, more attention should be payed in both academia and industry to education about effecting debugging process.

Testing: Because the contestants are penalized for incorrect submissions, it is important to submit a program that works correctly the first time. Further, because the requirements are often problematic and no suite of adequate test cases is provided as part of the contest materials, on-line integration and system test development is absolutely crucial to successful detection and diagnosis of program defects. The participants were instructed in the basics of informal test construction and evaluation [7].

Small team software development projects usually lack the resource of an effective independent testing group. In this situation, it is highly important that the implementors construct effective tests, and use a systematic test execution and evaluation process. The benefits of early defect detection and high-quality error information should more than pay for the extra effort.

As important as the emphasis on some phases of the software lifecycle is the de-emphasis of others. Of particular note is the near-total lack of attention to actual implementation. What instruction there was in this regard focused principally on fundamentals of readability and on the unusual topic of how to sacrifice efficiency for correctness and brevity during implementation.

Because of the poor computing infrastructure and limited control of programming language and environment available under contest conditions, unit and integration testing were deemed to be of limited value. Students were encouraged to unit test particularly problematic procedures or functions, but this encouragement was supplemented by little guidance or assistance.

4 Experience With The Process Model

In the 1999 PNW Regional ICPC, the two PSU teams each solved 3 of 8 problems, placing in the top half of the 61 schools participating. Students present on both 1998 and 1999 teams reported subjectively that the process employed was crucial to their improved performance. In particular, a marked improvement in estimation and time management was felt to greatly improve teamwork, leading to serious synergy in solution.

Post-mortem analysis suggested some minor improvements to the process, notably better team partitioning techniques and a slightly different allocation of team members to roles. Somewhat surprisingly, the emphasis on up-front activities in the model was endorsed by the students: they felt that accurate problem selection was key to their increased success.

Based on this feedback, the model was revised to include a limited amount of overlapping work on problems. In essence, while two team members (the implementor and the tester) finished up the implementation and submission, the other team member would become the principal implementor on the next problem. This was felt to be a bit more efficient. In addition, more emphasis was paid to careful testing and debugging discipline, which is difficult to enforce under the pressure of an actual contest.

Preparation for the 2000 contest began somewhat earlier in the academic year. One result of this earlier start and of the 1999 success was that the student pool was larger, and not all interested students were able to participate at the regional level. The final team composition was initially set by "coach's *fiat*", but was then altered by negotiations between team members. The resulting teams seemed to contain an effective combination of skills and experience.

The 2000 PNW Regional ICPC problems were substantially harder than those of 1999. For example, in 1999 the winning teams solved 8 of 8 problems, while in 2000 the top team solved 5 of 8. Similarly, in 1999, only 4 of the 61 teams participating solved no problems, while in 2000 10 of the 57 teams failed to achieve a single solution. The experience of the PSU 2000 teams was similar: each team solved 2 of 8 problems. This once again put them in roughly the middle of the overall ranking.

Several things were notable about the 1999 and 2000 PSU teams experiences. First, they solved their first problem substantially later than other teams with comparable records. Indeed, in 2000 only one problem was solved by either PSU team with only two hours remaining in the competition. This statistic seems to accurately reflect the extra up-front work put in, and this delay seems to have payed off in improved overall performance.

Second, the inter-team performance was remarkably consistent. This may be due to chance and the small sample size, but it seems intuitively to be due to the similar careful preparation and similar skill levels of all team members.

5 Application To Industrial Practice

Basic software process engineering has produced a software development process which seems to be reasonably effective in a highly unusual development environment. While the small sample size and unscientific nature of the experiment make it impossible to draw definitive statistical conclusions, a few modest observations seem appropriate.

The objective and subjective results seem to reflect truisms of the software process engineering community. First, software process was not a "silver bullet" [4]: the PSU teams did not become indomitable software power-houses. A fairly dramatic improvement was produced, but from the lower range of performance to the middle range. Second, while the improvements produced did not seem to be merely a function of greater resource availability or better-quality personnel, they were conditioned by more careful attention. Third, effective process engineering was not inordinately difficult: substantial gains were achieved by simple

and relatively common-sense methods. Finally, good software process was not necessarily unpalatable to those involved: in this case, student volunteers enthusiastically endorsed the increased focus and discipline associated with organized software development.

Much of the literature on software process and productivity is targeted at large organizations working on a series of large-scale and often safety-critical or mission critical software projects. Where attention has been paid to software process components and customization, it has usually been to techniques designed to scale well and to regularize the use of known best-practice techniques. There are several good reasons for this, notably the more extreme need for good process in development on this scale, and the fact that the whole notion of process-based development is borrowed from large-scale industrial engineering.

However, the gap between this sort of large-scale development process and the perhaps more common problem of small teams working on medium or short term projects, often under intense schedule pressure, has only recently begun to be addressed, notably by work on the Extreme Programming [3] model. In this setting, it is perhaps both easier and more necessary to adopt a wider variety of more "lightweight" approaches to software process.

Many of the process elements used in the ICPC process are applicable to this sort of development situation. But more importantly, the identification and selection of appropriate process elements, their combination into a sensible development process, and the constant re-evaluation of this process, are efforts well worth pursuing in small-scale projects.

Much of the process described in this document has been codified and published online (see **Availability** below). In the future, the process will be further tuned. In particular, a greater emphasis on earlier practice and training will be necessary. One interesting unexplored variable is the relationship between individual preparation and team preparation effort. The former has been largely neglected at PSU to date, and there is some evidence to indicate that the students are now largely skills-limited rather than process-limited. Given stronger attention to skills development, better preparation, and the vagaries of fortune, there is reason to hope that this model for ICPC programming will raise team performance even higher in the future.

In addition, as opportunities arise for exploring this sort of approach in industrial settings, they will be eagerly taken advantage of. A close personal friend, who has written hundreds of thousands of lines of industrial-quality code on a solo basis, was recently asked for an opinion on software engineering. The response was telling: "I wish there was some." As lightweight software process elements begin to percolate into this sort of development, perhaps this wish can begin to come true for a larger part of the development community.

Acknowledgements

The author wishes to thank the 1999 and 2000 PSU ACM ICPC teams for their support, encouragement, and tolerance during the development of the materials described here. They are all terrific people, and should be congratulated for their hard work, dedication, and talent. The support and encouragement of the faculty, students and staff of the PSU Computer Science Department and College of Engineering and Computer Science is gratefully acknowledged. In particular the encouragement and advice of Warren Harrison and Dick Hamlet in the preparation of this document have helped to make this work a reality.

Availability

The process model and training materials described are available to the Portland State University ACM Programming Team at `http://bart.cs.pdx.edu/acm-prog`. Those interested in sharing in the development and use of these materials are encouraged to visit the web site, and to contact the author at `bart@cs.pdx.edu` to obtain access to the protected portion of the content.

References

[1] The 2001 ACM International Collegiate Programming Contest. Web document `http://acm.baylor.edu/acmicpc/` accessed June 23, 2001 07:19 UTC.

[2] Donald Bagert and Barbara Boucher Owens. Organizing a team for the ACM programming contest. In *Proceedings of the 26th Technical Symposium on Computer Science Education*, volume 27 of *SIGCSE Bulletin*, page 402. ACM Press, March 1995.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[4] Frederick P. Brooks. No silver bullet. *IEEE Computer*, 20(4):10–19, April 1987. Reprinted in [5].

[5] F. P. Brooks, Jr. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, second edition, July 1995.

[6] Raymond Klefstad. The UCI Programming Contest Handbook. Web document `http://www.ics.uci.edu/~klefstad/contest/` accessed June 22, 2001 21:02 UTC.

[7] Glenford J. Myers. *The Art Of Software Testing*. John Wiley & Sons, 1979.

MANAGING THE PROPORTION OF TESTERS TO (OTHER)[1] DEVELOPERS

Cem Kaner, J.D., Ph.D.
Florida Institute of Technology

Elisabeth Hendrickson
Quality Tree Software, Inc.

Jennifer Smith-Brock
Ajilon Software Quality Partners

Pacific Northwest Software Quality Conference[2]

October 2001

ABSTRACT

One of the common test management questions is what is the right ratio of testers to other developers. Perhaps a credible benchmark number can provide convenience and bargaining power to the test manager working with an executive who has uninformed ideas about testing or whose objective is to spend the minimum necessary to conform to an industry standard.

We focused on staffing ratios and related issues for two days at the Fall 2000 meeting of the Software Test Managers Roundtable (STMR 3).[3] This paper is a report of our thinking. We assert the following:

---

[1] In most companies, testers work in the product development organization and they are part of the technological team that develops software products. Testers *are* developers. The ratios that we are interested are the ratio of testers to the *other* developers on the project.

[2] An earlier version of this paper appeared in the 2001 Proceedings of the International Software Quality Week.

[3] Software Test Managers Roundtable (STMR) meets twice yearly to discuss test management problems. A typical meeting has 15 experienced test managers, a facilitator and a recorder. There is no charge to attend the meetings, but attendance must be kept small to make the meetings manageable. If you are an experienced test manager and want to join in these discussions, please contact Cem Kaner, kaner@kaner.com. The meeting that is the basis for the present paper was STMR3, in San Jose, CA. Participants included Sue Bartlett, Laura Anneker, Fran McKain, Elisabeth Hendrickson, Bret Pettichord, Chris DeNardis, George Hamblen, Jim Williams, Brian Lawrence, Cem Kaner, Jennifer Smith-Brock, Kathy Iberle, Hung Quoc Nguyen, and Neal Reizer.

- One of the common answers is 1-to-1 (1 tester per programmer) or that 1-to-1 is the common ratio in leading edge companies[4] and is therefore desirable. Our experience has been that (to the extent that we can speak meaningfully about ratios at all) 1-to-1 has sometimes been a good ratio and sometimes a poor one.
- Ratios are calculated so differently from project to project that they probably incomparable.
- Project-specific factors will drive you toward different ratios, and toward different ratios at different times in the project. Such factors include (for example) the incoming reliability of the product, the extent to which the project involves new code that was written in-house, the extent to which the code was subjected to early analysis and review, the breadth of configurations that must be tested, the testability of the software, the availability of tools, the experience of the testers and other developers, corporate quality standards, and the allocation of work to testers and other developers.
- More is not necessarily better. A high ratio of testers to programmers may reflect a serious misallocation of resources and may do more harm than good.
- Across companies, testers do a wide variety of tasks. The more tasks that testers do, the more tester-time is needed to get the job done. We list and categorize many of the tasks that testers perform.
- The set of tasks undertaken by a test group should be determined by the group's mission. We examine a few different possible missions to illustrate this point.
- A ratio focuses executives on the wrong thing. The ratio is a comparison of body counts. For this many programmers you need that many testers. The ratio abstraction focuses attention away from the task list that drives up the staffing cost. It conveys nothing about what the testers will do. Instead it focuses attention onto a pair of numbers that have no direct link to anything but each other. At this level of abstraction, it even sounds meaningful to say, "Last time, your staffing ratio was at 1.2 to 1. You need to work smarter, and so I am setting you an objective of 10% greater efficiency. Go forth and become 1.08 to 1." The ratio is likely to distract attention from the important questions, such as: What will you *not do* in order to achieve that 10% reduction? And what task list is appropriate for you in the context of this project?

THESE RATIOS ARE INCOMPARABLE

What do we mean when we refer to a 1-to-1 ratio of testers to other developers? Across groups or even across projects by the same groups, these ratios can have wildly different meanings.

Consider the following stories:

---

[4] We thank Ross Collard (1999) for providing us with a summary of his interviews of senior testing staff at 18 companies that he classed as "leading-edge," such as BMC Software, Cisco, Global Village, Lucent, Microsoft. Six companies reported ratios of 1-to-1 or more, and the median ratio was 1-to-2.

Jane manages a project with the following personnel:

Staff	Counted as
4 programmers	programmers
1 development manager	programmer
1 test lead	Tester
1 black box tester	Tester
2 test automation engineers	Testers
1 buildmeister	Tester

According to the numbers, there's a 1-to-1 ratio between programmers and testers. However, when a new build comes into the test group, only one person is available to test it full time—the black box tester. Because of the apparent 1-to-1 ratio, management is puzzled by how long it takes the test group to do even simple tasks, like accept or reject a build. Jane is hard-pressed to explain the bottleneck to management—they keep coming back to the 1-to-1 ratio and insisting that means there are enough testers. The testers must be goofing off.

Now consider Carl's dilemma. His staff looks like this:

Staff	Counted as
1 programmer	programmer
1 toolsmith	programmer
1 buildmeister	programmer
1 development lead	programmer
1 development manager	programmer
1 test lead	Tester
4 black box testers	Testers

According to these numbers, there are 5 programmers and 5 testers, a comfortable 1-to-1 ratio. The testers report dozens of bugs per week. However, because they have no access to the source code (they test at the black box level), they cannot isolate the bugs they report. It takes the programmers significant time to understand and fix each reported issue. Carl is hard-pressed to explain why the testers can find bugs faster than his staff can fix them. Are his programmers lazy?

Sandy's department provides even more counting challenges:

Staff	Counted as
5 programmers	programmer
5 on-site consultants (doing programming)	???
1 project team (10 people of various specializations) who are under contract with Sandy's company to write and deliver a series of components to be used in Sandra's product.	???
1 full-time, on-staff test engineer	tester
3 technicians (they work for Sandry's company, are supervised by the engineer, but have limited discretion and experience)	???
3 temporary technicians (they work for a contracting agency, not Sandy's company, they report to the test engineer, but are not counted in the company's headcount)	???
3 testers who work offsite in an independent test lab	???

Should we count consultants as programmers? What about programmers who work for other companies and are simply selling code to Sandy's company? Should we count technicians as testers? What about technicians or other testers who work for other companies and provide testing services under contract? We don't know the "right" answer to these questions. We do know that different companies answer them differently and so they would calculate different ratios (ranging from 1-to-10 through 10-to-1) for the same situation.

Here are even more of the classification ambiguities in determining the ratio of testers to programmers:

- Are test managers testers? Are project managers developers? What about test leads and project leads? If a test lead sometimes runs test cases, should we count her as a tester for the hours that she is hunting for bugs? What about the hours she spends reviewing the test plans of the other testers?

- When programmers do code reviews, they find defects. Should we count them as testers? Imagine a six-month project that has one officially designated tester and ten officially designated programmers. In the first four months, the programmers spend 60% of their time critically analyzing requirements, specifications, and code, doing various types of walkthroughs and inspections. They find lots of problems. (In the other 40% of their time, they write code.) The tester also spends 60% of her time reading and participating in the meetings. Her other 40% is spent on the test plan. For these four months, should we count the ratio of testers to programmers as 1-to-10 or as 7-to-4? (After all, didn't the programmers spend 6 person-months doing bug hunting and only 4 person-months writing code?)
- If the testers write diagnostic code or tools that will make the programmers' lives easier as well as their own, are they working as testers or programmers?
- Imagine a six-month project that starts with four months of coding by ten programmers. During this part of the project, there are no testers. In the last two months, there are ten testers. Should we count this as 10-to-10 ratio or 60-to-20? (After all, there *were* 60 programmer-months on the project and only 20 tester-months.)
- Suppose that your company spends $1,000,000 licensing software components. These components required 36 programmer-months (and an unknown number of tester-months) to develop. Your company uses one programmer for 6 months to write an application that is primarily based on these components. It assigns one tester for 6 months. Is the ratio of testers to programmers 1-to-1 or 1-to-7 or something in between?
- How should we count technical writers, tech support staff, human factors analysts, systems analysts, system architects, executives, secretaries, testing interns, programming interns, marketeers, consultants to the programmers, consultants to the testers, and beta testers?
- If the programmers dump one of their incompetents into the testing group and one of the testers has to work half-time to babysit him, did the ratio of testers to programmers just go up or down? In general, if one group is consistently more (or less) productive than industry norm should we count them as if there were more (fewer) of them?

The answers to these questions might seem to be obvious to you, but whatever *your* answers are, someone respectable in a respectable company would answer them quite differently. At STMR 3, we marveled at the variety of ways that we counted tester-units for comparison with programmer-units. Because of the undefined counting rules, when two companies (or different groups in the same company) report their tester-to-programmer ratios, we can't tell from the ratios whether a reported ratio of 1 (tester) to 3 (programmers) involves more or less actual quality control than a ratio of 3 (testers) to 1 (programmer).

To put this more pointedly, when you hear someone claim in a conference talk that their ratio of testers to programmers is 1-to-1, you will probably have no idea what that means. Oh, you might have an idea, but it will be based on *your* assumptions and *not their*

situation. Whatever your impression of the staffing and work-sharing arrangements at that company is, it will probably be wrong.

ARE LARGER RATIOS BETTER OR WORSE?

Most of the participants at STMR 3 (including us) had worked on projects with high ratios of testers to programmers, as many as 5 testers per programmer. Most of us had also worked on projects involving very low ratios, as few as 0-to-7 and 1-to-8. Some of the projects with high ratios had been successful, some not. Some of the projects with low ratios had been successful, some not. This corresponds with what we've been told by other managers, outside of STMR.

Why are some projects successful with very few testers while others need so many more?

Low Ratios of Testers to Programmers

Most testers have seen or worked in a test group that was flooded with work and pushed up against tight deadlines. These groups typically staff projects with relatively few testers per programmer. The work is high stress and the overall product quality will probably be low.

However, some projects are correctly staffed with low ratios of testers to programmers. In our experience, these projects generally involved programmers (and managers) who had high quality standards and who didn't rely on the test group to get the product right.

Projects with low ratios of testers to programmers might occur:

- routinely, in a company with a healthy culture whose projects normally succeed
- routinely, under challenging circumstances
- on a project-by-project basis based on special circumstances of that project

Healthy Culture

Healthy cultures that have successful projects with relatively few testers often have characteristics like these:

- The test group has low noise-to-work ratios. "Noise" includes wasted time arising out of organizational chaos or an oppressive work environment.
- Staff turnover in the testing and programming groups is probably low. It takes time for testers to become efficient with a product--time, for example, to gain expertise and to build trust with the programmers.
- The company focuses on hiring skilled, experienced testers rather than "bodies."
- There is a shared agreement on the role of the test group, and little need for ongoing reevaluation or justification of the role.
- There is trust and respect between programmers and testers, and members of either group will help the other become more productive (for example, by helping them build tools).

- Quality is seen as everyone's business. The company emphasizes individual accountability. The person who makes a bug is expected to fix it and to learn something from the experience. There is a low churn rate for bugs--they don't ping-pong between programmers and testers ("Can't reproduce this bug", "I can", "It's not a bug anyway", "Marketing says it is", etc.)
- The code coming into testing is clean, designed for testability, and has good debug support.
- There may be an extensive unit test library that the programmers rerun whenever they update the product with their changes. The result is that the code they give to testers has fewer regression errors and needs less regression testing (Beck, 2000)
- In general, there is an emphasis on prevention of defects and/or on early discovery of them in technical reviews (such as inspections).
- In general, there is an emphasis on reuse of reusable test materials and on intelligent use of test tools
- The expectation is that reproducible coding errors will be fixed. Testers spend relatively little time justifying their test cases or doing extensive troubleshooting and market research just to convince the programmers that an error is worth fixing.
- The culture is more solution-oriented than blame-oriented.

Challenging Circumstances

Some companies need much more testing than they conduct, but they might not do it because:

- The product might be so complex that it is extremely expensive to train new testers. New testers won't understand how to test the product, and they'll waste too much programmer time on unimportant bugs and misunderstandings.
- They may have decided that time to market is more important than finding and fixing defects.
- They are in a dominant market position in their niche and their customers will pay extra for maintenance and support. There is thus (until competitors appear) relatively little incentive to the company to find and fix defects before release.
- They believe that it's right and natural for people in high tech to work 80+ hour weeks for 6+ months. In their view, adding staff will reduce the free overtime without increasing total productivity.
- The testing group may be perceived as not contributing because they aren't writing code.
- Other members of the project team might believe that testing is easy. ("What's so tough about testing? Just run the program! I can find bugs just by installing it! In fact, we should just bring in a bunch of Kelly temps to do this.")

- Testers might be perceived as too expensive.
- Testers might be perceived as incompetent, counterproductive twits.
- The test manager might be perceived as a whiner who should use his staff more effectively.
- The test group's work might be perceived as poor, with an overemphasis on unimportant issues ("corner cases") and the superficial aspects of the product.
- The testing group may have little credibility. They are seen as politically motivated and being preoccupied with irrelevant tests (e.g. some extreme corner-case tests). Therefore they are not sufficiently funded.
- The relation between testers and programmers may be toxic, resulting in excessive turnover in the testing group.

Some companies will never develop respect for their testing staff, and will never staff the test groups appropriately, no matter how good the testers or test managers. But in many other companies, testing groups build their own reputations over time. Some testing groups work too hard to increase their power and control in a company and not hard enough to improve their credibility and their technical contribution. Down that road, we think, tight staffing, high turnover, and layoffs are inevitable.

Project Factors

To some degree independently of the corporate culture, some ***projects*** are likely to succeed with few testers because of factors specific to those projects. For example:

- The product might involve low risk. No one expects it to work well and failures won't harm anyone.
- There might be little time-to-market pressure.
- The product might come to the test team with few defects (perhaps because this particular project team paid a lot of attention to the design, did paired programming or did a lot of inspections, etc.)
- The code might be particularly easy to test or relevant test tools that the testers are familiar with might be readily available.
- There might be no need to certify this product, no need for extensive documentation of the tests or (except for bug reports) the test results, and no requirement for detailed evaluations of the final quality of this product.
- The testers might simply not have much work to do on this project because it is easy, reliable, intuitive, testable, etc.

High Ratios of Testers to Programmers

We've met testers who respond enthusiastically when they hear of a group that has a very high ratio of testers to programmers. The impression that they have expressed to us is that

such a high ratio must indicate a corporate commitment to quality, and a healthier lifestyle (less stress, less grinding overtime) for the testers.

In many cases, though, a high number of testers results from (and contributes to) dysfunction in the product development effort.

One of us worked on a project that had roughly three times as many testers as programmers by the end of the project. The programmers were under intense time pressure—and they couldn't help but notice the large pool of people next door just waiting to catch their mistakes. The result? The bug introduction rate skyrocketed. One programmer commented about a particularly buggy area of the program under test, "Oh, yeah. I knew there would be bugs there—I just didn't have time to look for them myself."

Programmers find the vast majority of defects in their own code before they turn it over for testing. When a programmer finds a bug in her own code, she can usually isolate it quickly. She doesn't have to spend much time documenting the bug, replicating the bug, tracking it, or arguing that it should be fixed. When programmers skimp on testing, testers must spend much more time per bug to find, isolate, report, track, and advocate the fix. And then the programmer wastes time translating a black box test result back to code.

We suggest that there is a significant waste of project resources whenever an error is found by a black box tester that could have been easily found by the programmer using traditional glass box unit testing techniques. Some of these errors will inevitably creep through to testing, but we think staffing and lifecycle models that encourage over-reliance on black box testers are pathological.

Having an army of testers can encourage a spiraling drop in productivity and quality. (We talk more about this in Hendrickson, 2001, and Kaner, Falk & Nguyen, 1993, Chapter 15).

The best solution for severely buggy code is not to add testers. The best solution might be to freeze (or even reduce) the size of the testing group while adding programmers. The programmers should fix and test code, not add even more buggy features.

Healthy Cultures

Some companies need more testers because of the market they are in or the technology they use. The examples below might describe the culture of the company or the circumstances of a particular project. Examples:

- Much more formal planning, documentation, and archiving of all artifacts of the testing effort is needed when developing safety-critical software. Heavy documentation might be required for other software because of regulatory agency interest or high litigation risk.
- Extensive documentation may also be needed for software that will be sold in its entirety to a customer, with the expectation that the customer will assume responsibility for future maintenance, support and enhancement.

- Some markets are particularly picky about fit and finish errors or are more likely to expect / demand technical support for problems that customers in other markets might seem small or easy to solve. If you are selling into that market, you'll probably do much more user interface testing and much more scenario testing.
- Extensive configuration testing is needed for software that must work on many platforms or support many different technologies or types of software or peripherals.
- Load testing is needed for software that is subject to bursts of peak usage.
- Some companies hire domain experts into testing or train several testers into domain expertise. These testers become knowledgeable advocates for customer satisfaction improvements and are particularly important in projects whose designs emerge over time.
- Some testing groups have a broad charter. Along with testing, they provide several other development services such as debugging, specification writing, benchmarking competing products, participation in code reviews, and so on. The broader the charter, the more people are needed to do the work.
- If the company relies on outsourced testing (this is sometimes a requirement of the customer's), there is substantial communication cost. The external testers need time to understand the product, the market, and the risks. They also need significant support (people to answer questions and documentation) from in-house testing staff.
- Software that involves a large number of components can be very complex and requires more testing than a simpler architecture.
- The development project might involve relatively little fresh code, but a large end product. The product might be knitted together from many externally written components or it might be an upgrade of an existing product. The testers will still have to do system testing (the less you trust the external code or the modification process, the more testing is needed). When the external components come from many sources, the test group may have to research, design and execute many different usage scenario tests in order to see how well the components work together to meet actual customer needs.

Challenging Circumstances

Some companies or projects back themselves into excessive testing staff sizes. For example:

- Some test groups don't understand domain testing or combinatorial testing, so they try to test too many values of too many variables in too many combinations.
- Some test groups rely on large numbers of low-skill testers. Manual execution of large sets of fully scripted test cases can be extremely labor-intensive, mind-numbing for testers and test case maintainers, and not very effective as a method of finding defects.

- Test groups that suffer high turnover are constantly in training mode. The staff may never get fully proficient with base technologies, available tools, or the software under test. Tasks that would be easy for a locally experienced tester might take a newcomer tremendously longer to understand and do.
- Testers may be given inadequate tools. Most testers need at least two computers, access to a configuration or replication lab, a decent bug tracking system, and various test automation tools. To the extent that the software under test runs on platforms for which there are few test tools, the testers have less opportunity to become efficient.
- Testing can be inefficient because the team doesn't use basic control procedures such as smoke tests and configuration management software.
- Software that was not designed for testability will be more difficult and thus more time consuming to test.
- Some corporate metrics projects waste time on the data collection, the data fudging (see Kaner, 2001; Hoffman, 2000), and the gossiping about the dummies in head office who rely on these stupid metrics. We are not suggesting that metrics efforts are necessarily worthless. We are saying that we have seen several such worthless efforts, and they create a lot of distraction.
- Programmers might focus entirely on implementing features. In some companies, testers write installers, do builds, write all the documentation, etc. This is not necessarily a bad thing. Instead it reflects a division of labor that might be wise under the circumstances but that must be factored into the budgets and staffing of both groups.
- Programming teams might send excessively buggy code into testing, perhaps because they are untrained in base technologies, or new to the project, or managed to implement features as quickly as possible, leaving the testing to testers. The worst case of this reflects a conscious decision that they don't have to test the code because they can count on the testers to find everything. Add more testers and the programmers do even less checking of their work. This can become a vicious spiral of increasing testing costs paired with declining quality (Hendrickson, 2001).

One-to-One Ratios of Testers to Programmers

Sometimes groups that describe their work in terms of one-to-one ratios really mean that they use paired teams of programmers and testers. For a given type of feature, a specific tester and a specific programmer work together, perhaps for several years.

There are many advantages to this approach. In particular, the tester is in a position to become very knowledgeable about the types of features this programmer works on and the types of errors this programmer makes. If the programmer and tester get along well, their communication about product risks and bugs will probably get very efficient.

On the other hand, a programmer-tester pair sometimes develops an idiosyncratic model of what things are acceptable to customers or reasonable to report and fix.

FACTORS THAT INFLUENCE STAFFING RATIOS AND LEVELS

Suppose that your boss calls you into a meeting and says, "We have just been assigned Project Whizbang. It will take 20 programmer-years. They started last week and will be done in 6 months. How many testers do you need?"

How do you come up with an answer?

If you use the ratio approach, you might say something like, "In the last 10 projects, we averaged 1.2 testers for every programmer. We were pretty understaffed, though. We had to work lots of overtime, and the product still went out with bugs. So I think we need a ratio of 1.5 testers per programmer, or 30 tester-years. How soon can we start?

A few colleagues of ours, who have significant and successful management experience, have had good experiences with this approach. They don't expect to stop at 30-tester years. They use this number to open the negotiations, to give them a reasonable place to start from.

We agree, *we strongly agree,* that historical data is useful. But we are concerned that historical data, *summarized this way*, can be counterproductive.

A ratio focuses on a *relationship* between two numbers.

These numbers have no direct link to anything but each other. The relationship conveys nothing about the task list, about what the testers will do or what the programmers will do.

At this level of abstraction, it might sound reasonable to say something like, "Last time, your staffing ratio was at 1.2 to 1. I am setting you an objective of 10% greater efficiency. Go forth and become 1.08 to 1."

So, how can we use historical data but still steer the conversation to our needs and responsibilities?

We think we can use the same information that we'd use to justify a ratio, to justify a predicted staffing level. The difference is that when someone asks about the staffing level, we'd talk about the tasks that the testers do (and the ones they won't do if they run out of time) rather than the proportion of testers to programmers.

A Two-Factor Model for Predicting Staffing Requirements

Here's a simple approach for estimating the testing needs for a project. Create a table that shows project size and risk:

		Project Size				
		Very Small	Small	Medium	Large	Very Large
Project Risk	Very low					
	Low					
	Medium					
	High					
	Very high					

As you complete projects, fill in the staff size you had and the staff size that (at the end of the project, with the benefit of hindsight) you think you needed.

Here's an example. The project is maintenance of a product that has been reasonably stable in the field. There are several bug fixes, totaling about 1.5 programmer-months of work. The programmers involved are familiar with the program and have a good track record. However, some of the bug fixes involve device-handling, so extra configuration testing is needed. Therefore, you spend 3 tester-months on the project and at the end of the project, you feel that this was about the right number. You might class this as a small project with low risk—enter 3 months into the Small/Low cell.

Project size is partially determined by the number of programmer-hours, but there are many other factors. For example, adding a hundred new components makes the project large (from the viewpoint of what will have to be tested) even if the programmers took them from a library and spent almost no time on them. Similarly, the project size is increased by an extensive documentation requirement.

The level of risk is affected by such factors as the technical difficulty of the programming task, the skills of the programmers, the expectations of the customers, and the types of harm that errors might cause. The more risk, the more thoroughly you'll have to test, and the more times you'll probably have to retest.

As you gain experience (yours or colleagues'), you'll fill the table with values.

When someone asks you to estimate a new project, use the table. Ask questions to get a sense of the size and level of risk, and then you can say, "This is like these projects, and they took so much staff."

This two-factor model is quite useful.

Rothman (2000) proposed a useful three-factor model, that considered the product (some are harder to test than others), the project and its process (some projects employ better processes than others), and the people and their skills. As with the two-factor model, this folds quite a few considerations into a few variables.

We think it is useful to think explicitly in terms of a longer list of factors, such as the following ten:

- Mission of the testing group.
- Allocation of labor (responsibility for different tasks) between testers and programmers.
- People and their skills.
- Partnerships between testers and other stakeholders.
- Product under test.
- Market expectations
- Project details (e.g. what resources are available when.
- Process (principles and procedures intended to govern the running of the project)
- Methodology (principles and procedures intended to govern the detailed implementation of the product or the development of product artifacts)
- Test infrastructure

We don't think this is the ultimate list. You might do well to generate your own. Our point is that if you are trying to understand your staffing situation, it can help to start by listing several different dimensions to consider. Considering them each in turn, alone or preferably in a brainstorming session with a small group, can lead to a broad and useful set of issues to consider.

Here are additional thoughts about two of these factors, the group mission and the allocation of labor.

Mission of the Testing Group

Different testing groups, even within the same company, have different missions. For example, these are all common missions (although some of them might not be possible):

- Find defects.
- Maximize the number of bugs found.
- Block premature product releases.
- Help managers make ship / no-ship decisions.
- Assess quality.
- Minimize technical support costs.
- Conform to regulations.

- Minimize safety-related lawsuit risk.
- Assess conformance to specification.
- Find safe scenarios for use of the product (find ways to get it to work, in spite of the bugs).
- Verify correctness of the product.
- Assure quality.

A group focused on regulation will spend far more time pre-planning and documenting its work than a group focused on finding the largest number of bugs in the time available. The staffing requirements of the two testing groups will also be quite different.

Allocation of Labour

The most important driver of the ratio of testers to programmers should be the allocation of labor between the groups. If testers take on tasks that go beyond the minimum essentials of black box testing, it will take more time or more testers to finish testing the software.

To estimate how many testers you need to perform the job, you need a clear idea of what those testers are going to do. At a bare minimum, the testers will probably:

- Design tests
- Execute tests
- Report bugs

They will probably also spend time interpreting results, isolating bugs, regressing fixes, and performing other similar tasks.

In some organizations, the testers have a much broader range of responsibilities. For example, testers may also:

- Write requirements
- Participate in inspections and walkthroughs
- Compile the software
- Write installers
- Investigate bugs, analyzing the source code to discover the underlying errors
- Conduct unit tests and other glass box tests
- Configure and maintain programming-related tools, such as the source control system
- Archive the software
- Evaluate the reliability of components that the company is thinking of using in its software
- Provide technical support

- Demonstrate the product at trade shows or internal company meetings
- Train new users (or tech support or training staff) in the use of the product
- Provide risk assessments
- Collect and report statistical data (software metrics) about the project
- Build and maintain internal test-related tools such as the bug tracking system
- Benchmark competing products
- Evaluate the significance of various hardware/software configurations in the marketplace (to inform their choices of configuration tests)
- Conduct usability tests
- Lead or audit efforts to comply with regulatory or industry standards (such as those published by SEI, ISO, IEEE, FDA, etc.)
- Provide a wide range of project management services.

We do not espouse a preferred division of labor in this paper. Any of the tasks above might be appropriately assigned to a test group, depending on its charter. There is nothing wrong with that, as long as the group is appropriately staffed for its tasks.

We think that the best way to estimate your staffing level for a project is task-based. Start by listing the tasks that your staff will do and estimate, task by task, how much work is involved. (If you're not sure how to do this, Kaner, 1996, describes a task-by-task estimation approach.) The total number of staffed tester-hours should be based on this estimate. The ratio of this staff to the programming staff size will emerge as a result, not as a driver of proper staffing.

CLOSING COMMENTS

Ratios out of context are meaningless. Attempting to use industry figures for ratios is at best meaningless and more likely dangerous.

Testers often ask about industry standard ratios in order to use these numbers to justify a staff increase. To justify an increase in staff, we suggest that you argue from your tasks and your backlog of work, not for a given ratio.

Even if you have a backlog, adding testers won't necessarily help clear it (Hendrickson, 2001). Many problems that drive down a test group's productivity cannot be solved by adding testers. For example, poor source control, blocking bugs, missing features, and designs that are inconsistent and undocumented are not going to be solved by doing more testing.

References

Beck, Kent (2000), *Extreme Programming*, Addison Wesley.

Collard, Ross (1999), "Testing & QA Staffing Levels: Internal IS Organizations", Collard & Company.

Hendrickson, Elisabeth (2001), "Better Testing--Worse Quality?", Proceedings of the International Conference on Software Management, San Diego, CA. <http://www.qualitytree.com/feature/btwq.pdf>

Hoffman, Doug (2000) "The Darker Side of Metrics", Proceedings of the 18th Pacific Northwest Software Quality Conference, Portland, OR.

Kaner, Cem (1996) "Negotiating Testing Resources: A Collaborative Approach", Proceedings of the Software Quality Week conference, San Francisco, CA.

Kaner, Cem (2001) "Measurement Issues & Software Testing", QUEST Conference Proceedings, Orlando, FL.

Kaner, Cem, Jack Falk, & Hung Quoc Nguyen (1993; republished 1999) *Testing Computer Software*, John Wiley & Sons.

Rothman, Johanna (2000), "It Depends: Deciding on the Correct Ratio of Developers to Testers," <http://www.testing.com/test-patterns/index.html>.

Estimating Tester to Developer Ratios (or Not)

Kathy Iberle
Hewlett-Packard
18110 SE 34th Street
Vancouver, WA 98683
Kathy_Iberle@hp.com

Sue Bartlett
STEP Technology
6915 SW Macadam Blvd., Suite 200
Portland, OR 97219
sbartlett1@qwest.net

Abstract

Test managers often need to make an initial estimate of the number of people that will be required to test a particular product, before the information or the time to do a detailed task breakdown is available. One piece of data that is almost always available is the number of developers that are or will be working on the project in question. Common sense suggests that there is a relationship between the number of testers and the number of developers.

This article presents a model that can be used in describing that relationship. It is a heuristic method for predicting a ratio of testers to developers on future projects. The method uses the model to predict differences from a baseline project. A reader with some projects behind her will be able to come up with a rule-of-thumb model to suit her most common situations, to understand when the model might not give accurate answers and what additional factors might need to be taken into consideration.

Biographies

Kathy Iberle is a senior software process engineer at Hewlett-Packard. Over the past 16 years, she has been involved in software development and testing for products ranging from data management systems for medical test results to inkjet printer drivers and, most recently, e-services. Kathy has worked extensively on researching appropriate development and test methodologies for different situations, training test staff, and developing processes and templates for effective and efficient software testing.

Sue Bartlett is the QA Manager at STEP Technology, a software company that builds technically advanced, high performance, scalable, eBusiness and Enterprise applications. She has over 18 years of experience in software quality assurance and testing, software process improvement and software development. Her domain experience ranges from Computer Aided Design (CAD), color printers, and most recently, web and other business applications. Over half of her career has been at a technical lead or managerial level. Sue has been actively involved in testing education and process.

Acknowledgement

Some of the material in this paper was developed by the participants of the Software Test Managers Roundtable (STMR). The Software Test Managers Roundtable meets twice yearly to discuss test management problems. The meeting that is the basis for the present paper was STMR3, in San Jose, CA. Participants included Sue Bartlett, Laura Anneker, Fran McKain, Elisabeth Hendrickson, Bret Pettichord, Chris DeNardis, George Hamblen, Jim Williams, Brian Lawrence, Cem Kaner, Jennifer Smith-Brock, Kathy Iberle, Hung Quoc Nguyen, and Neal Reizer. Facilities and other support for STMR were provided by Software Quality Engineering, which hosts these meetings in conjunction with the STAR conferences. We also thank the members of the email discussion group for their comments in reviewing our paper.

The discussion at STMR3 started with the paper "It Depends: Deciding on the Correct Ratio of Developers to Testers" by Johanna Rothman, http://www.jrothman.com/Papers/ItDepends.html.

Introduction

Past tester to developer ratios are often useful for making rough estimates of required test staff based on the number of the developers that will be working on the project in question. Common sense tells us that there must be *some* relationship between the amount of code and the amount of testing needed – and some relationship between the amount of code and the number of developers. This suggests that there is a relationship between the number of developers and the amount of testing needed.

The oft-quoted search for the "right" ratio of testers to developers assumes that these relationships are always linear, although there is no proof of this and considerable evidence to the contrary. We take a different approach, building a systems model that does not assume a linear ratio but instead can be used to predict the direction (if not the magnitude) of different factors on the relationship between the number of testers and the number of developers. This allows the systematic examination of the effect of a number of different factors at the same time.

This heuristic method does require at least one baseline project in a similar domain – a project where the ratio of testers to developers was known, and the relative values of the influences as compared between the baseline project and the new project is known. It isn't necessary to know absolute values if you can say that something is "oh, maybe twice as big" or "about the same".

The Model

The effects that drive the number of testers needed on any particular project are quite complex. The diagram below shows the major effects that contribute to the need for the number of testers.

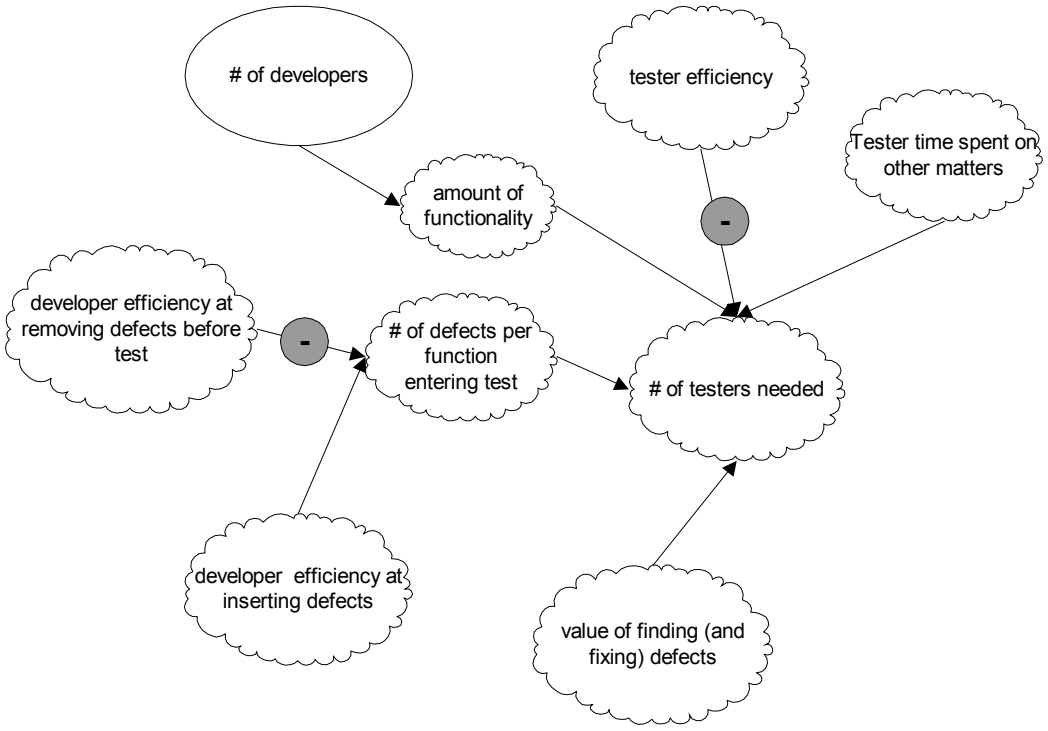


Fig. 1: Diagram of Effects on the Number of Testers Needed

This is a diagram of effects[1], in which each node represents a potentially measurable quantity and the arrows represent effects. The clouds are quantities that could be measured, at least approximately, but have not been. An ellipse represents a quantity that has been measured. The arrows are labeled to indicate whether one node affects another positively or negatively. For example, if tester efficiency is better (higher), then the number of testers needed is lower. If the time testers spend on other matters is increased, then the number of testers needed is higher. If the developers are more efficient at inserting defects, then more testers are needed to find those defects. ("Efficiency at inserting defects" sounds bad, but it is sometimes a natural outcome of overall productivity improvements in development, as we shall see.)

But You Promised Me a Ratio…

The "Ideal" Tester-Developer Ratio

The concept of an ideal, unvarying tester-developer ratio is the same as believing that all the effects shown above are either

- Constant and never-changing for all projects in all situations
  or
- Magically cancelling each other out exactly

which would reduce the complexity of the previous effects diagram to Fig. 2. below. This should strain the credulity of any thinking person, including high-level managers.

[1] For a detailed description of effects diagrams, see G.M. Weinberg, *Quality Software Management, Vol. 1: Systems Thinking*. (New York: Dorset House Publishing, 1992).

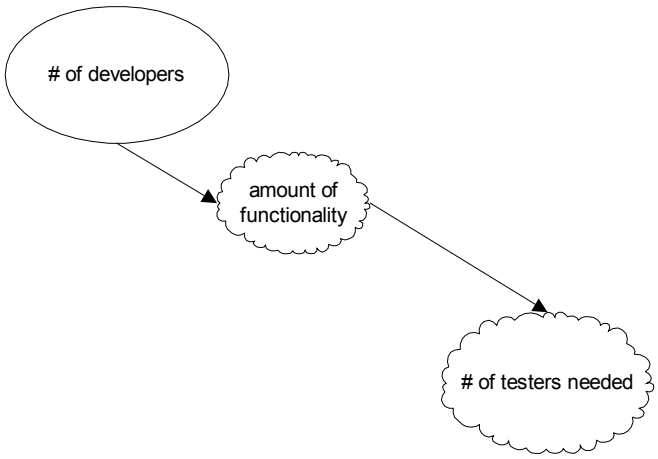


Fig. 2: The ideal tester-developer ratio ignores most effects

Furthermore, a ratio means a linear relationship, so the concept of an ideal ratio implies that the relationship between number of developers and amount of functionality produced is also linear – i.e. that Brooks' "mythical man-month"[2] is not mythical at all.

Comparing Ratios Between Businesses or Companies

It may occur to you to wonder whether the effects in the diagram can be accounted for by having several different ratios, to represent different sets of values of the various effects.

Unfortunately, it is not possible to know whether there is an ideal ratio for any particular business situation. The reported ratios have been measured in a huge variety of ways. The ratio varies dramatically depending on the definitions of "tester", "developer", "headcount", and so forth.[3] Unless you can find out in considerable detail how the other people's ratios were measured, the accuracy of any extrapolation will be seriously questionable.

However, the measurement systems don't appear to vary infinitely. For instance, no one reports ratios of 1 tester to 1000 developers, or 1000 testers to one developer. The combined margin of error in the measurement systems and the actual difference between various projects and businesses doesn't appear to exceed an order of magnitude. Therefore, published ratios can sometimes be used to defend oneself from utterly ridiculous proposals, such as a ratio of 1:30.

So is There Any Good Use for Ratios At All?

The authors believe that there is.

Ratios are useful to estimate the needed number of testers when all of these are true:

- the amount of planned functionality is **not** known or very vague
- the number of developers proposed is considerably more reliable than the functionality description
- data from past projects on number of developers and number of testers is available
- the staffing data is using known units (this almost always means that it is your own data from your own past experience)
- some relative information about the magnitude of the various effects is available.

---

[2] *The Mythical Man-Month, 1st edition;* Frederick P. Brooks; Addison-Wesley; 1975

[3] see "Managing the Proportion of Testers to (Other) Developers" by Kaner, Hendrickson, and Brock, Quality Week 2001 Proceedings for an in-depth discussion.

If the amount of planned functionality is known in detail, it is more accurate to estimate the number of testers by doing a work breakdown of the proposed testing tasks, and then estimating the number of testers needed to complete those tasks. (Note that both methods involve using your own estimate of tester efficiency in your organization).

Estimation in the Absence of Data

The authors have observed a variety of cases in which the number of developers is more reliably known than the functionality and other information on which a work breakdown would be based.

- **Long-term planning in a company that can project its headcount**. Future projects whose start is a year or more away are often represented simply by the number of developers and a project title, such as "Driver for the next cool printer".
- **An early estimate for Return-On-Investment calculation while a project is still in the idea stage**. The available data is a few paragraphs of project description and a number of developers, often stated as a range. In this case, reporting the estimate as a ratio is often the most useful prediction for the company – because they are trying to get a quick handle on whether this project would be worth doing at all, and your test cost definitely figures in.
- **Initial bid submissions**. The initial proposal stage for contract software in a fast turnaround business can be as short as a two-page estimate or description document. Due to market conditions, turnaround is critical. There are many more proposals being bid on than actually landed. In order to save time, a "solutions architect" often does the initial analysis and proposal without involvement from QA or engineering. The QA staff provides a "rule of thumb" for estimating test time by giving the solutions architect a ratio, plus a couple of items which are added to the estimated test time if needed, such as performance testing for high reliability situations or formal usability testing.

How To Estimate Using Ratios

Given that you have an estimation job that fits the above criteria….

1. Choose a baseline project or projects.
2. Collect data on its ratio of testers to developers.
3. Collect data on the various effects shown in Figure 1.
4. Make an initial estimate of the number of testers, based on the ratio in step #2.
5. For each effect bubble, compare the factors that are substantially different between the upcoming project and the baseline project. (An extensive list of factors is provided in the appendix to this paper.)
6. Combine the results of the various differences.
7. Adjust the number of testers accordingly, using your professional judgment.

If you have more baseline projects, you may wish to repeat the exercise and see if the second estimate is close to the first.

Your ability to estimate accurately will increase over time if you collect enough data to have several interesting points of comparison.

The Various Effects

The STMR group came up with a long list of factors that had been observed to affect the ratio of testers to developers on a particular project. The authors grouped the factors into the effect nodes shown in Figure 1.

Developer Efficiency at Inserting Defects

The more defects that are inserted, the longer it will take to find them all, write reports on them, talk to the developers about the details, and re-run the original tests when the fixes arrive in testing.

While it is tempting to say that defects are entirely due to sloppy development work, in truth there are quite a few factors that are completely outside of the developer's control. Some of these factors are multiplicative. For

instance, products that are meant to run on a variety of operating systems or browsers are particularly vulnerable to this multiplying effect. One line of code can spawn dozens of different failures.

Other factors are actually good for the project overall – for instance, code reuse. Incorporating large amounts of reused software is a very easy way to increase the number of defects inserted per developer, because the developer now has many invisible and uncountable developers helping to insert defects, as well as additional functionality. It would be more accurate to say that the number of developers has increased, and thus the amount of functionality has increased, but we don't know any ways to count the number of developers for off-the-shelf software.

- **People** – Inexperienced people make more mistakes. Understaffing or rushing the developers will also cause errors. Some programmers have habits that reduce errors, such as routinely initializing all variables, and others have habits that tend to introduce errors, such as leaving error-handling code until the end.
- **Organization** – Anything that hinders communication among the developers or from sponsors will increase the number of defects inserted.
- **Product** - It's easier to make mistakes on some products – poorly documented legacy code, for example. Some products use a line of code in only one environment, whereas others run a line in dozens of environments, thus increasing the developer's efficiency at inserting defects through unintended consequences in one or more of the environments. On the other hand, it's difficult to make interesting mistakes on websites that are essentially online brochures. Incorporation of others' code, including off-the-shelf software, increases the efficiency of the visible developers at inserting defects (which they themselves did not create).
- **Process** – Use of good software engineering practices such as determining user requirements and modular design will result in fewer defects inserted.

Developer Efficiency at Finding Defects before Entering Test

Developers find almost all of their own errors before the code ever reaches the test group. Even small increases in efficiency at finding defects will reduce the number entering test substantially.

- **People** – As before, experience, talent, education in relevant techniques, and motivation all help.
- **Organization** – The STMR group reported many cases in which the organization actually discouraged developers from finding their own errors by dictating that they must not spend time looking for them, or refusing to provide the appropriate equipment. For instance, if the product will run on multiple operating systems, then multiple computers are usually needed.
- **Product** – I can't think of any product factors that would make it easier to find defects.
- **Process** – Code inspections and unit testing increase the number of defects found by the developers. Frequent milestones also tend to encourage finding defects by encouraging focus on the current work, with more confidence that all the work will get done.

Tester Efficiency at Finding Defects

This factor inversely affects the number of testers – the more efficient the testers, the fewer of them are needed. This is primarily affected by the skill and experience of the testers, and by the tools available to them. Automation can increase tester efficiency, but it can also decrease efficiency if the testers end up enslaved to maintenance of marginally effective automation.

- **People** – Inexperienced testers take longer to set up for the tests, to find the defects, to isolate and characterize the defects, and to write the defect reports. Inexperienced testers often use an inefficient test strategy. They spend more time testing in places where the defects are not lurking, or test the same code multiple times, or order tests in such a way that the setup or execution is less efficient than it could be.
- **Organization** – Anything that hinders communication between the testers themselves or between developers and testers will slow down the finding of defects and communicating them to the developers.
- **Product** – Some types of products take longer to test – those with no user interface where a test harness must be written, for instance, or where a great deal of setup time is required.
- **Process** – Failure to communicate the correct behavior of the product to the testers will really slow down the defect finding (for instance, no specifications at all). Not using appropriate tools for the situation will

slow things down. Insistence on extraneous process (ill-conceived metrics programs, for instance) can also slow things down.

Value of Defects Found – A Look At Risk

The cost of unfound defects varies tremendously across different businesses and types of products. It's reasonable to spend a lot more time and money to find a defect in a missile targeting system than one in the latest experimental Internet game. When the defects are not particularly valuable, it doesn't make much sense to do a lot of testing to find them.

The magnitude of this effect is directly related to the potential impact of errors. If a defect that slips through can cause serious damage or injury, or fixing the defects after release would be prohibitively expensive (e.g. firmware), then the testing effort must be greater and therefore the number of testers goes up. The increased testing effort consists both of looking more carefully and thoroughly for defects on the first time through any area, and increased regression testing.

The value of defects found considers only the impact of defects, whereas the probability of the presence of defects is represented as part of the "developer efficiency at inserting defects". Both are part of risk.

- **Product** – The end use of the product, as one would imagine, has a huge effect on the value of defects found. Medical products, missile control systems, air-traffic control – all require placing a high value on defects found.
- **Customer Expectations** – If the customers are more tolerant of defects in exchange for a lower price or an earlier delivery, it may be acceptable to have less testing and therefore fewer testers. This changes over the course of the adoption of a product, so it's not the same as the end use of the product. Currently, the users of many Internet products will accept lower quality because it's a hot new product, and some businesses take advantage of that by skipping testing[4].

Time spent on other matters

The STMR group reported a wide variety of jobs being performed by testers that were not strictly testing. This ranged from documenting product requirements, to writing installer code, to staffing the technical support help desk. This isn't necessarily a bad arrangement for a business, but it does affect the number of people labeled as "Tester" who will be needed to get a particular project done.

How to Use the Model and Ratios to Estimate Needed Testers

In this section, we will walk through some examples of using the model, using case studies derived from some actual projects. Note that if we had had data about the amount and type of functionality in the proposed project, we would have used a more traditional work-breakdown method. The names and some details have been changed to preserve anonymity.

Collecting Data for a Baseline

The first step involves collecting some data on past projects. The data will involve both hard data and soft data. Hard data includes the number of testers and the number of developers in whatever format you count – hours billed, Full-Time-Equivalents, engineer-months. Soft data can include a wide variety of information that is implicated in the effects discussed above. You may have hard data for some of these, or they may all be subjective.

We have two potential baseline projects: MergoApp and DinkyApp.

Baseline 1 – MergoApp

MergoApp was a fairly straightforward e-commerce website tying into a generic back end SQL database, with third party credit card tie-ins and a custom administration tool.

---

[4] You can learn more about "early adopters" in Geoffrey Moore's *Crossing the Chasm*, Harper Business, 1999.

Developer efficiency at inserting defects:

- **People:** The technical lead, project manager and developers were all quite experienced and quality conscious.
- **Organization:** The project manager was excellent at enabling good communication between team members. There was good communication between client and project manager.
- **Product:** The product was reasonably well documented and there were no legacy issues.
- **Process:** Good development processes were followed: with reasonable specifications and tools used when needed.

Developer efficiency at removing defects before test:

- **People:** The technical lead and developer had the necessary skills to remove most defects.
- **Organization:** The developer was expected to unit test.
- **Product:** No barriers to defect removal.
- **Process:** Supportive of early defect removal.

Tester efficiency at finding defects:

- **People:** The QA person was an expert tester with extensive programming and testing experience.
- **Organization:** The environment was conducive to good communication, and team members respected each other.
- **Product:** It was reasonably well documented and the user interface was conducive to testing.
- **Process:** A good interactive process was followed between test and development.

Value of defects found:

- **Product:** There were no injury issues to worry about. There was an average level of concern for accuracy, security, and privacy, and medium desire for reliability.
- **Business goals:** There was a low push for time-to-market and little value was placed on low rated bugs.

Time spent on other matters:

- Minimal.

The tester – developer ratio for this project was 1:4. Remember that this ratio makes sense only when compared to other ratios measured using the exact same measures. Since we aren't going to explain how we measured this ratio, it isn't a relevant baseline for your own projects.

Baseline 2 - DinkyApp

Now, we have DinkyApp. DinkyApp is a fairly simple and small e-commerce site. However, an explosion of factors caused quite a variation in the tester-developer ratio.

Developer efficiency at inserting defects:

- **People:** The technical lead was the only programmer on the project. His experience was relatively low compared to the programmers on MergoApp.
- **Organization:** Little attention was given to this tiny project from the project manager.
- **Product**: The product interfaced with unfamiliar third party software and hardware.
- **Process:** There was only a two-page product proposal, no specifications, and little or no up-front design.

Developer efficiency at removing defects before test:

- **People:** Compared to the programmers on MergoApp, the programmer had lower expectations of his work.
- **Organization:** Unit testing and reviews were not encouraged.
- **Product**: It was difficult to do good unit testing due to unfamiliarity with the third party software.
- **Process:** No real feedback on defect release into test.

Tester efficiency at finding defects:

- **People:** The QA person was less experienced, with only a couple of years in testing.
- **Organization:** Neither the developer nor the project manager communicated effectively with the tester. The tester had to pull every bit of information, including when code was ready for testing.
- **Product:** DinkyApp was simpler than MergoApp and thus was easier to test.
- **Process:** There was no specification, written or verbal.

Value of defects found:

- **Product**: There was a medium desire for reliability. Little value was placed on low rated bugs.
- **Business:** nothing relevant.

Time spent on other matters:

- The tester took up the slack in documenting much of what the product does in the test plan and deployment documents.

The tester-developer ratio for this project was 1:1.

Estimating the Number of Testers

Case Study

Now we are asked to make a quick-and-dirty estimate of the number of testers required for DataApp. DataApp is a database implementation to replace an existing spreadsheet solution. We know the estimated number of developers.

1. *Choose a baseline project or projects*.
   Our two previous projects had dramatically different ratios, so we need to choose one to be the baseline in this case. We decide to choose MergoApp – so all subsequent comparisons will be made with MergoApp.
2. *Collect data on the baseline's ratio of testers to developers*.
   That was 1:4.
3. *Collect data on the various effects shown in Figure 1*.
   We can use the information given in the section labeled "Baseline 1" above.
4. *Make an initial estimate of the number of testers, based on the ratio in step #2*.
   The number of developers on this project is 8, so the initial number of testers is 2. This is the preliminary value for the "# of testers" bubble on the effects diagram.
5. *For each effect bubble, compare the factors that are substantially different between the upcoming project and the baseline project.*
   This is where it gets interesting. For each effect cloud, compare the current project with the baseline project. If the comparison suggests that the effect will be more powerful, put a "+" in the bubble. If it suggests that the effect will be less powerful, put a "-". You will have to use your own judgment as to whether factors should be counted equally or not.

   Developer efficiency at inserting defects:
   - **People:** The developer is quality conscious with considerable experience. SAME as MergoApp.
   - **Organization:** Adequate coverage of project management duties and reasonable communication between client and team is expected, based on past performance of the assigned staff. SAME as MergoApp.
   - **Product**: The staff acts like they understand the project. SAME as MergoApp.
   - **Process**: There are reasonable specifications and good process will be followed. SAME as MergoApp.

   It appears that the number of defects inserted per developer will probably be pretty similar for MergoApp and DataApp.

   Developer efficiency at removing defects before test:
   - **People:** The developer is skilled at removing defects. SAME as MergoApp.
   - **Organization:** Some level of unit testing is expected. SAME as MergoApp.
   - **Product**: No barriers to defect removal. SAME as MergoApp.
   - **Process**: Again, process expectations appear much the SAME as MergoApp.

Tester efficiency at finding defects:

- **People:** The QA person is less experienced than MergoApp's tester, with only a few years in testing. (Same tester as in DinkyApp.) LOWER than MergoApp – put a "-" in the "Defects found per tester" cloud.
- **Organization:** We anticipate good communication and respect between team members. SAME as MergoApp.
- **Product:** A test harness and some SQL programming will be required in order to test. Also a "-" – this is going to take extra time.
- **Process:** The test process is quite similar to that of MergoApp. SAME.

The "defects found per tester" cloud gets two "-", suggesting that we will need extra testers.

Value of defects found:

- **Product**: There is a high desire for reliability, some value on low rated bugs. A little HIGHER than MergoApp.
- **Business**: Nothing relevant.

The value placed on defects found appears to be a little higher for DataApp than for MergoApp, so we put a "+" in the cloud for "value of defects found".

Time spent on other matters:

- Tester is to participate in installation at client site. HIGHER than MergoApp.

The "Time spent on other matters" cloud ends up with a "+", since this effect will be stronger than it was on MergoApp.

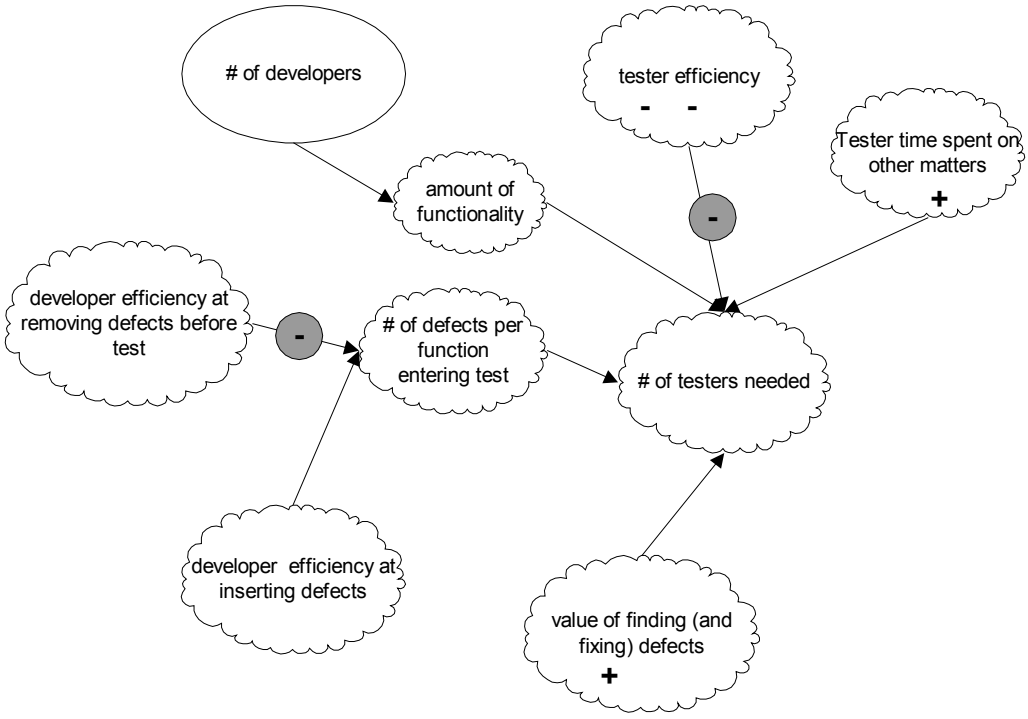


Fig. 3: A worksheet for comparing DataApp with MergoApp.

6. *Combine the results of the various differences*. For each bubble, follow the arrow to the next bubble and put the same number of "+" and "-" in it. Don't forget to change sign if the arrow represents an inverse relationship!
   When you are all finished, the "# of testers needed" cloud should have a bunch of "+" signs in it, and a bunch of "-" signs, as shown in Figure 4.

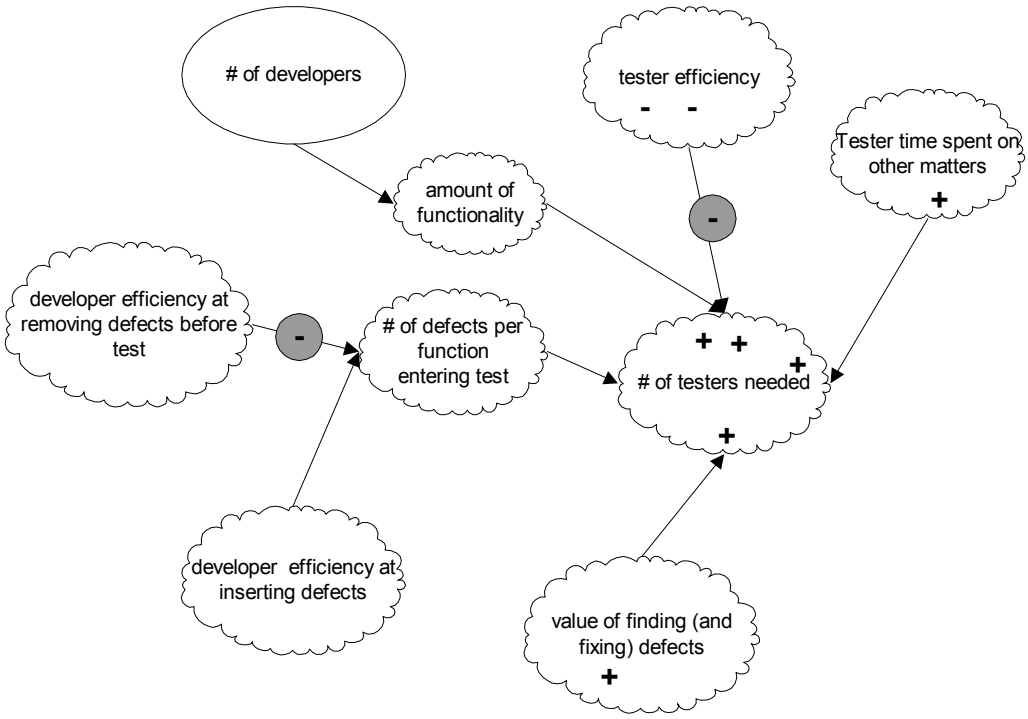


Fig. 4: The completed worksheet for comparing DataApp with MergoApp.

7. *Adjust the number of testers accordingly, using your professional judgment.*
   The number of "+" and "-" in the "# of testers needed" cloud suggests the necessary adjustments to the number of testers estimated in step #4. You will have to decide for yourself whether the effects are of comparable weights or very different weights. In this case, the number of testers clearly needs to be higher than 2.

If you have more than one baseline project, it may be useful to run through this exercise once for each project. We repeated the exercise using DinkyApp as the baseline project. The comparison worksheet is shown in Figure 5.

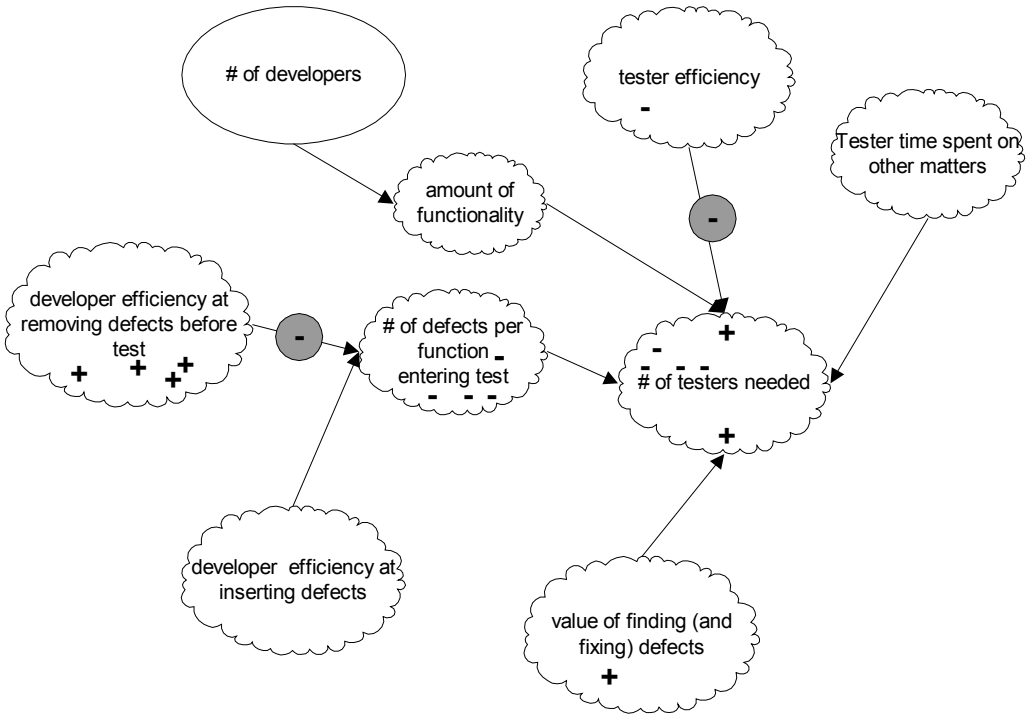


Fig. 5: DataApp compared with DinkyApp

The ratio of developers to testers on DinkyApp was 1:1, so the starting value for number of testers would be 8. The effects diagram suggests that the actual number of testers needed is probably less than 8.

This brackets the estimate for DataApp with a floor of 2 and a ceiling of 8. The margin appears considerable in both cases, so neither 3 nor 7 seems like a reasonable value. That leaves 4, 5, and 6 as possible numbers of testers (assuming that they have to be whole numbers, which usually isn't the case).

The margin of error in the estimate appears to be around 25% to 50%. If you cannot afford this margin of error, then we don't recommend this method.

Incidentally, the actual tester-developer ratio for DataApp was about 4:8.

Summary of Case Studies

As we used this model with this particular case study data, one thing that stood out was how much the experience of the people involved affected the factors and thus the ratio. A more experienced developer or tester is much more likely to follow good process, to be quality conscious, to insist on some documentation or write it herself, and to communicate in a positive way with her entire team.

Other Uses for the Model – Explanations to Management

We found the model and its effects diagram to be useful in explaining a number of things to management. An effective presentation in slide format can be done by starting with the entire effects diagram, and then circling or marking the effects that have changed or that you propose to change.

How Can We Reduce the Number of Testers?

Test managers are often asked to reduce the ratio of testers to developers. This model can be used to demonstrate that development activities and organizational activities play a large role in the ratio, which is often overlooked by test managers. For instance, a large organization with many teams and several layers of integration may need more

testers proportionally, because the developers have more opportunity to make mistakes based on miscommunication with each other.

The technology used in the product itself can also affect the number of testers in ways that management may find hard to understand without a pictorial representation. The replacement of in-house code with an off-the-shelf software application will reduce the number of developers, but the number of testers will not go down proportionally because the off-the-shelf software is not itself perfect. The effects diagram can be used to demonstrate that there are, in effect, invisible and uncountable developers who have inserted defects into your system via the off-the-shelf software.

More Testers, Worse Code?

We found the model useful in analyzing or explaining some situations that defied the ordinary logic of the tester-developer ratio.

For instance, some STMR members observed instances in which increasing the number of testers (thus raising the ratio) resulted in more defects inserted and yet more need for testers. In one instance[5], the increased number of testers caused the developers to start skipping parts of unit test and generally take less responsibility for the quality of their code, in the belief that the increased number of testers would pick up the errors. This reduced the developer efficiency at removing defects before test. Using the model to analyze this situation suggests a positive feedback loop, which could be dangerous.

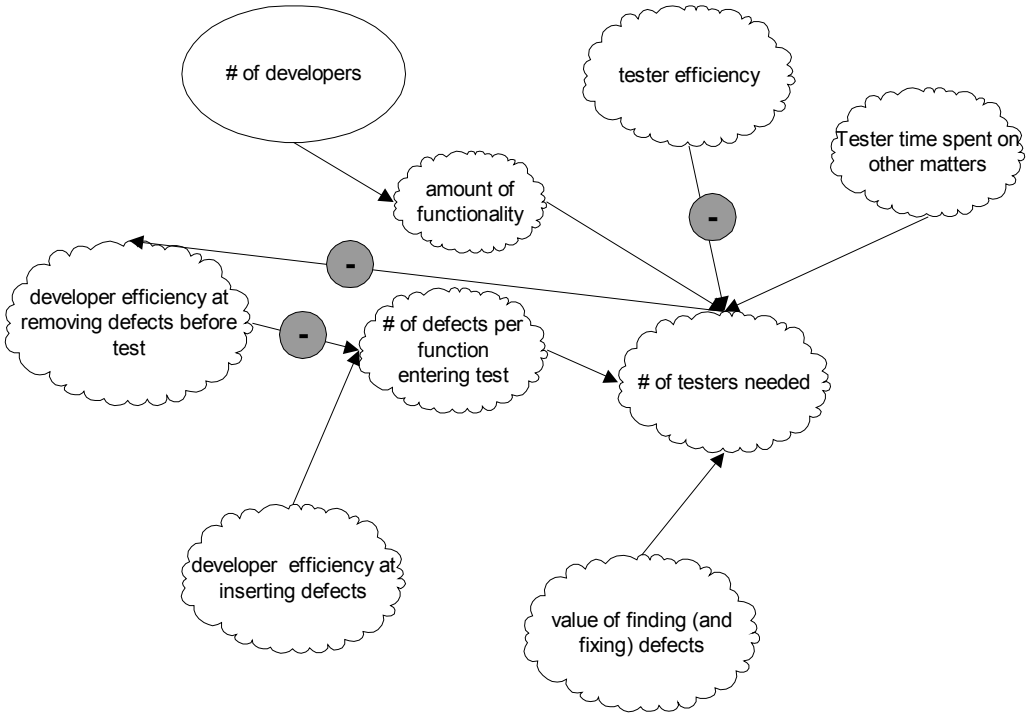


Fig. 6: Effects diagram of a feedback loop between number of testers and developer efficiency at removing defects before test

[5] "Better Testing, Worse Quality?" by Elisabeth Hendrickson, presented at SM/ASM 2001, San Diego, CA. Available on her website at QualityTree.com.

Fewer Testers, Better Code?!

Another interesting situation was the surprisingly low ratio of testers to developers on a few good, solid products. The STMR group collectively reported a handful of instances in which the tester to developer ratio was 1:10 or less and the product was successfully released with appropriate quality levels.

This counter-intuitive result appears to result from the effects of the development staff placing a high value on finding and fixing nearly all defects, and the sensible observation that they wouldn't have to fix what they didn't insert in the first place. The developers adopted careful practices (usually by choice) that resulted in either never inserting the defects in the first place or finding the defects before the product entered test.

The high value placed on the defects may be a result of an unusually high potential impact of errors (such as in life-critical software), but it also occurred in other circumstances where possibly the developers themselves were deciding to place a high value on finding the defects.

There were some common factors observed in these situations:

- The developers fairly carefully followed the usual recommended practices – requirements, design, coding, and unit testing.
- In some instances, the product was intentionally constrained in order to reduce errors – for instance, was marketed to run on only one operating system even though theoretically capable of handling a family of operating systems. Products that run on a multiplicity of platforms allow developers to insert many more defects per hour of work.
- The developers were unequivocally responsible for the quality of the release. The testers were hired by the developers, and the developers themselves were expected to understand testing – in some cases, were the primary test planners.

To a testing manager, this scenario may be counter-intuitive. The lowest tester-developer ratios were achieved on the products with highest expectations for reliability, and the test group was not only unimpressive in terms of skills, but in at least one case no one was identified as a tester! Obviously some of the developers were indeed performing testing, but estimates of developer time spent on testing still suggested that the tester-developer ratio was far lower than usual. The key factor here may be the perceived value of defects found, which influenced the processes followed in development (see Figure 7).

It's not clear that this ratio actually minimized the number of people needed per function point delivered (and therefore cost), although current software engineering wisdom would certainly say it does.

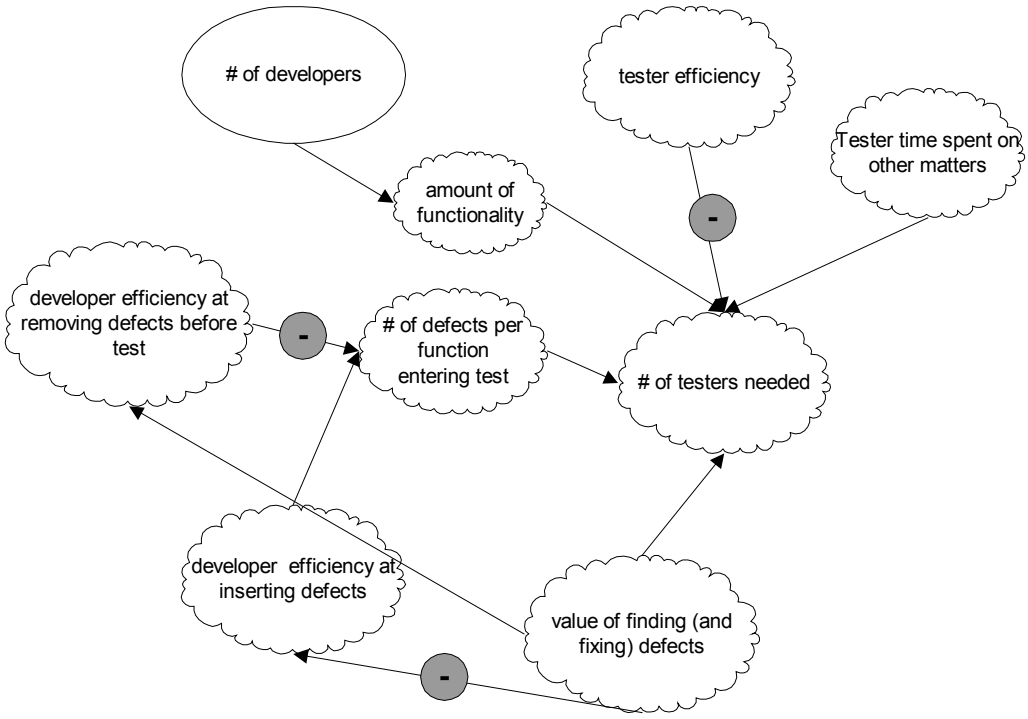


Fig. 7: Effects diagram of the results of developers placing a high value on defect-free software

Summary

The ratio of testers to developers on past projects in a well-known domain can be used in conjunction with an analysis of effects on the relative number of testers vs. developers to roughly predict the number of testers needed for future projects. This method is most useful when information on the functionality and features of the proposed project is not available, or when a quick estimate is desired and a large margin of error is acceptable.

The effects diagram can also be used to graphically demonstrate the probable effects of changes in people, organization, product, or process factors when making presentations to management.

Appendix – The List of Potential Factors

This list is derived from work done at STMR 3. You can use these lists to consider the differences that might affect the tester-developer ratio on a given project as compared to a baseline project.

Developer Efficiency at Removing Defects before Test

These factors *increase* the number of defects that get by the developers.

People factors:

- Inexperience with unit testing or review techniques.
- Over-confidence.

Organizational factors:

- Upper management belief that testers can discover all types of errors more efficiently than the original developers and therefore developers should not unit test. This is also known as "development already did enough work."

Product Factors:

- Code or design that is difficult to understand and thus hard to test.

Process factors:

- Failing to unit test.
- Failing to use reviews and inspections appropriately.

Developer Efficiency at Inserting Defects

These factors *increase* the number of defects inserted by the developers, which *increases* the tester to developer ratio.

People factors:

- Inexperience with software engineering principles, leading to poor design.
- Inexperience with the technologies or business domain of a particular project.
- Belief that working fast is better for the schedule than working carefully.
- Burnout.

Organizational factors:

- Clear communication between developers is hindered by:
    - Geographic separation between developers.
    - Failure to pass on requirements for product to developers.
    - Layers of management or organization between different developers, including outsourcing part of the development.
- Arguing or hostility between teams.
- A culture that does not hold developers responsible for quality of their product.
- Disfranchised engineers with imposed schedules.
- No accepted measure of success or "doneness" except "code complete".
- Death spiral: too many testers can cause developers to get sloppy.
- Little interest in finding patterns of mistakes (defect prevention).
- Many teams, resulting in many layers of integration, which increases the integration problems

Product factors:

- Complexity of product and range of technologies included.
- Incorporation of reused software, third-party custom-written software, or off-the-shelf software.
- Size. There tends to be a nonlinear relationship between lines of code and number of defects.

- Poorly designed code inherited from others – highly coupled, poorly modularized.
- Legacy code, which is not fully understood by the programmers.
- A single design fault can be multiplied into numerous failures when the product:
    - is translated or localized,
    - will run on a wide variety of platforms, or
    - will interoperate with a wide variety of other products.

Process factors:

- Use of powerful programming tools which allow each programmer to generate more code
- Insufficient configuration and build tools or process.
- Poor planning, resulting in developers in a hurry.

Defects Found per Tester

Factors that *decrease* the number of defects found per tester, and thus *increase* the ratio of testers to developers.

People factors:

- Inexperienced or poorly trained testers.
- Poor attitude or morale.
- Testers do not have the specific knowledge for this particular job. For instance: exploratory testing, familiarity with user domain, performance testing, combinatorial methods, or knowledge of a range of appropriate patterns for test design.
- Developers do not know how to write code to be easily testable.

Organizational factors:

- Barriers to communication with development staff.
- Inappropriate use of outsourcing of test, as in duplication of work between outsourcing and in-house test, not anticipating overhead in coordinating testing, using insufficiently skilled outsource test house.
- Culture of mistrust – time is spent gathering data to protect oneself later.
- Disagreement on role of testing – time is spent on arguments regarding who is responsible for what.
- Expectation of superfluous status reports, metrics, and so forth that do not help make testing more efficient nor assist the programmers.
- Insufficient time allowed for test group to develop the appropriate test tooling and collective knowledge and processes to be optimally efficient.
- Teams are constantly reshuffled, leading to loss of knowledge and barriers to communication.
- Lack of adequate equipment, work space, or ergonomics for the test team.
- Testers handle multiple projects simultaneously, leading to loss of time on task switching. (Some sources estimate a 5-10% loss for each additional project).
- Automation group is disconnected from the needs of the organization and burns resources without corresponding return on investment.

Process factors:

- Code is submitted to test before it works at all, wasting test time on installing broken code.
- Regulatory environment requires a lot of extra documentation. (e.g. ISO 9001, FDA approval)
- Test plan and tests will be sold along with the product, and therefore must look nice.
- Testing will be taken over by totally unrelated group, requiring much greater communication time.
- Test group has not acquired, built, or learned to use appropriate tools, such as test automation.
- Regression testing is repeated unnecessarily just to clock hours (often due to misuse of metrics)
- Test group does not store their tests and reuse them when possible.
- Testing does not start until code complete, which cuts amount of time to find defects, not efficiency of tester.
- Production of useless documentation of any kind, including metrics that don't guide testing effectively.
- Lack of metrics that would guide testing more effectively.
- Lack of test planning to use time most effectively.
- Defect tracking process or tools not understood by development staff.

Product factors:

- Needs a lot of automation to execute any test. (e.g. no user interface, like firmware.)
- No test interfaces built in, or insufficient or unclear error logging.

Value of Defects Found

These factors *increase* the value of defects found, which *increases* the ratio of testers to developers.

Product:

- Expensive to fix problems after release. (e.g. firmware.)
- Regulatory requirements are stringent.
- Will be changed frequently, thus maintainability is important.
- Has high need for reliability or safety.
- Will have long support life.

Customer expectations:

- Has competition giving users the choice to use the product or not.
- Is expected to be a professional-looking product, as opposed to having only internal users.
- Is for actual use, rather than being a demonstration or sample.
- Has customers who are unusually particular - adopters, resistant, cultural differences.
- Customers would rather have a good product later than a buggy product now.
- Competing products are less buggy than ours.

Tester Time Spent on Other Matters:

If testers spend time on these tasks, the ratio of testers to developers *increases.*

- Inspecting requirements
- Inspecting code
- Writing requirements
- Preparing material for customer support
- Participating in rollout of product
- Writing product documentation
- Taking support calls
- Handling configuration management and builds
- Writing installers
- Making site visits
- Running clinical testing
- Supervising beta programs
- Testifying or preparing for court cases involving your products
- Researching customer use models

Remote Communication on Project Teams: When to be Face-to-Face

Prepared for the 2001 Pacific Northwest Software Quality Conference

by Jean Richardson and Lisa Burk

Remote communication, particularly email, while increasingly prevalent, is not always the best "channel" for dealing with conflict and building trusting working relationships. While "flaming" is easily recognizable as a conflictual response, the purpose of this paper is to discuss and recommend mitigating techniques for less extreme indicators of conflict in online interpersonal transactions. We believe that the need to be face-to-face in a conversation is directly related to the proportion of relational to task-oriented content in the transaction. And, noticing the "transitional moment" – that moment when an online conversation moves to conflict – is key to being successful in dealing with conflicts that surface online.

It is easy to acknowledge that email is a ripe breeding ground for conflict escalation. Most of our readers are likely to be very frequent users of email, likely have been for years, and may use email as their primary method of communicating, particularly at work. For most of the business world, however, email as a dominant form of workplace discourse is a relatively new phenomenon, occurring primarily within the last five to seven years. It's taken thousands of years for human beings to develop the interpersonal communication rules, norms, and a whole host of non-verbal cues to aid in interpersonal communication. In contrast, virtual communication has had barely enough time to develop shared meaning about this new form of human communication.

Non-verbal communication and the Social Construction of Virtual Communication and Communities

The use of words is extremely important in the human communication process, but words are a small part of how people have learned to communicate. A large portion of our communication with one another involves using non-verbal cues, such as gestures, tone of voice, posture, the clothes we wear, and the ways in which we touch.

Most of us are not conscious of the non-verbal channels we use to communicate, even though they comprise the majority of the way we communicate in a face-to-face discussion or conversation. Albert Mehrabian found in his research in the 1970's that there are three ways we communicate.[1] Mehrabian found that only about 7 percent of the emotional meaning of a message is communicated through explicit verbal channels. About 38 percent is communicated by paralanguage, which is basically the use of the voice. About 55 percent comes through nonverbal, which includes such things as gesture, posture, facial expression, etc. So it is behavior rather than spoken or written communication that creates or represents meaning. The richness of nonverbal

communication cannot be adequately represented in text and punctuation is less precise than intonation.[2, 3, 4] This may be why we are seeing more misunderstandings as we increase our use of virtual communication.

Berger and Luchmann, in their classic communication text, *The Social Construction of Reality*, make the case that human beings create their own reality through constructing shared meaning about their communication and communities over a period of time.

> *"Because they are historical products of human activity, all socially constructed universes change, and the change is brought about by the concrete actions of human beings." . . . "...concrete individuals and groups of individuals serve as definers of reality. To understand the state of the socially constructed universe at any given time, or its change over time, one must understand the social organization that permits the definers to do their defining."*[5]

> *"In order to maintain subjective reality effectively, the conversational apparatus must be continual and consistent. Disruptions of continuity or consistency ipso facto posit a threat to the subjective reality in question."*[6]

Therefore, we are creating a new subjective reality with virtual communication in a very short amount of time. Most companies have no formal rules about virtual communication. Michael Hattersley, conducted a brief survey of major companies (for example, General Motors), and found that very few have formal e-mail guidelines and policies.[7] Ultimately, all interpersonal realities are negotiated, but when our companies and communities don't create new rules for our virtual communication, there is no common basis for these negotiations. We, then, as individuals, construct our own relatively idiosyncratic realities, or rules. This can cause waves of misunderstandings if we do not share the same virtual communication rules and norms.

We also are creating new virtual communities, where virtual communication rules and norms are being developed at an accelerated pace. In "The Virtual Community," Howard Rheingold describes his over 15 year experience of plugging into a virtual community – the WELL (Whole Earth 'Lectronic Link). First he defines a virtual community:

> *"Virtual communities are social aggregations that emerge from the Net when enough people carry on those public discussions long enough, with sufficient human feeling, to form webs of personal relationships in cyberspace."*[8]

He then goes on to describe his experience and how it has changed over time:

> *"The idea of a community accessible only via my computer screen sounded cold to me at first, but I learned quickly that people can feel passionately about e-mail and computer conferences. I've become one of them. I care about the people I*

> *met through my computer, and I care deeply about the future of the medium that enables us to assemble."*[9]

Rheingold also sees an increase in virtual communities, and therefore in virtual communication. He sees computer use as inevitably leading to the construction of online communities and questions whether this will lead to rebuilding what has been lost as a result of suburbanization or if it is just another life-denying "simulacrum of real passion and true commitment to one another". [10, 11]

Rosabeth Moss Kanter, in her new book *Evolve: Succeeding in the Digital Culture of Tomorrow*, sees that there are both positive and negative aspects to these new Internet communities.

"This poses three challenges to everyone engaged with the Internet:

1. The Internet can greatly empower people and connect people, but it can also isolate and marginalize them.
2. The Internet can enable user communities to form and grow, but it can also use them to attack and deny.
3. The Internet can help build businesses and communities, but it can also destroy them."[12]

Will virtual communication and communities replace face-to-face communication and live communities? One thing is for certain, we are losing face-to-face communication with our co-workers, and most of our investigations below lead us to the conclusion that virtual communication is appropriate for some types of communication, but there a certain contexts where face-to-face communication is critical.

Effects of Losing Face-to-Face Communication

While there are distinct advantages to email communication and the experience of many avid e-mailers has demonstrated that a surprising depth of intimacy can be achieved in online relationships, for many people there is a sense of isolation, a loss of face-to-face communication skills, and an increase in negative thoughts conveyed and a new language created.

Hallowell is a psychiatrist who has been treating patients with anxiety disorders for years. He finds that as electronic communication increases, the human moment decreases and changing the landscape of work for the worse. "Human beings are remarkably resilient. They can deal with almost anything as long as they do not become isolated. …When human moments are few and far between, over-sensitivity, self-doubt, and even boorishness and abrasive curtness can be observed in the best of people. Productive employees will begin to feel lousy and that, in turn, will lead them to under-perform or to think of looking elsewhere for work. The irony is that this kind of alienation in the

workplace derives not from lack of communication but from a surplus of the wrong kind. The remedy is not to get rid of electronic (communication) but to restore the human moment where it is needed."[13]

He then describes the human moment: "an authentic psychological encounter that can happen only when two people share the same physical space. I believe that it has started to disappear from modern life – and I sense that we all may be about to discover the destructive power of its absence. The human moment has two prerequisites: people's physical presence and their emotional and intellectual attention."[14]

He then states why we may avoid picking up the phone or walking down the hall. "Human moments require energy. Often, that's what makes them easy to avoid."[15]

Hallowell describes another phenomena when the human moment fades from or lives – toxic worry. "…electronic communications remove many of the cues that typically mitigate worry. Those (non-verbal) cues are especially important among sophisticated people who are prone to using subtle language, irony, and wit…..Toxic worry is anxiety that has no basis in reality. It immobilizes the sufferer and leads to indecision or destructive action. It's like being in the dark."[16]

Apparently our brain chemistry may be altered over time when we lose the human moments. Hallowell notes that scientists don't know the whole story yet, but they do know that… "positive human-to-human contact reduces the blood levels of the stress hormones epinephrine, norepinephrine, and cortisol….Furthermore, scientists hypothesize that in-person contact stimulates two important neurotransmitters: dopamine, which enhances attention and pleasure, and serotonin, which reduces fear and worry. Science, in other words, tells the same story as my patients. The human moment is neglected at the brain's peril. [17]

> *"...in the last ten years or so, technological changes have made a lot of face-to-face interaction unnecessary. I'm talking about voice mail and e-mail mainly – modes of communication that are one-way and electronic. Problems that develop when the human moment is lost cannot be ignored. People need human contact in order to survive. They need to maintain their mental acuity and their emotional well-being."*[18]

Can virtual communication decrease our ability to actually communication face-to-face? In the two examples below, our answer is – yes. The brain is similar to a muscle. It needs exercise to thrive. When we don't practice our face-to-face communication skills, we either get "rusty" or lose them. Hallowell cites an example of a patient who came to him because she felt like she was going "brain dead:"

> *"She consulted me because she actually thought she was losing her memory. In meetings, words were not coming to her as quickly, and decisions she had once*

> *made in a snap were now taking her hours or days…A few simple tests conducted in my office reveled that Lynn's brain itself was in fine shape. Her work habits (relying more on one-way electronic communication) were diminishing her brain's performance. Your psyche, just like your muscles, needs rest and variation to perform at its peak."*[19]

Individual team members can begin to feel isolated and under-stimulated, resulting in poor face-to-face communication when team members come together:

> *"I work at my home exclusively on my computer…it's the main link to my office in Eugene. We talk on the phone maybe once a week, but we exchange e-mails 10, 15 times a day, at least. I can't remember the last time I went to a bank (or went out of the house)…I'll go out and get the dry cleaning and it's hard for me to make sentence…I sort of resort to 'Me.' 'That.' 'Give it to me.' Like cavewoman stuff."* [20]

When conflict is present, and particularly as it escalates, more, rather than less, complex communication skills are required.

Hallowell also cites several studies that indicate that depriving human beings from other human face-to-face contact produces sensory deprivation, an altered sense of reality, higher death rates, and an overall damage to a person's emotional health.[21]

Another negative effect that can occur when we increase virtual communication and decrease face-to-face communication is an increase in negative communication. For some reason, people are willing to voice more negative thoughts to another person using the electronic medium than we ever would face-to-face. Maybe we feel we are "protected" behind the computer screen, or maybe it's a way for people who normally avoid conflict to "get it off their chest." Take a look below at the comments on this phenomenon. [22, 23]

> *"What it boils down to is: This is much more than writing. It's about human nature. Everything that is human has simply bubbled up and is in front of us now. So people think, 'Why are we having so many problems in this company with e-mail?' Because it's not about e-mail. It's about people. And they'll find a way to express their feelings, passive-aggressively or in other ways, in e-mail that perhaps they haven't been able to do face-to-face. A lot of people hide behind it and use it as weapon. It can be used as a tool or a weapon."*[24]

In an April 29th New York Times article, electronic message boards are likened to "The Electronic Water Cooler" – a public forum where employees hold conversations about other employees (often negative), with one major difference. Where the old water cooler conversations could be kept somewhat private, electronic message boards are very public and leave a formal, electronic trail that often puts the company at great liability. [25]

> *"On message boards for particular companies, …some employees are anonymously expressing thoughts they would not dare say out loud. They are freely showing their prejudices or denouncing other employees by name, sometimes accusing them of incompetence or misconduct or recounting salacious rumors about their sex lives".*[26]

The following is an example of an online message board posting from a thread that Startec Global Communications cited in the article:

> "*It's time to go. You have been transferred from dept. to dept. Why? You continue to screw up and …will not lay you off. You have become a worthless, ineffective manager without a cause. Everyone laughs behind your back. No one has any respect for you. Do yourself a favor and leave."*[27]

Virtual communication is also influencing the negotiation process. The Harvard Business Review ran an interesting interview with Harvard Business School Professor Kathleen Valley on what happens when negotiations are conducted via e-mail.

> *"The norm in face-to-face negotiation is something we call "openness script" – your instinct is to share. What we see as the norm in e-mail, by contrast, is something we call "haggling script" – you hold information much closer to the chest.*
>
> *"With e-mail, negotiations are considerably more likely to degenerate into an unpleasant exchange… When the interaction is purely electronic, people are more willing to escalate conflict…*
>
> *"In a recent study comparing e-mail, telephone, and face-to-face negotiations, we found that when people meet face-to-face, the most frequent outcome is a mutually beneficial agreement. When people talk over the phone, the most frequent outcome is that one party takes the greater share of the profits; it's asymmetrical. With e-mail, the most common outcome is impasse. We found that 50% of e-mail negotiations end in impasse; only 19% end that way in face-to-face negotiations."*[28]

With all these negative aspects of email communication, is there a positive side? Certainly. Valley states that…"e-mail can be much more efficient. You avoid having to travel, organize meetings, play phone tag and all the attendant costs….you can communicate when you feel comfortable communicating; you don't have to rush a response or a counteroffer….you just have to weigh the greater efficiency against the negative factors."[29]

An informal survey of regular telecommuters on the STC-Telecommute[1] email mailing list surfaced the positive aspects of asynchronous communication for its ability to allow time for tempers to cool before responding to a potentially inflammatory message and to take one's time in crafting a thorough response. One respondent noted that side conversations, which can be very annoying in face-to-face meetings, could easily be "taken offline" in a secondary thread without interrupting the flow of an online conversation. Another respondent noted that she found it easier to deepen informal relationships online because the technology itself was a drastic improvement over the logistical impediments to frequent written correspondence via "snail mail" – specifically, printing or writing, enveloping, addressing, stamping, and mailing the communication. And email provides the potential for a much more immediate response time than does regular postal correspondence.

Dr. John Suler of Rider University in his paper *E-Mail Communication and Relationships* notes that email "creates a context and boundary in which human relationships can unfold . . . Avid e-mailers have developed all sorts of innovative strategies for expressing themselves through typed text. A skilled writer may be able to communicate considerable depth and subtlety in the deceptively simple written word. Despite the lack of face-to-face cues, conversing via e-mail has evolved into a sophisticated, expressive art form." Further, Suler notes that "an advantage of email conversations over face-to-face ones is that you have the ability to quote parts or all of what your partner said in his previous message."

In *The Virtual Community* Rheingold notes, as did one respondent on the telecommuters' mailing list, that "Some people– many people– don't do well in spontaneous spoken interaction, but turn out to have valuable contributions to make in a conversation in which they have time to think about what they have to say. These people, who might constitute a significant proportion of the population, can find written communication more authentic than the face-to-face kind. Who is to say that this preference for one mode of communication– informal written text– is somehow less authentically human than audible human speech?"[30]

In his article "text talk," Dr. Suler examines several environments where typed text is the norm for communication – primarily on-line chat rooms.

> *"TextTalk on online chat environments has evolved into a fascinating style of communication. In some ways it is strikingly similar to face-to-face dialogue. In other ways, it is quite unique. Many of its unique qualities revolve around the fact that it is an austere mode of communication. There are no changes in voice,*

[1] This list is sponsored by the Society for Technical Communication. For more information see http://www.stcwvc.org/sigs/sigs_tele.htm.

no facial expressions, no body language, no (or very little) visual/special environment as a context of meaning. There's just typed words. Some people find this experience too sparse. They feel disoriented, disembodied, adrift in a screen of silently scrolling dialogue. Other people love the minimalist style of TextTalk. They love to see how people creatively express themselves despite the limitations. They love to immerse themselves in the quiet flow of words that feels like a more direct, intimate connection between one's mind and the minds of others. Almost as if the other is inside one's head. Almost as if you are talking with a part of yourself. Without the distracting sights and sounds of the face-to-face world, TextTalk feels like a more pure communication of ideas and experiences."[31]

"The terse style of talking in chat environments can result in either superficial chat, or a very honest and 'to the point' discussion of personal issues. One doesn't have the verbose luxury of gradually leading the conversation to a serious topic, so self-disclosures sometimes are sudden and very revealing. The safe anonymity resulting from the lack of face-to-face contact – as well as people not knowing who you 'really' are – also contributes to this honest and open attitude."[32]

Are There People Who Are Not Suited to Virtual Communication?

Some people may be better suited than others to communicate in virtual environments. Just who are those people? In *Managing Telework*, the author lists the following teleworker traits. "Aside from the characteristics of a person's job, some people work out better than others as teleworkers." The suggested traits include: [33]

- Self-motivation
- Self-discipline
- Job skills and experience
- Flexibility and innovativeness
- Socialization
- Life cycle stage
- Ability to balance family and work
- Ability to avoid compulsions, i.e., overeating, drug abuse and workaholism.

The author also notes that informal communication is often as important as the formal communication that takes place, for instance, in meetings.[34] He suggests that very extroverted employees are not likely to make good frequent telecommuters. [35]

Some companies are beginning to screen potential applicants for telecommuting positions.

> *"Merrill Lynch runs a telecommuting lab to acclimate candidates for the alternative (virtual) workplace before they formally adopt the new style of working. After extensive prescreening, employees spend two weeks at work in a simulated home office. Installed in a large room equipped with workstations in their conventional office building, prospective telecommuters communicate with their managers, customers, and colleagues solely by phone and e-mail. If they don't like this way of working, they can drop out and return to their usual workplace."* [36]

Trust: The Hallmark of Effective and Efficient Virtual Communication

The fundamental ingredients of trust in any working relationship include reliability, consistency and integrity: I can count on you to follow through with what you have said you will do; I can predict a similar response given a similar situation; and I can count on you being honest. Trust, as described here doesn't happen overnight, and it may be particularly fragile in a virtual communication environment. This kind of trust builds slowly, through a series of shared experiences where expectations are met, belief in each other is validated, and individuals find they can depend on the predictability of each other's behavior.

The authors of this paper found many citations regarding the issue of trust in virtual communication. Overwhelmingly, all agree that face-to-face relationship building activities and actions must precede virtual communications and negotiations. Where that is not possible, if the relationship is to continue over an extended period of time or be subjected to periods of stress, a somewhat tentative and rather fragile relationship may develop. Face-to-face contact and conscious relationship building will likely be required to repair the relationship should it become damaged.

Consider the following statements:

> *"Underlying every successful relationship is trust. Without it people become suspicious, noncommittal, uncaring, undermining and jaded – all of which leads to deteriorated and nonproductive relationships……As a telecommuter, establishing unwavering trust in relationships with colleagues and your boss is particularly vital, since distance and the absence of day-to-day interactions can create pressure on relationships that will erode trust."*[37]

> *"[E-mail negotiations work best] when you've already established rapport with the other person….when you have a utility for their outcome. There are a number of ways to build such rapport. Meet the other party face-to-face first; if possible, or at least have a phone conversation. Then continue the negotiation over e-mail."*[38]

> *"If you must use e-mail as your only medium, at least spend some time up front sharing social information….make a general introduction….find some common social ground that makes the ensuing negotiation proceed more smoothly….resist the tendency [to cut right to the chase]…."*[39]
>
> *"Paradoxically, the more virtual an organization becomes, the more its people need to meet in person. The meetings, however are different. They are more about process than task, more concerned that people get to know each other than that they deliver."*[40]

Duarte and Snyder acknowledge "without trust, building a true team is almost impossible." However, they have identified what they refer to as three factors for building "instant" trust in a virtual environment. These three factors are:

- Performance and competence

  This factor is composed of team members having a reputation for performance and results, follow-through, and obtaining resources to address the team's needs, a reputation that may have been built over time before the team members were assembled.

- Integrity

  The authors define this as "the alignment of actions and stated values . . . The two primary behaviors that indicate integrity in a team are: (1) standing behind the team and all its members and (2) maintaining consistent and balanced communication."

- Concern for the well-being of others

  This factor is composed of intentionally and compassionately facilitating the transition of team members as they move on and off the team, as well as a concern and awareness of the team as a whole and its impact on the larger organization of which it is a part.

Duarte and Snyder also advocate for at least an initial face-to-face meeting even with virtual teams. "Currently, no technology can provide the give-and-take feeling of human interaction, and the understanding that develops from a face-to-face meeting. A virtual team leader should lobby diligently for the resources and time for a face-to-face meeting."[41]

Overwhelmingly, people writing and expressing their views about virtual communication seem to agree that there is no replacement for face-to-face, human contact. This may change over time, as we all get better skilled at communicating virtually; however, there may be no replacement for the human moment.

Transitional moments: What are they and why manage them

There are "transitional moments" we can recognize and that signal to us that it is time to move from one communication venue to a new one – from virtual communication along a continuum to face-to-face communication. These transitional moments are analogous to radio "sound bytes." They transition us from one reality to another. The commonality behind all of these transitional moments is that we do not have shared meaning about the topic under discussion or the problem we are trying to solve.

While these moments may be fleeting, noticing them and acting on them are critical to our personal effectiveness at work. Poorly managed conflict sacrifices team synergy and productivity. And, as Hallowell states, not attending to the human moment can ultimately do emotional harm to ourselves and others. The most common harbingers of a transitional moment that we have been able to identify are:

1. Shifts in tone of response.
2. Response time changes.
3. Response seems out of alignment with what you've communicated.
4. Correspondent "shuts down".

In this section of this paper we offer the reader some specific skills to manage potential or real time communication difficulties when we identify a transitional moment and some insights about transitioning to other modes of communication to manage a conflict.

Developing a strategy for responding to conflict online

When first joining a team or making a new contact with whom you will engage remotely in the future, it's helpful to find out what your communication partners' communication norms are. In other words, find out whether they prefer to define problems, gather information, provide status, and work toward solutions face-to-face, over the telephone, fax, in email, using online meetings, or in a chat room environment. Learn to gather this information as routinely as you would other contact information such as phone number, email address, URL, and physical address: it can often turn out to be every bit as important.

If you have the benefit of an organizational team meeting at the beginning of a project or regular status meetings, raise the issue of norming communication. Some people prefer periodic status reports in email only, others prefer them over the phone or in face-to-face one-on-ones. Some people are uncomfortable handling any form of conflict, or problem solving, in email. Some prefer to involve multiple team members in an online thread so as to have a record of the discussion which may prove handy when writing a decision document, if your team formally documents significant decisions.

Walking a tightrope without a net

It's not uncommon for teams not to have done the work of exploring and norming their communication at the beginning of a project. Teams are frequently quickly assembled to complete a short-term or deadline oriented project or solve a pressing business problem. Often the management focus is on the problem to be solved, not the way the team will communicate about the problem. When this happens, you can find yourself quickly in the throes of problem definition and resolution without much information on how to proceed when conflict begins to escalate. It can feel like walking a tightrope without the security of a net.

When you notice a transition in the interaction, pause and consider which of the following ways of proceeding would be the most effective:

- Querying carefully in email.
- Picking up the phone and calling your correspondent for clarification.
- Requesting an online meeting or teleconference.
- Requesting a face-to-face meeting.
- Enlisting the assistance of a third party intermediary.

Querying carefully in email

Remember that since email lacks the nuances of tone, gesture, and facial expression, it's very possible to misunderstand your correspondent's, likely, most recent contribution to the discussion. For example, when an email correspondent sends you a potentially inflammatory message, the best way to respond may be to ask for clarification. Do not assume the writer was intending to be offensive. Instead, assume there is some sort of communication disconnect between you.

Picking up the phone

Kathleen Valley offers some simple advice to manage these transitional moments:

> *"If you ask a question and you get a response that seems defensive instead of one that offers more information, don't respond in kind. Don't get angry or personal. Stop typing and pick up the phone."*[42]

Where making a phone call is an option – for instance, you are both in time zones that allow you at least *some* working hours in common – this can be the quickest and easiest way to address a conflict that is initiated online. Of course, you have to be comfortable calling your correspondent and feel that it is possible to effectively address the conflict over the phone.

The telephone offers the additional advantage of providing auditory information during the conversation that email does not. You can hear tone of voice, pacing, pauses, and so on. A person who sounds angry in email may simply sound confused, or even frightened, in a telephone conversation. If you seem to be perceived by your correspondent as imposing or threatening, a telephone conversation is also a way to communicate a non-aggressive stance.

Requesting an online meeting or teleconference

Sometimes it will be helpful to have more than two team members involved in a conversation that results from an emerging online conflict. Arranging for an online team meeting or teleconference can be a way to raise issues in a forum that isolates the factual content and depersonalizes the problem. Online meeting facilities can provide both voice and data so that an online whiteboard or other applications can be shared to document the problem definition and brainstorming around solutions. During a teleconference one or more people can volunteer to take notes and shared meaning can be crystallized for the group with a follow-up email message that describes the agreements reached in the meeting. Also, a facilitator can be assigned to ensure the meeting goes smoothly and that all people have an opportunity to participate.

Requesting a face-to-face meeting

Sometimes the best approach is, finally, a face-to-face meeting. In some virtual environments this can be extremely difficult because team members may be spread across great distances and multiple time zones. (Note that, while videoconferencing is a step better than teleconferencing, even the best technologies are still not truly "real-time" solutions and lack the essence of what a true face-to-face meeting provides. Face-to-face meetings, in the context of this paper, do not include videoconferences.)

Some personal styles, – and it is important to consider the other person's style as well as your own – require a subtler use of the individual's senses when a conflict arises or has progressed to a certain point, particularly if trust has been impaired. While a person whose first preference for dealing with conflict is to have a face-to-face meeting may not be suited to working on a virtual team, even the most experienced, adept remote worker may sometimes feel the need for "the human moment" in defining a problem and moving effectively toward resolution.

Enlisting the assistance of a third party intermediary

You may find it helpful to drop out of the thread for a moment and query another member of the online discussion for feedback. Essentially you're asking: This is my take on that last contribution. Am I reading this right, or did I miss something?

Often, another person's perspective can shed light on the problem. You are sitting in front of your PC in your immediate reality and all it entails – concerns about other deadlines, the last sharp comment from your supervisor in the hallway, the dog barking next door, or that last interruption from a telephone solicitor – the fifth today. All of this can color your response, as can "unfinished business" with the contributor whose remark you found offensive or who seemed to be offended by your last contribution. Another person might read the situation differently – likely without the baggage you are currently carrying. That person might even, as a result of your query, be able to make a contribution to the thread that clarifies the misunderstanding and provides you with the information you need.

This technique, however, is not best used to start another thread or to factionalize against someone who has offended you. It is only best used to actually gain clarification when you feel you may have misinterpreted a comment that need not necessarily be the initiation of an unproductive conflict.

The stop-drop-and-role of crisis conflict management

Sometimes it seems like conflict is escalating like a wildfire, which may be why a hostile, rapidly escalating email-based conflict is called "flaming". Just as in fighting wildfires, there are ways to address a rapidly escalating conflict online. This problem-solving model is used when two or more people have a problem and want to find a solution together and can be applied to online discussions as well as face-to-face discussions:

- W ait a minute
- I nteract, don't react
- N egotiate a solution[2]

When you feel yourself becoming emotionally involved, feeling defensive or like you want to attack, STOP. Calm yourself down. Ask yourself what you really want out of this interaction. Strategize about how to get the other person to talk to you. You might, for instance, draft a few questions to ask the person to help you clarify the facts and intentions in their statement. You will know you are ready to move on to the next step when you believe you can listen to the other person and suspend judgment.

Pose questions to gather information. Remember that most conflicts are based on misunderstandings and nearly as many are based on a too-hastily assumed belief that our positions are more important than our basic interests in the problem. During this phase of the conflict, make every effort to try to understand the other person's point of view by

[2] The WIN model of conflict management was devised by Lisa Burk.

asking open-ended questions. While we all know that these are questions for which there is not a "yes" or "no" answer, it's also important to keep in mind that they are not the kind of question that Perry Mason made popular: "Isn't it true that . . ." This approach does not tend to inspire trust in your correspondent. Be willing to share your point of view in non-blaming ways. Assume that your correspondent is as interested in understanding your point of view as you are in understanding hers.

Finally, work with your correspondent on coming up with multiple solutions together and pick one that meets both of your needs. Agree to try out the solution you have chosen while remaining open to modifying it as needed without blame.

Proactively addressing emerging conflict

When confusions or misunderstandings arise among people who communicate exclusively or almost exclusively through email, tensions can rise quickly because of elements like time delays and lack of a sense of ongoing interpersonal connection and, therefore, a lack of a sense of responsibility to treat the other as an equally valuable being. If your team's working environment has degraded to the point that a dispute has impacted team equilibrium, coaching team members on a proactive communication process can help head off feuds.

Proactive communication[3], when practiced regularly, tends to build strong working relationships and provide a kind of social/emotional credit balance in those circumstances where tempers rise over data lines. This skill can be practiced on many levels from the fundamentally attitudinal to just-in-time crisis intervention. At the highest level, it is an orientation on the world and the individual's place in it. However, it can also be practiced as a set of learned skills, just like first aid and CPR can be practiced without a clear understanding of anatomy and physiology. A few guidelines are in order.

- First, be aware that confusion and even conflict **will** arise. Know that conflict is often based on lack of understanding between conflicting parties. Few people enjoy conflict, but avoiding it or allowing it to degrade to blame placing or escalate to name-calling will not solve the problem and allow you to move forward toward your goals.
- When you sense confusion or conflict arising, STOP. Adopt a position of eagerness to accept responsibility as necessary. This is not the same as placing or accepting blame. Neither placing nor accepting blame is productive in resolving conflict or confusion.

[3] This protocol was developed by Jean Richardson.

- Verbally indicate to the person you are speaking with that you are interested in partnering with them or cooperating to clarify the misunderstanding.
- As much as possible, focus on gathering information with that motive using the following three step process:
  - Describe the observable facts.
  - Interpret those facts and compassionately verify your interpretation with the other party.
  - Evaluate the facts and your understanding of them.

Based on your collection of observable facts, focus on the actions and agreements that will move both of you toward your goals. It is wisest not to try to negotiate this agreement in a context of blame placing or name-calling. Both parties must be aware of their intentions in order for this phase of the process to work, and those intentions must be in alignment with moving you both toward releasing a high quality product on time.

The technology is not the problem

Remote work, telecommuting, virtual teams all require new communication skills. Old school managers who resist gaining these skills are losing ground in the new work environment – and losing valuable employees to teams where virtual communication skills are seen as part of the base skill set. Increasingly, as companies convert teams or whole divisions to home-based workers, it is necessary for individual contributors who may not have otherwise chosen to be teleworkers to develop these communication skills. In some parts of the country – in Oregon, for instance– sound and air pollution concerns have risen to an extent where ecological and quality of life concerns have resulted in legislation that encourages remote work.

Does this mean that we are doomed to seeing an increase in unproductive workplace conflict? No. The technology itself is not the problem. Rather, our lack of thoughtful use of the technology is the problem. Companies such as Mentor Graphics have learned that a well-managed virtual team spread across multiple time zones can have distinct advantages, both in terms of team member recruiting and retention and maximization of time spent on a project. But Mentor has also learned that lack of thoughtful management of this special kind of time or total disregard for the importance of "face time" can result in the disintegration or under-productivity of the team – and the loss of a valuable investment in critical human resources.

An important aspect of using email wisely as a dominant form of business communication is noticing the moment when an online conversation has gone awry and

doing what it takes to get the interpersonal interaction back on the right footing. That can mean:

- Querying carefully in email.
- Picking up the phone.
- Requesting a face-to-face meeting.
- Enlisting the assistance of a third party intermediary.

But **always** using email wisely means focusing on factual content, giving emotions their due, and moving forward with integrity and with the purposes of your role in the interaction in mind.

Sources Cited

1 Albert Mehrabian , Nonverbal Communication Chicago: Aldine-Atherton, 1972.
[2] Regina Fazio Maruca , "How Do You Manage an Off-Site Team?" The Harvard Business Review (July – August 1998), 4
[3] Maruca, 1998, p. 6
[4] Watzlawick, Paul "How Real Is Real? Confusion, Disinformation, Communication" New York: Vintage Books, 1976, pp. 65-66
[5] Peter L. Berger and Thomas Luchmann. The Social Construction of Reality (Anchor Books/Doubleday 1966),116
[6] Berger and Luchmann, 1966, p. 154
7 Michael Hattersly, "Does Bill Gates Know from E-Mail?" Harvard Communications Update Reprint #C9801C: 11.
[8] Howard Rheingold , The Virtual Community (Harper Perennial 1993), 5
[9] Rheingold, 1993, p. 1
[10] Rheingold, 1993, p. 6
[11] Rheingold, 1993, p. 26
[12] Rosabeth Moss Kanter, Evolve! Succeeding in the Digital Culture of Tomorrow (Boston: Harvard Business School Press 2001), 17.
[13] Edward M. Hallowell, "The Human Moment at Work" Harvard Business Review (January-February 1999), 4.
[14] Hallowell, 1999, p. 3
[15] Hallowell, 1999, p. 4
[16] Hallowell, 1999, p. 5
[17] Hallowell, 1999, p. 7
[18] Hallowell, 1999, p. 5
[19] Hallowell, 1999, p. 6
[20] The Oregonian, 10 June 2001: E1-E2
[21] Hallowell, 1999, p. 6
[22] David Angell and Brent Heslop, The Elements of E-Mail Style: Communicate Effectively via Electronic Mail (Addison-Wesley Publishing Company 1994) 4
[23] David Stauffer, "Can I Apologize by E-Mail?" Harvard Management Communication Letter (November 1999): 5
[24] The Oregonian, 8 Feb. 1999. Republished at http:/oregonlive.advance.net/technw/9902/tn99020805.html
[25] Reed Abelson, "By the Water Cooler in Cyberspace, the Talk Turns Ugly," The New York Times, 29 April 2001: 26
[26] Reed Abelson, p.1
[27] Reed Abelson, p.1
[28] Kathleen Valley. "The Electronic Negotiator." The Harvard Business Review (January – February 2000): 2-3
[29] Valley, 2000, p. 3
[30] Rheingold, 1993, p. 23-24
[31] text talk, www.rider.edu/users/suler/psycyber/psycyber.html
[32] text talk, www.rider.edu/users/suler/psycyber/psycyber.html
[33] Jack M. Nilles, Managing Telework: Strategies for Managing the Virtual Workforce (New York: John Wiley & Sons, 1998), 34-37
[34] Nilles, 1998, p. 36
[35] Nilles, 1998, p. 36
[36] Mahon Apgar, "The Alternative Workplace: Changing Where and How People Work" The Harvard Business Review (July – August 1998): 135

---

[37] Debra A. Dinnocenzo, 101 Tips for Telecommuters (San Francisco: Berrett-Koehler Publishers 1999): 101
[38] Valley, 2000, p. 3
[39] Valley, 2000, p. 3
[40] Charles Handy, "Trust and the Virtual Organization" The Harvard Business Review (May-June 1995): 6
[41] Deborah L. Duarte and Nancy Tennant Snyder, Mastering Virtual Teams (San Fransisco: Jossey-Bass 2001) 100.
[42] Valley, 2000, p.3

Sources Consulted

"Communicating with Virtual Project Teams" Harvard Management Communication Letter 3, no. 12 (2000): 9.

"Communication: Should We Write to Be Understood?" http://www.j-bradford-delong.net/email/communication.html

"Creating Successful Virtual Organizations" Harvard Management Communication Letter 3, no. 12 (2000): 10-11.

Balu, Rekha "Please Don't Forward This Email!" http://www.fastcompany.com/invent/invent_feature/email.html

Drucker, Peter, "Drucker on Communication: What the Master has to say about this most difficult business skill" Harvard Management Communication Letter (November 1999): 10-11

Glassman, Audrey. Can I Fax a Thank-You Note? New York: Berkley Books, 1998.

Gunderson, Laura, "E-mail notes prove sticky for Beaverton teacher bargainers" The Oregonian, 11 June 2001: E3.

Imperato, Gina "Action Item – The Write Stuff: A Short Course on Writing Effective Emails" http://www.fastcompany.com/online/12/actionwritestuff.html

Imperato, Gina "Real Project Managers Don't Use Email" http://www.fastcompany.com/online/15/emailnot.html

Keel, Jon "15 Top Tips for Effective Email Communication" http://www.thewritemarket.com/mcnn/keel.shtml

Krattenmaker, Tom. "What's Your Company's Culture?" Harvard Management Communication Letter 3, no. 12 (2000): 1 –4.

Layne, Anni "He Seconds That Emotion" http://www.fastcompany.com/launch/launch feature/emoticon.html

Overholt, Alison "Intel's Got (Too Much) Mail" http://pf.fastcompany.com/online/44/intel.com

Quain, John R. "Quain's Top Tips" http://www.fastcompany.com/online/02/etips.html

Rogers, Carl R. and Roethlisberger, F. J. "Barriers and Gateways to Communication" Harvard Business Review (November-December 1991): 105-111.

Wreden, Nick, "How to Be in Two Places at Once" Harvard Management Communication Letter (October 1999): 3-4.

Facing up to the Truth

Esther Derby
Esther Derby Associates, Inc.
DERBY@ESTHERDERBY.COM

ABSTRACT

The ability to face the truth is critical management ability…but what is the Truth? What happens when two people have different "truths" for the same situation? How can managers navigate multiple meanings?

BIOGRAPHY

Esther Derby is a writer, consultant and facilitator based in Minneapolis, MN. Esther has worked in software development for over 20 years. She's been a developer, a system manager, a development manager, and a project manager. She's the founder and principal of Esther Derby Associates, Inc., a one-woman firm that works with individuals and teams to improve their effectiveness working with complex human systems like software development projects and software development organizations.

Esther has an MA in Organizational Leadership and is a Technical Editor for STQE.

Facing Up to the Truth

Esther Derby
Esther Derby Associates, Inc.
612-724-8114
DERBY@ESTHERDERBY.COM

Facing the Truth

- Have you ever seen a situation everyone knew the software was not going to be ready to ship yet key managers seemed to be in denial?
- When was the last time you watched a project spiral out of control as the project manager watched apparently unaware?

Bad News

- Humans often fail to see “the truth” when it is perceived as bad news.

Why is it so hard hear bad news?

- Filters
  - Our senses filter out much of the data available from the environment.
- Mental models
  - Our mental models organize the data we take in into patterns we recognize.
- Investment
  - “We’ve put so much into this…”
- Stories
  - We all have internal “stories” about the consequences of failure.

- "Bad news" is ***information*** about the gap between the desired state and the current situation.

"There is nothing either good or bad, but thinking makes it so."

William Shakespeare's Hamlet, Prince of Denmark, Act II, Scene 2

How can we do better at facing the current situation?

- Notice your physical state when you hear the "bad news."
  - Are you tense or nauseated? Are you clenching your teeth?
  - Your physical state gives you clues about your internal state.

Emotions aren't the problem

- We all have emotions.
- When we pretend they don't exist or aren't important, they can get in the way.
- Deal with the emotions so that you can bring your intellect to bear on the problem.

Center

- Remember to breath.
- If you can, go for a walk.
- What are you telling yourself about the current situation?
- What is the worst that could happen?
- Do you think you could live through it? (You can).

Look at alternatives

- What options do you have?
  - Run and hide?
  - Quit?
  - Find out who is to blame and kick some butt?

  Let these thoughts scroll by….
- Brainstorm at least three options to move back towards the desired state.
- Can think of any?
  - Find a colleague you trust and brainstorm options.

Multiple Meanings

- What we see depends on where we stand.
- When we are surprised by someone else's perception or reaction, it's often because we don't know their context.

What can he see from where he stands?

- People do not (generally) act in ways that don't make sense to them.
- We may not know how the situation looks from another point of view.
- Try to find out about the other persons context.

Get more information

- Ask questions:
  - "What have you seen or heard that leads you to that belief?"
  - "Where do you see the risks in this situation?"
  - "What would you like to have happen?"

Look for common ground

- Where do you have common interests?
- What are the other person's interests in this situation?
  - Does he have the same objective you do?
- What is the best outcome from this person's perspective?
  - Is it compatible with the outcome you want?

 13

Shifting perceptions

- A frontal assault seldom works.
  - "That's not the way it is!"
  - "You just don't care about quality!"
- State how you see the situation from where you stand.
  - "I see it differently..."
  - "I'm concerned about…"
- Expand the mental model
  - "What would happen if this were true?"

 14

Reasonable people sometimes disagree

- Who really owns the decision?
- Do they have all the information they need?
- Sometimes you have to let go.

Sources

- Congruent Leadership Change-Shop Notes and Readings. Jean McLendon, Daniela Weinberg, Gerald Weinberg. Photocopy. 2001. (see www.geraldmweinberg.com)
- The Image. Kenneth E. Boulding. 1956.
- The Power of Image: Tools for Creative Facilitation. ICA. 1999. (see www.ica-usa.org)
- The Art of Focused Conversation. R. Brian Stanfield, ed. 1997.

PNSQC Speaking Abstract

Jim Heumann
Requirements Management Evangelist
Rational Software Corporation

Is a Use Case a Test Case?

Understanding how a user will interact with a system is important if the goal is to deliver a product that will ultimately meet the users' needs. With the surge in popularity of use cases, software development teams are discovering/rediscovering how helpful this method is in defining and building a user-oriented solution.

Each use case defines a series of steps that describe how a user interacts with a system and what the system does in response. A test case is a set of test inputs, execution conditions, and expected results that test for the achievement of a particular system result. Are these the same thing? Are they like things? Can the use case *be* the test case?

In his presentation, Jim Heumann, Requirements Management Evangelist, Rational Software, will provide an overview of use case methodology and describe how use cases can be applied to increase testing productivity. In addition, the talk will share lessons learned from applying use cases to the software testing challenge at Rational Software.

Attendees will learn:

- What use cases are and how they are used
- How a use case can be used to create a test case
- Lessons learned in applying use cases to software testing

Biographical Information

Jim Heumann, Requirements Management Evanglist at Rational Software, specializes in the front end of the software development lifecycle. Jim has been with Rational Software for over two years, most of which has been spent helping customers understand and implement software processes and tools. Heumann has been in the software business since 1982, having worked on analysis, development, design, training, and project management in several organizations of various sizes and industry segments.

There Are No Stupid Users: It's Time to Face Up to Usability

Joe Auer
ExperienceUsability.com
joeauer@acm.org

INTRODUCTION

Application usability has become a critical success factor for business applications and websites. From a product marketing perspective, customer-facing software is evolving from a technology product into a consumer product. As this happens, user expectations increase significantly. Demands for accomplishing tasks easier, faster, and more conveniently will replace the "wow" factor of a new technology. At the same time, a tight economic situation increases competition, demands higher employee productivity, and focuses closer examination of return on investment. Improving usability by incorporating user-centered development methods will have a direct positive impact.

This paper takes a broad brushstroke of several issues related to usability. The main goals are to whet appetites for further exploration and move beyond the foggy term "user-friendly".

AN EMERGING SPECIALTY

Although specialists in user interface design and human-computer interaction are nothing new, the growth of personal computer and Internet use has led to a dramatic increase in people and companies focused in this area. It's intriguing that there is no common field of study for those drawn to the field. Backgrounds include quality assurance, cognitive psychology, technical communication, library sciences, software development, and marketing. With such diversity, it is no surprise that usability specialists have strong, but dissimilar opinions on everything from term definitions to the number of people to include in a usability test. They can even disagree on their scope of concern. Some see boundaries at the user interface, while others will include all interaction between a customer and a company. What these folks do have in common is a passion to create products that work well for end users.

COMPUTERS AND OUR PREHISTORIC BRAINS

Why can working with a computer be so frustrating? Research by Reeves and Nass [9] substantiate the importance of usability by showing that the human brain has not evolved quickly enough to subvert our subconscious treatment of computers as real people. Their experiments show people unknowingly being polite to a computer, treating female computer voices differently than male voices, and responding to a computer's praise and criticism. Their work accentuates the necessity of designing an interface that interacts respectfully with users.

USABILITY IS TOUGHER THAN ONE WOULD EXPECT

It's difficult to build an application or website that is highly usable. Common obstacles to usability include users not having a voice on development teams, using opinion over action as an information source, and unfavorable attitudes toward usability.

Including the User's Perspective in The Development Process

It takes a lot of highly skilled and talented people to produce software and get it in front of a user. Most team members have a sincere desire to produce a successful and usable product. Unfortunately, even though team members think they can represent the user's interests; no one can do it effectively. Marketing people understand what people will buy, which has limited value in how software is effectively used. Developers love details, while most users do not. And because of their close ties to technology, a developer's design may unconsciously be the quickest or most similar to a solution used in the past. A technical communicators focus leans toward software learning rather than use.

People are Problematic Information Sources
Although marketing researchers and business analysts often gather data through interviews or focus groups, these have been shown to be misleading and error-prone. One reason is that people often don't do exactly what they say they do. For example, when describing a work process, several steps may be unintentionally skipped because they are automatic. In addition, a recent study by researchers at Intel [11] found that people's assessment of task difficulty changed when asked while they were doing the task versus after they were finished. The post-task evaluation was consistently higher. The only proven effective method of collecting task information is through observing users.

In addition, users work under a specific mental models and domain language. Forcing them to translate their language for a person who is unfamiliar with the work can also lead to errors.

Possible Negative Perception of the Term Psychology
As mentioned earlier, many usability practitioners have expertise in Cognitive Psychology. Perhaps some individuals on development teams are uncomfortable involving expertise that may be perceived as a "soft science". This may sound far-fetched, but why would two very popular books remove the term "Psychology" from the titles in newer versions.

1. The Design of Everyday Things by Donald A. Norman was originally published as The Psychology of Everyday Things
2. Influence: Science and Practice by Robert B. Cialdini was previously released as Influence: The Psychology of Persuasion

Survival of the Fittest Attitude
Although rare, there are individuals on development teams who believe there really isn't a problem with software design, some people are just too stupid to use their products. A great non-software example of this came out of the Florida ballot controversy in the 2000 Presidential Election. Figure 1 shows the butterfly ballot used in Palm Beach County, Florida. As the story goes, some voters who intended to vote for Al Gore by choosing the hole below the one for George W. Bush accidentally voted for Pat Buchanan, since this hole was directly beneath the Republican punch. Figure 2 shows was a cartoon sent extensively over email poking fun at the ballot. More concerning, however, is Figure 3, which places the blame on the voter, implying they are too stupid to vote.

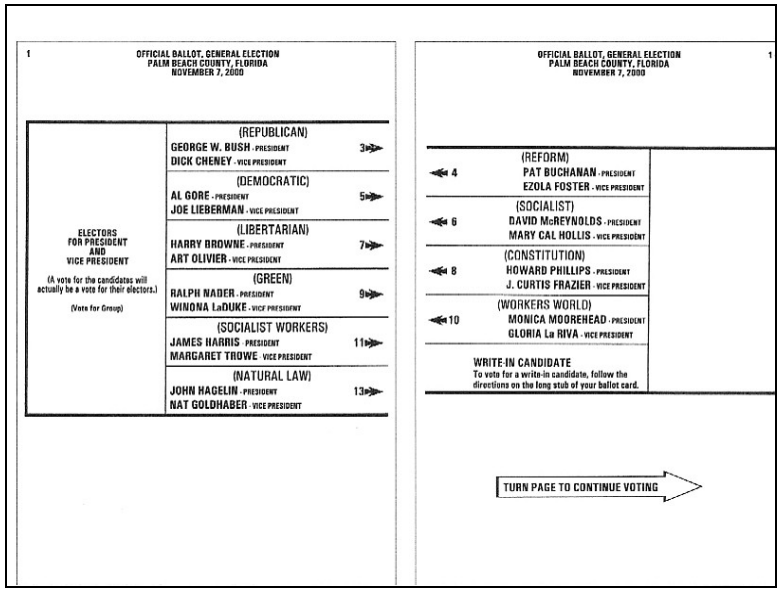


Figure 1: 2000 Presidential Election Ballot From Palm Beach County, Florida
Courtesy of www.bricklin.com

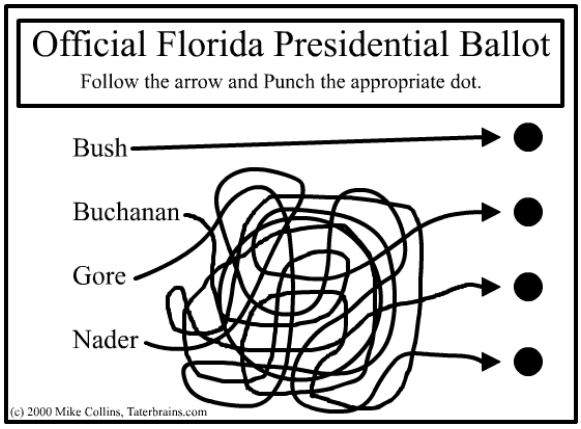


Figure 2: Political Cartoon of Florida Ballot – Stressing Design Problem
Courtesy of www.taterbrains.com

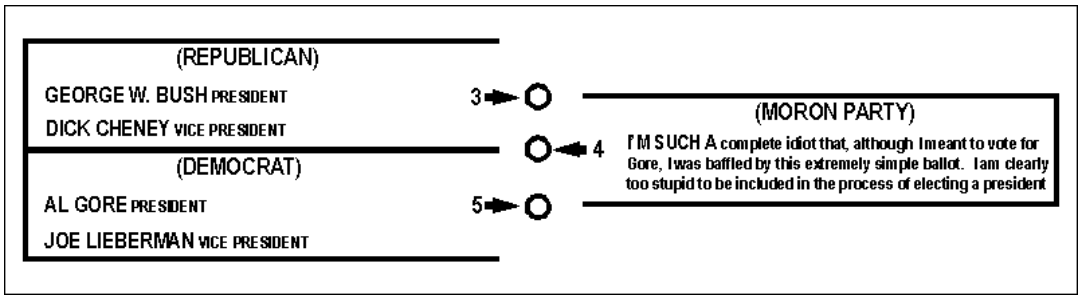


Figure 3: Political Cartoon of Florida Ballot – Stressing Stupid Voter
Courtesy of www.hardocp.com

Occasionally, attitudes like the one reflected in Figure 3 can be found on development teams. Imagine the loss in sales for a commerce website where it's believed that some users might be too stupid to complete the checkout process.

COMPONENTS OF A USER-CENTERED DEVELOPMENT PROCESS

Like other aspects of quality in software development, usability cannot be an afterthought. It takes a lot of research, time, energy and creativity. Attempts to fix weak usability with training and documentation are costly and ineffective. Decisions that affect usability are continuously made throughout the development process, consciously or not. Incorporating user-centered methods into analysis, design, construction, and testing leads to improved usability. Furthermore, the process becomes most effective through continuous improvement by reevaluating and enhancing designs during subsequent development cycles.

Learning About Users and Their Work Tasks

To successfully design a product for users, it is necessary to understand their mental model, contextual language, and the tools they use. As mentioned earlier, standard interviews and focus groups may only reveal part of the story. An effective alternative is conducting a field study. These are visits to a user's workplace to collect artifacts, observe work tasks and conduct interviews. The field study team typically brings video and audio equipment to record activity. The tapes are reviewed after the visit to clarify details.

Another aspect of a field study is learning about the physical environment. It is usually assumed that users are seated at a desk, with a large, clear monitor, in quiet surroundings where their full attention is on the application. This may be far from reality. In some cases, people must stand to work or read data from several feet away. Will one person use the system exclusively or do

several users share? Will users have both hands free? These are easy to answer after observing the work environment.

Data from a field study can be used to create a set of user profiles for the different types of users that will interact with the software. A profile includes items such as the experience level, goals, preferences, and priorities. One effective way of displaying this data is creating a user matrix that includes each user type in a table with their associated characteristics. This tool is helpful when considering design alternatives and how they may impact each of the user types. Some designers create a persona to go along with the profile [2]. The persona is given a name, photo, and history to make them more real. During design meetings, team members refer to the persona by name when considering the impact of an option. This helps members remove their personal opinion from the design discussion and focus on the target user.

Next, task scenarios and/or use cases are created for particular work tasks. In addition to describing the steps of the task, its frequency and importance are noted. These factors are used to create interaction models. For example, *frequently* performed tasks should be easily accessible and fast. *Infrequently* performed, but *important* tasks should have additional assistance since users will not likely become experts in these activities.

Evaluate Existing Products

Valuable measurement data can be obtained by conducting usability tests (discussed in detail later) on current products in use as well as competitors' products. Any usability issues identified can be addressed in the new design.

When there is an existing production application, an excellent resource is the technical support group. Features that get the most support calls are great candidates for improvement. Also, support personnel can identify features that users are asking for.

Knowledge about when users typically open Online Help provides another usability insight. While observing users performing new or unfamiliar tasks, researchers found that users typically have a single goal in mind and look over the current screen to figure out what to do to get them to their goal. If their choice is good, they continue on. If not, they try and go back. Only if they get completely lost or frustrated will they consult the help file. Including a small amount of assistance text on the window or web page may guide the user toward their goal and reduce the need to consult the Help system.

Although some companies use website logs to evaluate usability, they are only partially effective. They do a good job showing the most popular pages and browsing routes, but cannot tell the story of why people left the site or if there was something else they were trying to do on the site but could not.

Consider Known Design Principles

There have been many studies of user interaction with graphical user interface (GUI) applications and websites. From these, many rules-of-thumb have been written to help designers and developers avoid pitfalls [6]. For example, researchers found that people scan rather than read content on web pages [5]. If users want to read something to fully understand it, they will typically print it out. Therefore, web pages that require significant onscreen reading should make use of bulleted lists and plenty of white space to improve on-screen scan-style reading, plus offer a version that is readable when printed. While these guidelines should not be treated as steadfast rules, they should be strongly considered. Once again, usability testing will provide the answers in the long run.

Another human factor to consider in application or site design is that people often make mistakes. With this in mind, designs should expect mistakes, give polite feedback, and provide methods to correct a mistake.

Evaluating Usability During Design and Construction
Several techniques can be used to check usability during the design and construction phases. Task walkthroughs with users ensure all steps are properly accounted for. Next, paper prototypes can be created simply and quickly to demonstrate proposed application interaction to users. Because they are on paper, it is easy to update them on the fly to test several design scenarios. Another advantage of using paper is that they won't look like a final product. This helps keep the focus on content and interaction rather than appearance. Finally, an experienced usability professional can conduct a Heuristic Evaluation, checking the design against known usability principles.

Evaluating the Final Product
Usability testing conducted during the system testing phase provides excellent information about user successes and challenges with a new product. In truth, usability testing will always take place; it is up to the development team to ensure it happens in a controlled environment rather than after product release.

A common vision of usability testing is a formal lab setting. A lone tester works on a computer while being monitored by a video camera and observers behind a two-way mirror. While this is an excellent, somewhat expensive way to learn how well the system works, it is not the only option. Excellent data can be obtained through informally observing as few as five users. According to Nielsen [7], testing five people on the most critical application tasks can uncover 85% of the issues. On the other hand, new research from Spool and Schroeder disagrees [10], saying it would likely take several more testers.

Examples of data that are valuable to collect in a usability test are general opinion of the site, task success, task completion time, and task satisfaction. Analysis of the user's clickstream path, their journey through the application as they complete the task, provides insight to whether users are accomplishing tasks in the way the application was designed to handle them.

Disclaimer: The following companies are discussed to show examples of different approaches to usability testing. Their inclusion does not constitute a recommendation or guarantee that the information described is complete or current.

Market Trends (www.markettrends.com) is a traditional market research firm that offers formal labs and usability test services across the US. Their services can be used for software applications or websites. They produce a series of formal research reports.

Netusability (www.netusability.com) provides a portable software solution to test websites. The setup can be as small as a laptop with a web camera. The test records clickstream data in sync with video of the user. Usability tests can be played back later at different speeds to quickly locate key events.

Vividence (www.vividence.com) provides a tester community of over 150,000 people that use a custom-built browser to perform a scripted usability test in a home or work environment. Clients can choose a tester group to match demographics of their user profiles. Test reports include answers to opinion questions, task success and failure, clickstream data, and comments testers enter while performing the test.

Webcriteria (www.webcriteria.com) performs much of its services with an automated tool that scans a website and predicts user satisfaction based on page load times and navigation theory. The tool is effective in providing a speedy review of large sites, pointing out potential trouble spots for a future detailed evaluation.

WHERE DOES A USABILITY ROLE FIT IN WITH THE DEVELOPMENT TEAM?

Having a few usability evangelists at the worker level is not usually effective in changing the inertia of a development process. A usability focus must come from the top and be communicated strongly throughout the organization.

When a company decides to increase product usability by dedicating resources, Norman [8] believes it is most effective to create a special group focused on usability. The team members would sit in the same area and collaborate as a unit on projects. This arrangement develops group knowledge and skill, while at the same time building a name and reputation within the organization. It is critical for the group to create physical deliverables. Without documentation, usability team members are sometimes perceived as adding little or no value to a product. The goal is to start small and build on continual successes. Once the usability processes are more established, a matrix approach is common. The usability personnel administratively report to a usability manager, but are located with the project teams and are functionally responsible to a project manager.

CONCLUSION

As expectations for business applications and websites rise, it has become more important to follow user-centered design principles to increase product usability. Leaning about the users through field studies provides knowledge of users tasks and goals. Designing the application with known usability guidelines and performing usability reviews during design and construction provide valuable feedback early in the cycle. Usability tests after a product is built measure application effectiveness and can be used in subsequent development phases. There is no doubt that it will take time and effort to include more user-centered methods in the development process, but market forces will continue drive improved usability.

References

1. Beyer, Hugh and Holtzblatt, Karen. Contextual Design: Defining Customer-Centered Systems. Morgan Kaufmann Publishers, Inc., San Francisco, CA. 1998.
2. Cooper, Alan. The Inmates are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity. SAMS Publishing, Indianapolis, IN. 1999.
3. Gomoll, Kate and Bond, Eric. “Discovering User Needs: Field Techniques You Can Use”, Proceedings of User Interface 2000 West (San Francisco, CA, April 2000).
4. Hackos, JoAnn T. and Redish, Janice C. User and Task Analysis for Interface Design. John Wiley & Sons, New York, NY. 1998.
5. Nielsen, Jakob. “How Users Read on the Web”. www.useit.com. October 1, 1997.
6. Nielsen, Jakob. “Ten Usability Heuristics”. www.useit.com.
7. Nielsen, Jakob. “Why you only need to test with 5 users”. www.useit.com. March 19, 2000.
8. Norman, Donald A. The Invisible Computer: Why Good Products Can Fail, the Personal Computer Is So Complex and Information Appliances Are the Solution. MIT Press, Cambridge, MA. October, 1999.
9. Reeves, Byron and Nass, Clifford. The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places. CSLI Publications, New York, NY. 1996.
10. Spool, Jared and Schroeder, Will. “Testing Web Sites: Five Users is Nowhere Near Enough”. Proceedings of CHI 2001, (Seattle, WA, April 2001). ACM Press.
11. Teague, Ross, De Jesus, Katherine, and Nunes-Ueno, Marcos. “Concurrent Vs. Post-Task Usability Test Ratings.” Proceedings of CHI 2001, (Seattle, WA, April 2001). ACM Press. 2001.

Survival of the Riskiest

A pragmatic approach to managing a test effort

By Jennifer Smith-Brock
Ajilon

August 16, 2001

Abstract

This paper outlines an approach to help test managers and test team leads prioritize and communicate strategy and status for the test effort in a pragmatic and useful manner. It also provides insight into approaches for utilizing resources efficiently and effectively. This risk-based approach compliments and encourages teamwork across functional boundaries. It provides insight into how test managers and test leads can communicate effectively the status of the test effort as it relates to risk, therefore provides much needed information as to business risks discovered or possible business risks not yet uncovered. Several papers are published on risk based testing, such James Bach's Heuristic Risk Based testing, and Ingrid Ottevanger's Risk Based Testing. This paper attempts ways to use the information once collected. The content of this paper is from practical experience utilizing parts of the above-mentioned papers to develop this unique approach.

Biographical information

Jennifer Smith-Brock has over 17 years experience in Test management and software testing. She was instrumental in implementing Test Process Improvements in a large development organization. One of the test process improvements she has focused on over her career is Risk Based testing. She has tried many approaches and has found some work better than others. She has also led the Test effort of several high risk, high profile projects to a successful conclusion, exceeding quality expectations and adhering to established schedules. Jennifer was the program chair for Star West 99, Star East 00 and Star West 00. Currently she is consulting in a large development organization implementing new test methodologies, and again is focusing on Risk Based Testing.

RISK:

The possibility of loss or harm; danger

TEST:

A set of questions, problems or tasks used to measure

Introduction

What do you do when you are given a project to manage the testing and you don't have enough resources and time to conduct all the testing you would like to do? Many times we feel compelled to test it all anyway, because we think this is what is expected of us. Now think about what it would be like if you said "I don't have enough time and resources to test everything the way I would like, so let's look at it carefully and collaborate as to the smartest approach".

The message you are sending is multifold, you are sending the message there is more than you can comfortably handle, that you want to work as a team to decide the best approach, and that maybe the way you've always done it may not work.

This can provide many opportunities to improve both the quality and focus of the testing and the perception and relationships of the test organization.

There are many ways to start this process. The one that works for you may not work for others or another organization but there are some items you should address early (very early).

The quality characteristics that are critical to this project or organization are crucial to identify early on.

The project breakdown is also crucial to identify early on, you may choose to break it down by component or by deliverable. The right answer depends on what is right in your organization and for your project.

Once these two elements are identified you are almost ready to start conducting some test risk analysis. By almost, the remaining important task is to learn as much as much as

you can, within the time frame allotted, about the project or program before starting into the risk analysis portion.

Prior to going into the actual analysis, it is important to understand the elements that will be considered.

Quality Characteristics

Quality characteristics are the attributes you and your project team identify as being critical to the success of the project they will differ from project to project but shouldn't differ within a given project. They need to be defined upfront for the entire project so when you are conducting your comparative analysis of the risk you are on an equal playing field. Many of the quality characteristics will apply to most if not all projects such as functionality and testability, there are several that may or may not apply such as localizability and portability. Up until recently most of us have focused our test efforts on functionality, but are now expanding the focus to include additional attributes of a product. The Risk Factors are the aspects that effect the risk for a given quality characteristic. Rather than providing definitions of each one of the elements, considerations are provided which give you a framework to think about.
List of Quality Characteristics and considerations follows:

Figure 1

Quality Characteristic	Consideration
Usability	What is the effectiveness, efficiency and overall satisfaction for the users to achieve their goals?
Functionality	Can it perform the required function? Is it doing what it was designed to do?
Performance	How responsive is the application or system? Are there any bottlenecks with the application or system, which could cause it to slow down or be inefficient?
User Friendliness	How easy or difficult is it to learn to use the application or system?
Reliability	What is the mean time to failure? What is the rate of occurrence of failure?
Installibility	How easy can it be installed and uninstalled?
Compatibility	How well will it work with external components and configurations?
Testability	How much effort will it take to economically and effectively test the application or system?
Maintainability	How economical will it be to build, fix or enhance the system or application? How economical will it be to provide support?
Portability	How economical will it be to port the technology?
Localizability	How economical would it be to release the product in another language or culture?
Credibility	Will the customer have confidence to make decisions based on the system or application results?
Data Integrity	How reliable must the data be in order to be successful?

Risk Factors

These are the facets that introduce risk to a project, release or business process. When the risk factors are inherent in a release or application it is probable the risk will increase accordingly. A highly complex module is more likely to introduce risk to a deliverable than a simple module. List of Risk factors and considerations follow:

Figure 2

Risk Factors	Considerations
Security	Does security introduce risk to the application or system?
New Technology	Are any technologies being introduced to the project team that is new to them?
High Usage	Would high usage from users or high volumes of data introduce risk to the system or application?
Complexity	Does the complexity (I/O, interfaces, calculations, data base updates etc) introduce additional risk to the application or system?
Development Volatility	Is the application or system volatile? In other words are there rapid and unexpected changes occurring in the code or requirements?
Defect History	Historically has there been a high amount of defects in the application or system and what has been the fix rate if there has been a high number?
Cascading Impacts	Is there a possibility of a chain reaction impact with the application or system?

Now, think of a project you are familiar with. Keeping in mind this project do a mental risk analysis using defect history as the risk factor and functionality as the quality characteristic.

What is the probability of defect history introducing risk to the functionality of this project or program?

One answer may be very likely there is always a high amount of defects in the code affecting the functionality and it takes the programmers forever to fix the defects.

Or it could be very unlikely because historically the code has been clean and what few problems there have been have been corrected quickly and with few errors. Each of the risk factors should be assessed against the quality characteristics in this manner.

Figure 3

	Security	New Technology	High Usage	Complexity	Development Volatility	Defect History	Cascading Impacts	*Total Risk Factors*
Functionality	**3**	**2**	**3**	**3**	**3**	**3**	**3**	**20**
Usability	NA	2	2	2	1	2	1	10
Performance	1	1	3	3	3	2	3	16
Installibility	NA	1	NA	2	2	1	2	8
Compatibility	NA	1	NA	2	2	1	1	7
Testability	1	1	2	3	3	3	2	15
Total	5	8	10	15	14	12	12	76

Risk Impacts

Risk Impacts represent the ramifications if there is a failure. By addressing the factors introducing risk to a project we are only looking at the problem in partiality since there could be a difference in the impact of the risk.

For example the likelihood of introducing risk to the functionality is low, but if it just happened to fail what would be the impact? It could be either higher or lower depending on the situation.

There are cases where there are many factors introducing risk to an application and historically many test resources are assigned and tied up for long periods of time, when in actuality a failure isn't catastrophic at all.
The risk analysis provides information that can assist the test manager or test team lead in making difficult decisions such as to the assignments of the testers based on skill level, risk and quality goals. Risk Impacts along with their considerations listed below:

Figure 4

Impact Factors	Considerations
Monetary Ramifications	What would the monetary impact be to either your business or your customer(s) if there were a failure for the system or application?
Regulatory/compliance/negative publicity	What would be the cost to the business in a) loss of existing business b) loss of potential business c) liability?
Other Internal systems impact	What would be the economical impact of this type of failure?
Missed Market opportunity	What would be the ramifications if you missed the market by either delivering late or missing the market needs?

Here is a sample of the Impact factors assessed against the Quality Characteristics:

Quality Characteristics	-								Impact Factors					
	Security	New Technology	High Usage	Complexity	Development Volatility	Defect History	Cascading Impacts	*Total Risk Factors*	Monetary Ramifications	Compliance/Negative Publicity	Other Internal systems impacted	Missed Market Opportunity (Deadlines)	*Total Impact*	*Total Risk & Impact*
Functionality	**3**	**2**	**3**	**3**	**3**	**3**	**3**	**20**	**3**	**3**	**3**	**3**	**12**	**32**
Usability	NA	2	2	2	1	2	1	10	1	1	NA	NA	2	12
Performance	1	1	3	3	3	2	3	16	2	3	2	3	10	26
Installibility	NA	1	NA	2	2	1	2	8	2	2	2	2	8	16
Compatibility	NA	1	NA	2	2	1	1	7	1	2	2	1	6	13
Testability	1	1	2	3	3	3	2	15	2	2	3	1	8	23
Total	2	6	7	12	11	9	9	56	8	10	9	7	34	90

Methods of documenting the assessed value

There are several methods of documenting and communicating the analysis. This approach doesn't prescribe any particular method. It depends on the situation and culture. One technique is the use of numeric data. Numeric data can mislead people to think it is scientific when in fact it is relative and no more scientific than stating the risk as high medium or low. With all that said the starting place can be normal risk and the risk is either higher or lower. This is the method James Bach uses in his heuristic risk based testing. There are many very good reasons for using this approach. Another approach is to assign numeric values such as 1, 2, 3 or 0.

- 0 meaning it is not applicable, there is no possibility at all the risk factor will introduce risk to a quality characteristic.
- A value of 1 may indicate there is a possibility of risk being introduced but is unlikely.
- A value of 2 may mean there is a possibility of introducing risk and it is likely to occur.
- A value of 3 could mean the likelihood is very high and probably will happen.

This is the scheme I have used on projects. You may notice both schemes are simple. Whatever scheme you prefer, remember that simplicity is critical to the practical use of the analysis. The analysis is a tool to assist with the planning and managing of the test effort and shouldn't consume the project.

The Analysis

The quality characteristics are determined for the project, and the project has been broken down into a logical structure for the team, the testers have learned as much as they can about the project prior to this assessment. You are now ready to assess the risk. This is a significant step and it is essential to have all the stakeholders involved in this aspect. Developers, Users and Testers are key participants and there are often several others who are key to participate in these sessions.

It is crucial for everyone involved to understand why you are doing this and how the results will help you. It is better this is explained up front otherwise some may not focus during the session, because they are so busy trying to figure out why and how this will be used later.

Review the project scope with the group prior to assessing the risk; some interesting information can be derived from this part of the session. Then begin going through each characteristic and ranking the risk factors and impact factors for each one. This exercise can be surprising especially when the developers are ranking items much higher than you would. Be open and ask lots of questions as to why someone ranks it differently than you would, a lot can be learned from this. It is valuable to reiterate how this information will be used upon conclusion of the session as well and ask if participant's want follow up information.

Using the information

The sessions are complete and you have a worksheet completed as a result of the discussion with the key players for the project. It is time to put this information to use, please note that many times the sessions themselves have proven to have incredible value for the team in its entirety. Most test managers assign resources, manage the focus of the test effort and schedule the test effort, and report status, just to name a few responsibilities.

As a result of the assessment a deliverable might have high risk factors and high impact; this information may influence a decision to assign experienced tester(s) to this area. The scheduling may be adjusted to provide additional time for this deliverable, this can be done in a couple of ways:

1) Influence the developers to develop in this area earlier and therefore turn it over sooner
2) Reduce the effort of another deliverable with considerable less risk.
3) Provide testing support to unit test.

On the contrary a deliverable that has low risk factors and low impacts, inspections may cover all the testing necessary for this particular deliverable. These types of deliverables can also provide an opportunity to get a new tester, or less experienced tester started with minimal risk to the project.

It gets trickier with the mixed bag so to speak, high in risk factors but low in impacts. This type probably requires more effort than an inspection, but less vigorous an effort than the high risk factors and high-risk impacts. You may want to assign some experienced testers, especially to the high impact deliverables, but may want to mix the expertise in order to gain some additional strength for future projects. Below is a sample of a chart used in communicating out the strategy the test organization took on a release assessed using this approach.

Figure 5

Release/Component	Skill Set	Effort	Unit test	Early delivery
Sample A	Expert	Moderate	Yes	Attempt
Sample B	Newbie	Low	No	No
Sample C	Moderate	High	Yes	Yes

It is recommended to report status based on risk rather than number of tests executed passed etc. By reporting on the number of passes and fails provides limited information at best and can be very misleading. When the status reporting is based on the risk

analysis it provides a much clearer picture of the true status. For example 60 tests have been executed out of 100 possible, of those 60 40 passed and 20 failed. At first glance this tells us 20% failed, 40% passed and we are 60% complete. From this information are you comfortable making a decision whether to ship tomorrow?

Now lets take the same 100 tests and break them down by risk. Of the 100 tests 40 cover the high-risk areas, 40 cover the moderate risk and 20 cover the low risk. The 40 tests that have passed are the tests that cover the high risk areas, so in effect you have tested 100% of the high risk tests, the 20 that failed are in the moderate risk area and the 40 that haven't been executed yet are the remaining 20 moderate and the 20 low. Now ask yourself am I more comfortable making a ship/don't ship decision. The tool is available to look further and decipher what types of problems you may encounter if shipped without further testing; this provides information valuable when making these types of decisions.

Another approach is to conduct a comparative analysis from deliverable to deliverable. This would entail holding two up and asking the team which is riskier? From what perspective are they riskier than the other? This is a valuable tool to use when up against a situation where your demand for expertise in one area is higher than your supply (or testers). This has happened to me, how about you?

As with most good things there are some unexpected benefits as well:

- Don't be surprised if the project is reevaluated if there are several areas with high risk and high impacts.
- It can be a tool for communicating with the project team. Only with this approach they are providing input and helping you with your planning, this fosters teamwork.
- Provides a clear, focused approach to the test effort and the assignments.
- The framework can be used as a mental checklist when in a position to make a "hasty" decision.
- Provides a tool for negotiating time, resources and functionality on a project or deliverable. It is easier to communicate the importance of the test effort in particular areas.
- Supports an iterative approach and should be revisited often to understand where the risks have changed and shifted.

Summary

This approach can be an effective tool for planning and reporting status for the test effort. Even though the advantages are far greater when used through out the development cycle there are definite advantages to begin using it later as well. It can be used as a tool to help prioritize and influence the effort. It can also help identify areas that should have more test focus than others, or on the other hand sometimes you might find an area that doesn't require as much focus since the risk impacts are minimal. It provides information that managers can use to utilize their resources wisely such as minimizing risk by assigning testers with higher expertise on higher risk assignments and assigning newbies (newcomers to the field) to the low risk areas. This also helps the team morale, since the experienced testers are recognized for their expertise and the newbies aren't assigned to difficult if not impossible tasks.

The next time you are asked to manage a test effort that you don't have enough time and resources to test as you would like, try this approach. Collaborate with your peers and determine which components or releases or projects are the riskiest and let them rise to the top of the priorities, thus, the Survival of the Riskiest.

How to collect more reliable defect reports

Abstract

Resolving defects in software effectively can be expensive. Lack of reliable defect reports is often the cause. Sometimes, the parties involved spend more time analyzing poor defect reports than resolving defects. Collecting reliable reports is challenging.

Testing defect reports for five characteristics and emphasizing seven process improvement tasks produces more reliable defect reports. Reliable defect reports tend to increase effectiveness of resolutions; quick and effective resolutions increase customer satisfaction and reduce maintenance cost.

This paper defines five characteristics of reliable defect reports. It emphasizes seven process improvement tasks to collect more reliable defect reports. It provides examples of how to use simple tools to improve the characteristics of defect reports and the process.

Keywords: Defect Management, Anomaly Management; Quality Assurance; SQA KPA of CMM; IEEE 1044; Process Improvement.

Target audience: Managers of Product Development, SEPG, and Customer Support functions.

Status of paper: Final

Author: Sudarshan “Sun” Murthy, Vice President, Sunlet Software Systems, Inc. Portland, OR 97205 USA. URL: http://www.sunlet.net/. mailto:SMurthy@sunlet.net

Author biography: Sun Murthy is employed at Sunlet Software Systems, Inc., a company he founded in 1998. Sunlet defines software engineering processes and develops tools to implement those processes. Sun practices, coaches, and learns software engineering with other enthusiastic engineers at Sunlet. He also assists engineers and managers of software engineering groups in the Pacific Northwest area in developing quality software. Before founding Sunlet, Sun was employed at Tiger Systems, Inc. in Portland, OR USA. At Tiger, Sun pioneered and led a successful software engineering team in building international trade finance applications.

Introduction

Users of commercial software want better quality. While users appreciate improvements in installation and integration procedures of new software, they are unhappy with upgrades and updates. The unhappiness often stems from defects (new or reappearing) in the upgraded software. Vendors have improved their defect management process significantly in recent years to satisfy users in this regard. They invest in increasingly sophisticated defect tracking systems to record and prioritize defects. Yet, vendors are not always able to provide effective solutions quickly. They are unable to keep maintenance costs as low as they would like. Cem Kaner et al. [1] explain the challenges in more detail.

Resolving defects in software effectively requires reliable defect reports. Defect reports often contain insufficient or incorrect information. Defects are sometimes recorded by someone other than the observers. Defect reports describe symptoms, not causes. Recording defects is time consuming. Defect reports don't reach vendors in time; sometimes they just don't reach the vendor at all.

Vendors have improved defect report designs significantly in order improve the reliability of reports collected. They have created and or adopted standards to classify defects. IEEE Standard 1044 [3] provides a uniform approach to classifying defects found in software. It describes how defects can be recorded and processed during the life of software. IEEE Standard 1044.1 [4] is a guide to applying IEEE standard 1044. The guide enables vendors to implement and customize the standard in an effective and efficient manner. IEEE Standard 12207.2 [5] guides vendors in implementing software. It explains the records a vendor should maintain to ensure successful implementation and maintenance of software. There is still much room for improvement in reliability, despite applying standards.

We propose process-based solutions to improve reliability of defect reports based on our observations and experience in software engineering. We describe five characteristics of reliable defect reports namely: clarity, correctness, conciseness, completeness, and consistency. We emphasize seven tasks to improve these characteristics and the software engineering processes (more specifically, software quality assurance key process area [2]). For each proposed improvement in the process we call out the characteristics of defect reports improved, the software life cycle phases in focus, and the roles in focus. The tasks emphasized are (in roughly the order vendors may approach improvement):

1. Collect information about the operational environment of products.
2. Preserve users' perspective in defect reports throughout the process.
3. Get users to record defects as soon as they observe defects.
4. Use recording tools that seamlessly integrate into products.
5. Exploit error-handling facilities in programming languages.
6. Integrate developers' and users' recording tools.
7. Reduce work for everyone involved (a result of other improvements proposed and an improvement goal by itself).

We provide tools to improve specific process activities involved. We explain the need for the proposed improvements with examples. Although our examples are described in the context of applications for Microsoft (MS) Windows, we expect the ideas to be applicable to other platforms as well.

The proposed improvements can result in the following:

- More reliable defect reports.
- Reduced dependence on users.
- Reduced cost of maintenance.

- More effective defect resolutions.
- Faster turnaround on defect resolutions.
- More maintainable products.

Here is a glossary of the terms we use extensively:

Life cycle:	Set of activities involved in life of a product. This ranges from conception of product to its retirement.
Life cycle phase:	A convenient grouping of related life cycle activities.
Roles:	Abstraction of set of responsibilities in life cycle activities. People generally play roles; a person may play many roles.
Users:	People consuming products to achieve business or personal goals.
Observers:	People who observe defects in products. These are mostly users for products in maintenance phase. We use the terms users and observers interchangeably.
Vendors:	Organizations or people creating and distributing products to users.
Defects:	Failures in behavior of products. IEEE Standard 1044 [3] uses the term “anomalies.”

Note: IEEE Standard 610.12 [6] defines a complete glossary of software engineering terminology. At times, we use more common software engineering terms (instead of using terms in the standard) in the hope we can reach more practitioners.

Characteristics of reliable defect reports.

Defect reports often contain insufficient or incorrect information. Poorly designed reports, or lack of designed reports are some of the causes. Users have first hand knowledge of defects since they observed the defects. Supplying well-designed reports to users enables them to record the first hand knowledge they possess. Educating them in recording the defects they observer helps produce reliable reports.

Reliable defect reports have the following characteristics:

- Describe defects unambiguously (clear).
- Describe defects accurately (correct).
- Describe defects crisply and focus on one defect at a time (concise).
- Contain all information necessary to reproduce defects (complete).
- Describe defects using the same terminology and same set of attributes each time (consistent).

Software engineering processes can be improved with these characteristics in mind. We first present an example of improvement possible. We then present a series of improvements to software engineering processes by emphasizing certain tasks. We propose these improvements with empirical support, but not statistical support.

Example: Mark User uses Doodad Pro 2.5 to operate his business. One day the product just froze soon after he started using it. Mark immediately sent a one-line mail to the vendor. The mail read: “Product is frozen cold.” Maintainers of the product sent Mark a defect report template (Microsoft Word document template) based on IEEE Standard 1044 [3]. They also trained Mark to report defects using the template. Appendix A shows the defect report Mark sent using the template. The clearer description of the problem helped the maintainers and Mark understand that the product was not frozen! Instead, they understood it was too slow!

Emphasis 1: Collect information about the operational environment of products.

Information about environment reduces cost of resolution and enables faster turnaround. Operational environment heavily influences product behavior. Version of operating system, amount of hard disk, and amount of memory are common environmental factors. Some defects manifest only when another product is installed on the same machine, some manifest on networked machines, some manifest on faster processors, and so on. Some defects even show up only when a particular user logs in. It is common for vendors to interact with users many times to obtain this information. A defect report without information about environment is unclear and incomplete.

Listing environmental factors of products is challenging. The list depends mainly on operating systems and hardware architectures. IEEE Standard 1044 [3] classifies environment factors as optional supporting data items. Collecting some basic environmental factors such as processor type, operating system version, memory available, disk space available, and system-level user preferences, as mandate, helps the process.

Characteristics improved: Clarity and Completeness

Life cycle phases in focus: Maintenance.

Roles in focus: Users and Supporters.

Example: Maintainers of Doodad Pro 2.5 were unable to reproduce the problem Mark User reported. However, Mark could reproduce the problem every time. The maintainers asked Mark to

send them details of the operational environment. Appendix A shows some of the software environmental factors the maintainers collected. From the information about hardware in the defect report (not shown in Appendix A due to space constraints), the maintainers noticed that Mark uses a Token-ring network, whereas the vendor used an Ethernet network. Maintainers managed to setup a Token-ring network and reproduced the problem immediately.

Emphasis 2: Preserve users' perspective of defects throughout the process.

Preserving users' perspective of defects throughout the defect management process increases effectiveness of resolution. The following attributes defined in IEEE Standard 1044.1 [4] are very important since they determine urgency of defect resolution:

- Priority: importance users attach to having defects resolved
- Severity: degree of effect on usability of product

Users and vendors may not always assess the same values for these attributes. Users assign priority and severity based on their needs. Vendors assign value to the same attributes based on risks, rewards and resource availability. Vendors also tend to change values of these attributes for a defect more than once in the defect resolution process. For example, a vendor may assess "high" priority to a defect when the report is received. The vendor may downgrade priority to "medium" after investigation. The change may be required because other defects need urgent resolution or there is a shortage of resources to resolve defects.

Preserving users' perspective reminds users' needs to vendors throughout the process. Vendors may lose sight of users' needs if they do not preserve users' perspective. Defect resolutions may become 'too little, too late.' Surprisingly, vendors managing defect reports using electronic databases are more susceptible to this mistake. Surprisingly again, some defect databases contain just one set of fields to describe a defect. Preserving a paper copy of users' perspective, or designing defect database to use different set of attributes to store users' perspective alleviates such mishaps.

Characteristics improved: Clarity.

Life cycle phases in focus: Maintenance.

Roles in focus: Users and Support Manager.

Example: Mark User came across a defect in Doodad Pro 2.5 that affected his business rather seriously. He assessed "High" priority to the defect and reported it to the vendor. However, the vendor assessed "Low" priority to the same defect due to resource shortage. When resource was available later, the vendor reassessed priority to "Medium" (other defects needed more immediate resolution). The vendor did not consider Mark's needs since Mark's priority assessment was not preserved. The vendor later discovered that Mark was considering a competing product since he was unhappy that the defect wasn't resolved quickly. The vendor understood the reason for Mark's disappointment and had the defect database redesigned (to store users' perspective separately).

Emphasis 3: Get users to record defects as soon as they observe defects.

Defect reports recorded by users who observe defects are more reliable. Sometimes users describe defects over telephone (or in a mail), and a vendor's representative records. Critical information may be lost in this process. Limitations of communication media may limit the type of information users provide. For example, a snapshot of the screen cannot be described accurately in a telephone conversation. Users who observe defects have first hand knowledge of defects. The defect records are clearer if the users record them.

Defect reports recorded soon after defects are observed are more reliable. Users may delay reporting defects if they can't readily access customer support phone number. Users may be forced to delay reporting if they have to wait a long time for a support representative to become available. Over time, users may forget details of defects they observed. Unreliable reports recorded later may send the team investigating defects on a wild goose chase! Simple tools supplied with products enable and encourage users to record defects early. The tools may be paper-based or electronic. However, the tools must be easy to use and readily accessible, regardless of sophistication.

Characteristics improved: Correctness, Conciseness, and Completeness.

Life cycle phases in focus: Maintenance and Design.

Roles in focus: Installers and Designers.

Example: Designers of Doodad Pro 2.5 altered their installation process to create a shortcut to MS Word document template, along with shortcut to the product. Appendix B provides the Visual Basic procedure they used to create the shortcut.

Emphasis 4: Use recording tools that seamlessly integrate into products.

Recording tools integrated into products are more readily accessible. Users sometimes do not record defects (immediately) because recording tools are not readily accessible. Tools integrated into products encourage users to record defects early. Additionally, depending on sophistication, these tools can collect some information about operational environment of the product. They can also allow users to transmit reports (immediately) to vendors. Such integration reduces the number of interactions required with users. It reduces cost of resolution and enables faster turnaround. Recording tools integrated into products may be as simple as document templates or they may be sophisticated programs.

Integrating recording tools into products has some caveats. The tools may become inaccessible when users need it most. This situation may arise if the host product is the only way to launch the tool, and the host product has a defect that prevents it from starting. Providing an independent access path to the tools alleviates this problem. Integrating recording tools into products may cause defects in host products. Selecting an integration technology that allows seamless integration, yet keeps the tool separate enough from host products is a key to success. Candidate solutions include component technologies such as Component Object Model (COM), Common Object Request Broker Architecture (CORBA), and Java Beans.

Characteristics improved: Clarity, Correctness, Conciseness, Completeness, and Consistency.

Life cycle phases in focus: Development, and Design.

Roles in focus: Developers and Designers.

Example: Developers of Doodad pro 2.5 added an item to the Help menu that allows users to record defects. Users now select this menu item within the product to access the recording tool supplied with the product. Appendix B shows the Visual Basic procedure the developers used to create new defect reports from a document template.

Emphasis 5: Exploit error-handling facilities in programming languages.

Defect reports that describe causes in addition to symptoms help investigation. Users judge defects based on symptoms of problems they see. They generally cannot describe causes.

However, the causes are more visible to developers of products. Recording causes using error-handling capabilities in programming languages helps investigation.

Recording errors in code helps analyze causes and effects. Code is structured as a set of interacting procedures designed to meet users' requirements. Each procedure has a clear purpose. The user triggers 'top-level' procedures with user interface elements. Top-level procedures are typically designed to provide broad services in limited contexts. Top-level procedures call 'lower-level' procedures. Lower level procedures are typically designed to provide specific services applicable in multiple contexts. These procedures may call other lower level procedures. This arrangement of code causes a 'call stack' or a 'call sequence' when programs run. The call sequence references the procedure currently executing, preceded by the procedure that called the current procedure, and so on up to a top-level procedure.

Recording errors in procedures, in same sequence as procedure call sequence, establishes cause and effect relationship among errors. Developers handle exceptional situations in procedures using error-handlers. A procedure may fail to meet its purpose in case of errors. This failure means the preceding procedure in call sequence may fail to meet its purpose. This cause and effect sequence propagates through the sequence up to the top-level procedure that initiated the sequence. Recording effects of failures of each procedure in sequence provides excellent insight into the situation during investigation.

The emphasis here is to record only errors, so nominal execution of call sequence does not create any record. Ben Ezzell and Ron Petrusha [7] explain building error messages in detail.

Characteristics improved: Clarity, Correctness, Conciseness, Completeness, and Consistency.

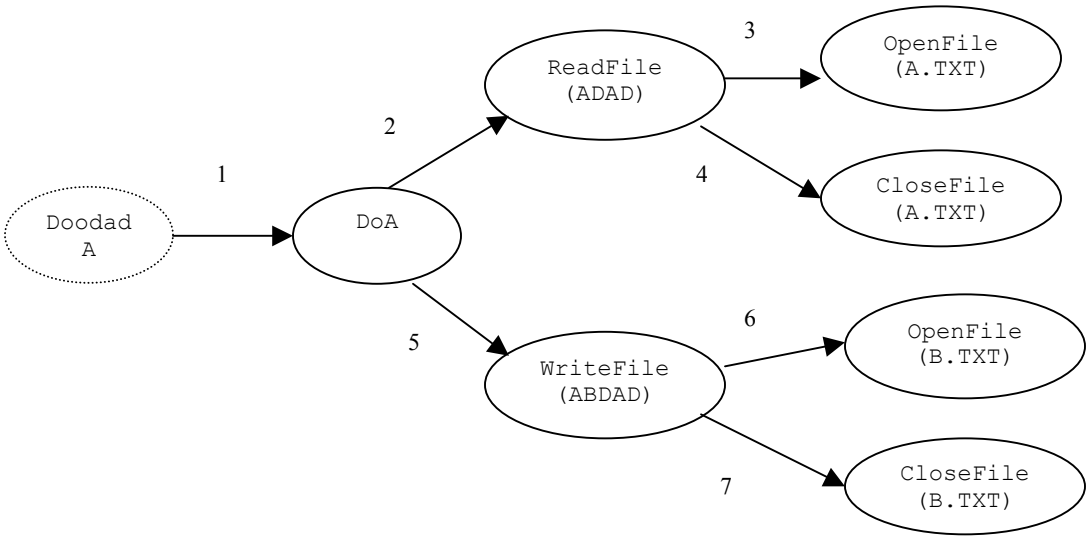
Life cycle phases in focus: Development and Design.

Roles in focus: Developers and Designers.

Example: Doodad Pro 2.5 provides several doodads to users. Users just push a button to start each doodad. Selecting Doodad 'A' calls the top-level procedure `DoA`. Procedure `DoA` reads the file `'ADAD'`, scrambles its contents, and creates the file `'ABDAD'`. Procedure `DoA` uses procedure `ReadFile` to read the file `'ADAD'`; it uses procedure `WriteFile` to write the file `'ABDAD'`. Procedures `ReadFile` and `WriteFile` both use procedure `OpenFile` to open a file and procedure `CloseFile` to close a file.

Figure 1 shows the nominal call sequence executed when users select Doodad 'A'. That is, this call sequence is executed completely only if all seven steps are executed successfully. The dotted bubble denotes the user interface element to start Doodad 'A'. The bold circles denote procedures called in response. The text in a bold bubble denotes name of the procedure. The text in parentheses denotes the parameter value passed to the procedure. The number on the edge between bubbles denotes the sequence of procedure calls.

Figure 1: Nominal call sequence to execute Doodad A.



The nominal call sequence is:

1. Procedure `DoA` is called first without any parameter.
2. Procedure `DoA` then calls procedure `ReadFile` with `'ADAD'` as parameter value.
3. Procedure `ReadFile` calls procedure `OpenFile` with parameter value `'ADAD'`.
4. Procedure `ReadFile` then calls procedure `CloseFile` with parameter value `'ADAD'`. The call sequence then returns to procedure `DoA`.
5. Procedure `DoA` calls procedure `WriteFile` with parameter value `'ABDAD'`.
6. Procedure `WriteFile` calls procedure `OpenFile` with parameter value `'ABDAD'`.
7. Procedure `WriteFile` then calls procedure `CloseFile` with parameter value `'ABDAD'`. The call sequence then returns to procedure `DoA`. Doodad 'A' has executed successfully.

Errors could occur in executing this call sequence. In case of any error, say opening file `'ADAD'` because the file does not exist, the sequence stops after step 3. The error handling sequence in that case is:

1. Procedure `OpenFile` handles file system error 'File not found' and records it. It raises custom error `'File ADAD. not found.'`.
2. Procedure `ReadFile` records the propagated error and raises custom error `'Unable to read file because the specified file ADAD does not exist.'`
3. Procedure `DoA` records the propagated error and informs the user that it could not start Doodad A. It displays a message that reads `'Doodad A could not be started because the input file ADAD was not available. Please make sure the file is available and try again.'`

Emphasis 6: Integrate developers' and users' recording tools.

Using a single tool with which both developers and users record their perspectives produces more reliable defect reports. While recording defects soon after users observe them, and recording errors in procedures both improve reliability, information from these records are generally disconnected. These records are connected if we view users' perspective as the final

effect in the cause and effect chain (recorded by developers). This extension connects errors developers handle with defects users observe. It produces reports that explain defects from both business perspective and implementation perspective.

Integrating developers' and users' recording tools requires sophisticated technologies and effective software engineering processes. An ideal integrated tool will be available system-wide so users can record defects using the same tool, regardless of the product where defects are observed. Developers will use the same tool to record the cause and effect relationship of errors as they occur. The tool will be standards based. It will allow customization, based on each vendor's needs. The tool will gather relevant information about the operational environment of the product. The tool will seamlessly integrate into applications, but not interfere with the functioning of host applications. Users will access the tool without needing any other software (in case the user needs to report a defect observed in products that are not accessible anymore). Object technologies such as COM, CORBA, and Java Beans are candidate technologies.

Characteristics improved: Clarity, Correctness, Conciseness, Completeness, and Consistency.

Life cycle phases in focus: Development and Design.

Roles in focus: Developers and Designers.

Example: Mark User starts Doodad 'A'. He sees a message box that reads `'Doodad A could not be started because the input file ADAD was not available. Please make sure the file is available and try again.'` Mark ensures the file exists and retries the operation. However, Doodad 'A' fails again and Mark sees the same message. Figure 2 shows the message box Mark sees.

Figure 2: Message box displayed to the user of Doodad Pro indicating failure to start Doodad A.

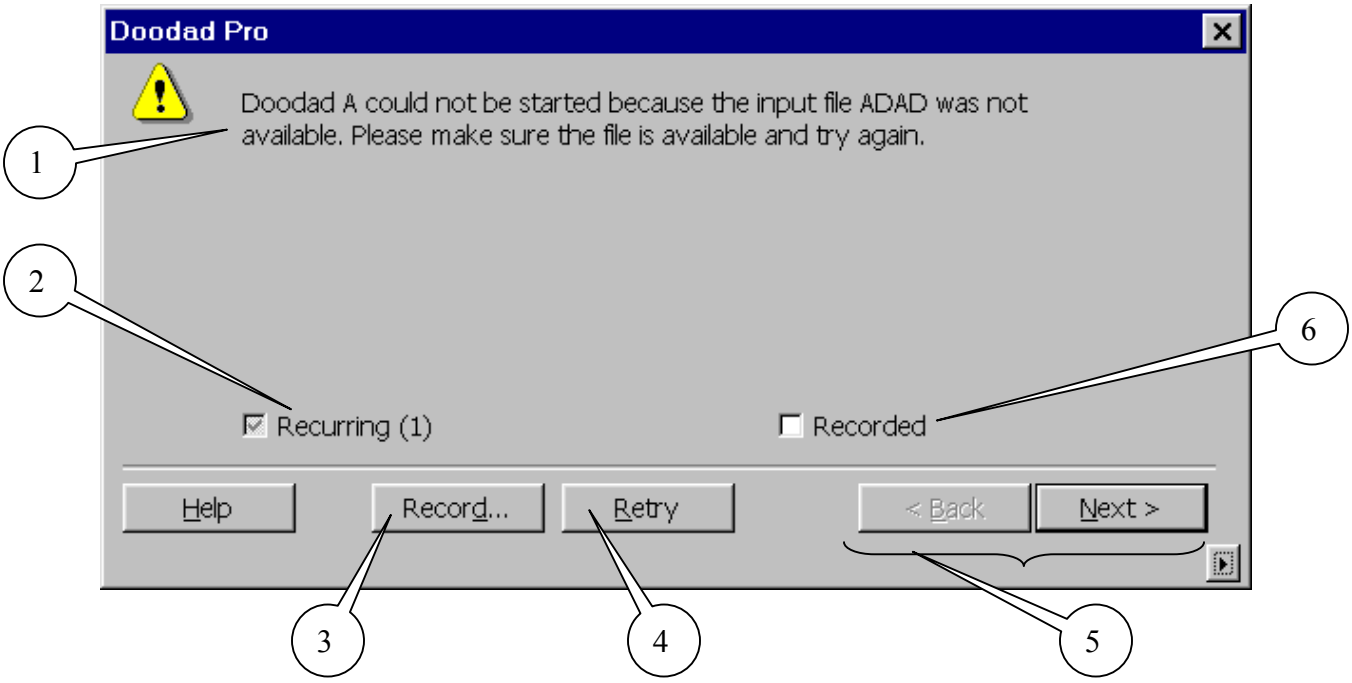


Figure 2 calls out the following components in the message box:

1. The effect of the error in terms of user function (also the defect users observe). The message informs the user that Doodad A did not start and the cause. This is better than just saying a file was not found.
2. A flag that indicates a recurring defect. The display also includes the number of times the error has recurred.
3. A button that allows users to record the defect. A recording window pops up when the user selects this button. The user records her perspective in the recording window.
4. A button that allows users to retry the operation. Selecting this button initiates the call sequence again.
5. Buttons that allow users to navigate through the series of errors that caused the defect. Users study the cause and effect chain using these buttons.
6. A flag that indicates if the defect has been recorded.

Mark records this defect, marks it urgent, and sends the report to the vendor. The report contains a clear, correct, concise, complete, and consistent report of the defect. The maintainers easily track the cause to procedure `OpenFile`. It turns out `OpenFile` appends a period to file names that do not have an extension!

Emphasis 7: Reduce work for everyone involved.

Reducing work encourages people to complete their responsibilities. Although emphasizing the tasks we have discussed so far reduce work for everyone involved, emphasizing reduction of work specifically may improve the process some more. In our experience, software engineering process implementations (or improvement efforts) often fail because processes cause more work. People find it easier to dodge such processes than to adhere. Parties to defect management process are no exception.

Processes that account for human factors build tools that automate data gathering and processing. Integrating recording tools and automating data gathering reduces work for users. Defect reports that contain both developers' and users' record reduce work for maintainers. The reduced work encourages them to proactively participate in process improvement. Work can be reduced further for developers by providing them with template code to handle errors and developing libraries reduces work for developers.

Characteristics improved: Clear, Correct, Concise, Complete, and Consistent.

Life cycle phases in focus: Development, Design and Process Management.

Roles in focus: Developers, Designers and Process Manager.

Example: Engineers of "Doodads 'r Us", makers of Doodad Pro, developed "add-ins" for developers. Developers use the add-ins in an Integrated Development Environment (IDE) to easily add error-handling code to procedures. Designers of Doodad Pro also enhanced the defect-recording tool to pre-fill information such as user name, company name, and other environmental information in defect reports. They found this easy to do since much of the information was already available in the system registry.

Summary

Collecting reliable defect reports is challenging. Using simple tools and emphasizing seven tasks in the engineering processes produces more reliable defect reports. Table 1 summarizes the tasks emphasizes and the characteristics improved. The 'Emphasis' column lists the tasks

emphasized for improvement. An 'X' mark in the 'Defect Report Characteristics' columns indicate the characteristic improved due to the emphasis.

Table 1: Process improvement task emphasized and defect report characteristics improved.

	Emphasis	Defect Report Characteristics				
		Clear	Correct	Concise	Complete	Consistent
1.	Collect information about the operational environment of products.	**X**			**X**	
2.	Preserve users' perspective of defects throughout the process.	**X**				
3.	Get users to record defects as soon as they observe defects.		**X**	**X**	**X**	
4.	Use recording tools that seamlessly integrate into products.	**X**	**X**	**X**	**X**	**X**
5.	Exploit error-handling facilities in programming languages.	**X**	**X**	**X**	**X**	**X**
6.	Integrate developers' and users' recording tools.	**X**	**X**	**X**	**X**	**X**
7.	Reduce work for everyone involved.	**X**	**X**	**X**	**X**	**X**

The benefits of making the proposed improvements are:

1. More reliable defect reports due to improvement in five reliability characteristics of defect reports.
2. Reduced dependence on users since users record defects early and report them early. Tools collect much of the information that users otherwise supply.
3. Reduced cost of maintenance due to reduced number of interactions with users; reduced investigation effort since report includes causal information of defects in addition to symptoms.
4. More effective defect resolutions.
5. Faster turnaround on defect resolutions since defect reports contain information that helps investigation.
6. More maintainable products since comprehensive defect reporting capability is built into products by design.

Acknowledgements

My family, for allowing me to indulge in writing this paper.
All reviewers, for their patient reviews of this paper.
Professors Tim Sheard and Todd Leen at Oregon Graduate Institute, for teaching scholarship skills.
My employer Sunlet Software Systems, Inc., for sponsoring this paper.

Copyrights and trademarks

References

1. Cem Kaner, Hung Quoc Nguyen, Jack Falk.
Testing Computer Software, 2nd Edition.
John Wiley & Sons. April 1999.

2. Mark C. Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis,
The Capability Maturity Model: Guidelines for Improving the Software Process.
Addison-Wesley Publishing Company, Reading, MA, 1995.

3. The Institution of Electrical and Electronics Engineers, Inc.
IEEE Standard Classification for Software Anomalies (IEEE Standard 1044).
IEEE 345 East 47th Street New York, NY 10017-2394. 1993.

4. The Institution of Electrical and Electronics Engineers, Inc.
IEEE Guide to Classification for Software Anomalies (IEEE Standard 1044.1).
IEEE 345 East 47th Street New York, NY 10017-2394. 1995.

5. The Institution of Electrical and Electronics Engineers, Inc.
Guide for information technology - Software Life Cycle Processes - Implementation considerations (IEEE Standard 12207.2).
IEEE 345 East 47th Street New York, NY 10017-2394. 1997.

6. The Institution of Electrical and Electronics Engineers, Inc.
Standard Glossary of Software Engineering Technology (IEEE Standard 610.12).
IEEE 345 East 47th Street New York, NY 10017-2394. 1990.

7. Ben Ezzell, Ron Petrusha (Editor)
Developing Windows Error Messages.
O'Reilly & Associates, Inc. March 1998.

Appendix A: Sample defect report design and environmental factors

Figure 3 shows part of a defect report filled using a MS Word document template. The template groups attributes of defects cohesively.

Figure 3: Part of a defect report filled using a MS Word document template.

Identification

1. Report number:	34		
2. Product name:	Doodad Pro	2a. Product version:	2.5
3. Registered user name:	Mark User	3a. Registered company name:	Surf Corp.

Observations

4. Date defect recorded:	3/14/2001	5. Date defect observed:	3/14/2001
6. Status of the product:	Degraded		

7. Brief description of defect:
The doodads take more than 20 minutes to produce results after the software has been running for about 10 minutes.

Categorization

8. Symptoms:	Slow performance	9. Nature:	Wrong
10. Repeatability:	One time occurrence	11. Severity:	High
12. Priority of resolution:	High	13. Value of resolution:	High

Hardware environment

Some hardware environmental factors pertinent to MS Windows applications are:

- Computer manufacturer name and model
- Computer name
- Type of processor and speed
- Total and available physical memory
- Total and available hard disk space
- Total and available virtual memory
- Type of network (Ethernet, Token-ring, etc.)

Some software environmental factors pertinent to MS Windows applications are:

- Operating system name and version.
- Computer user name
- Installed internet browsers and version
- Installed Office suites and version (MS Office, Lotus Notes, etc.)
- Type of network (Windows, Novell, etc.)
- Other products installed
- User preferences

Appendix B: Sample Visual Basic code

The sample code is created using Microsoft Visual Basic 6.0. The code is designed to work in Visual Basic (VB) version 4.0 and later, VB for Applications (VBA) embedded in Microsoft Office applications such as Word, Excel and Microsoft Access. The code is designed to run on Microsoft Windows 9x, Windows NT, Windows 2000, and Windows ME.

A. Creating shortcut to allow users to record defects

Listing A shows VB code to create shortcut to a defect report template. The code assumes the MS Word defect report template file is called '`Defect Report.dot`' and that the template file is stored in the folder '`C:\WINDOWS\Application Data\Microsoft\Templates\`' (this is the typical location for MS Word templates). The procedure creates a new shortcut in the current working folder. Change the procedure to your requirements. You need to reference the library 'Windows Scripting Host Object Model' in your project. Procedure `CreateShortcut` may be called from anywhere. For example, you may include it in your installation program.

Listing A: Sample Visual Basic code to create a shortcut to defect report template.

```
' Add reference to library 'Windows Scripting Host Object Model'
Public Sub CreateShortCut()
   Dim oShell As IWshShell_Class
   Dim oShellLink As IWshShortcut_Class

   ' Gain access to shell programming interface
   Set oShell = CreateObject("WScript.Shell")
   ' Alter the parameter to use a path to appropriate folder.
   ' The shortcut is created in the current folder by default.
   Set oShellLink = oShell.CreateShortCut("Record a defect.lnk")

   ' Set the properties of the new shortcut and save it.
   ' Alter the target path if the template file uses a different path
   ' Alter the icon to anything you like, remove the line completely
   ' to use default icon
   With oShellLink
      .TargetPath = _
       "C:\WINDOWS\Application Data\Microsoft\Templates\Defect Report.dot"
      .IconLocation = "Resourcefile, resourceindex"
      oShellLink.Save
   End With
   Set oShellLink = Nothing
   Set oShell = Nothing

End Sub
```

B. Creating a new defect report from a template

Listing B shows VB code to create a new defect report from an MS Word defect report template. The code assumes the template file is called '`Defect Report.dot`' and that the template file is stored in the folder '`C:\WINDOWS\Application Data\Microsoft\Templates\`' (this is the typical location for MS Word templates). The procedure executes the '`New`' command on this template to automatically load MS Word and create a new document based on the template. Procedure `ReportDefect` may be called from anywhere. For example, you may call this procedure from the Click event procedure of a menu item.

Listing B: Sample Visual Basic code to create a new defect report from a template.

```
' Place this declaration at module level
Private Declare Function ShellExecute Lib "shell32.dll" _
       Alias "ShellExecuteA" _
       (ByVal hwnd As Long, ByVal lpOperation As String, _
       ByVal lpFile As String, ByVal lpParameters As Long, _
       ByVal lpDirectory As String, ByVal nShowCmd As Long) As Long

' Call this procedure from anywhere you want to create
' Procedure can cause an exception if it fails. Exception number is
' between vbObjectError+1 and vbObjectError+31 (both inclusive)
Public Sub ReportDefect()
   Dim nResult As Long

   nResult = _
       ShellExecute(0&, "New", _
       "C:\WINDOWS\Application Data\Microsoft\Templates\Defect Report.dot", _
       0&, "C:\", 1&)

   ' ShellExecute returns a positive number less than 32 in case of error!
   If (nResult > 0 AND nResult < 32) Then
      ' Raise a custom error using the error number received from API
      Err.Raise vbObjectError+nResult, "ReportDefect", _
                "Unable to create new defect report"
   End If

End Sub
```

Adapting a Software Development Process to Varying Project Sizes

Sandy Raddue
Cypress Semiconductor
sur@cypress.com

ABSTRACT

This paper highlights the method developed at Cypress Semiconductor Corporation for adapting a software process definition to projects of varying sizes and complexity.

1.0 Introduction

When I first started at Cypress Semiconductor, there was a software process roadmap document in place for use by our division. It laid out a model to be followed that consisted of four milestones. These milestones were:
1. Product Proposal
2. Alpha Release (code complete)
3. Beta Release
4. Final Release.

It was apparent from the beginning that these four steps were terribly inadequate, and did not represent a “real” software development process. A lot of opportunities for economically improving software quality were being overlooked. Deadlines and milestones were being missed on projects, and product quality and effectiveness were suffering as a result. My job description included refining this process to make it more detailed, and more effective.

This paper discusses the varying levels of complexity, and presents the solution implemented at Cypress Semiconductor for project leaders and managers to evaluate the complexity and cost level of their projects, and determine where on the process complexity scale their project will fall.

The new software development process requires nine checkpoints, contrasted with the original four. It requires five documents (with a number of optional documents), contrasted with the original two (with test documentation optional). However, because this method is so unique and flexible, the majority of project leaders prefer the new to the old.

This paper and accompanying presentation will outline the processes to be followed from the most basic service pack release, to the implementation of a new, patentable software, and all size projects in between. The goal of the new software development roadmap at Cypress is to keep the necessary overhead to a reasonable level, but ensure that Cypress's needs are being met, from both quality and development perspectives.

2.0 History

As a definition, I need to explain some things at Cypress.

1) The term Milestone (Product Release (PR) and Software Release (SR)) means a formal review held with all project management present. The term review is used differently here than in standard Software Engineering terminology.
2) The software products generated by Cypress Semiconductor for use by an end customer are Hardware Design Language compilers, synthesis tools, and other support tools needed by our customers (engineers) to configure our programmable parts for use in their end products.
3) At Cypress, our customers are engineers, and therefore very savvy, and very intolerant of inferior products, because there is a direct effect on their own productivity and project schedules!

In 1997, the Software Process Specification in use by our division of Cypress Semiconductor encompassed the methodologies in use, and used limited Product Release milestone definitions. There were four milestones with accompanying Project Review (PR) meetings, consisting of
PR1. Product Proposal
PR2. Code Complete (alpha release)
PR3. Beta Release
PR4. Final Release

If you will notice, there is the initial management buy in PR1. Then the developers go off and do "whatever developers do." When the code is done, it is given out internally for "alpha test" (PR2), often the first time groups outside development would see any of the new functionality, or even descriptions of that functionality.

Alpha releases generally occurred much later than scheduled, because when the code was integrated "just before alpha" numerous issues were uncovered that should have been discussed and resolved prior to any code being written. Because of the lack of visibility into the development of the software, I have heard "It's already in the code - I'm not changing it now" many times, when, of course, adequate requirements definition and design reviews would have uncovered these issues much earlier, while they could still be repaired.

Beta release (PR3) was scheduled for 2-4 weeks following alpha, and was generally scheduled for 3-4 weeks duration. The software shipped for Beta release was generally held together by bandages and baling wire, and the customers figured out pretty quickly that they didn't want to use the software for their own designs. Therefore, we seldom had beta test cycles that were deemed successful. Many defects and requirements shortcomings were uncovered at this stage, thereby delaying the PR4 release, or causing large quality compromises.

PR4 (Final Release) was generally very late, and the project leaders were held fully accountable for the project slips. But, due to the lack of visibility into the process of the software development, they were at a loss to say what was wrong. The original process required no formal requirements review, no formal design of any kind, and no peer reviews of anything. Some of these items were done, but informally, with no way to track the history.

It was apparent from the beginning that a lot of opportunities for reviews, checkpoints, and other quality software methodologies were being missed or overlooked on projects, and that product quality was suffering as a result.

When I was hired four years ago, I was supposed to improve the existing software development process, and also start a formal QA group (read "test"). I observed and took a lot of notes, and made several revisions to the process documentation. Despite numerous metrics evaluations that proved we needed better specifications, the process milestones did not change - because "they'd always been that way." Many checklists and review points were added to the process, but project leaders continued to either ignore the milestone requirements, or combine them, sometimes at points that didn't make sense. Each project leader could negotiate away one or more milestones at their discretion, without necessarily meeting requirements of those milestones. It was inconsistent and hard to track, at best.

It was determined that the process had to have more checkpoints, called project reviews here at Cypress. Of course, when this was brought up, the majority of the software project leaders were opposed, sensing that process would just get "in their way" of getting the real work done. And in some cases, this was a very valid point. Why spend six weeks of overhead on a project that should REALLY take four weeks of development time? There had to be a better way. However, with upper management's backing, I refused to ease up on requirements documentation. Using measurements and metrics, it was proven that they had to define things better up front, so that we didn't spend so much time chasing non-existent problems at the end, when it is so expensive. So, where do we go from here?

3.0 Evolution

When the CEO of Cypress decided that all software produced for our programmable products should be presented to the customer on one CD, the software development process that was created within our division came under corporate-wide scrutiny. The other divisions had NO formal software processes defined, and weren’t doing things the same way, so our development process became a starting point. Upper management decided to define a top level process chart, filling one page, and showing all software efforts across the company. This chart is shown in Figure 1. The attempt was made to tie software

milestones more closely to those required for silicon development. The programmable products portion was one part of this document.

A cross-divisional team consisting of development managers, directors, and software QA engineers then began meeting once a week. As a direct spinout of the Level 0 Process Definition, the number of milestones within the corporate software process increased to nine.

The working group then broke each milestone in the top level process to its own section. We then determined the many steps needed for successful completion of software projects. One such milestone section, with all deliverables, activities, and project management reviews is shown in Figure 2.

FIGURE 1: LEVEL 0 PROCESS DEFINITION

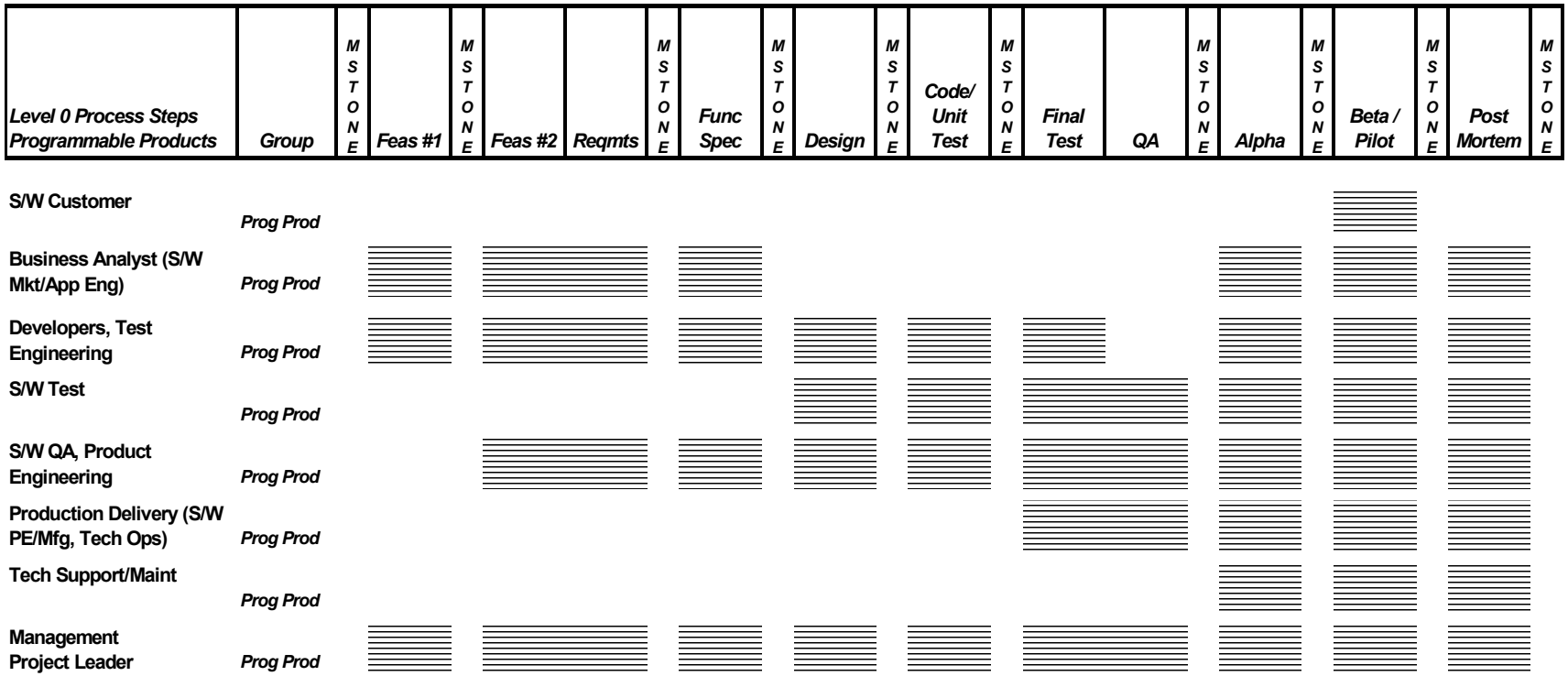
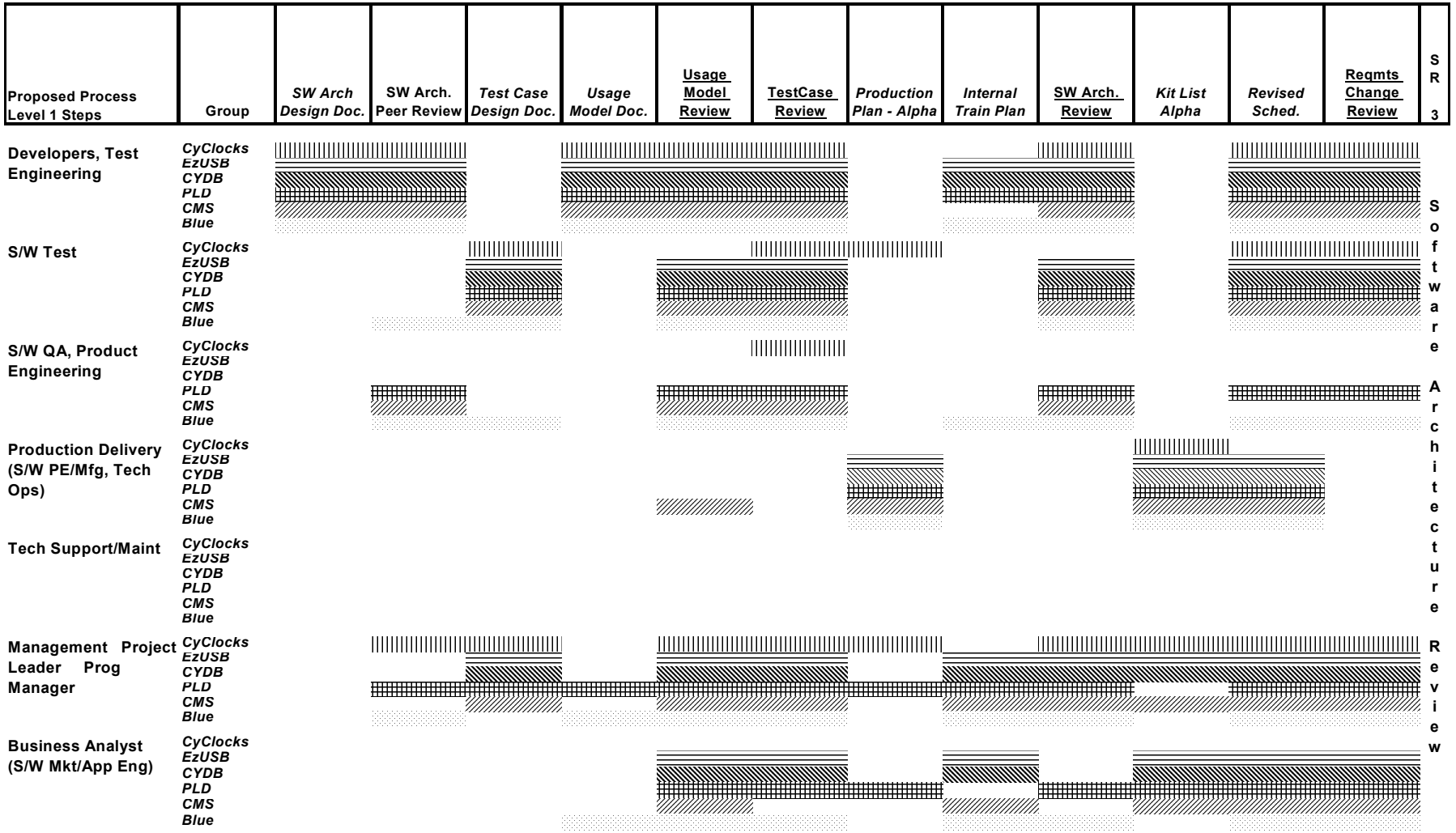


FIGURE 2: Software Architecture Definition



Each box across the top was discussed. The highlighted boxes indicate what actions were being done by each organization at the time this exercise was done. The boxes were broken into categories of Deliverables, Activities, and Internal Reviews. Definitions of each item were written. Consensus was reached that these were the things that needed to be done. All the team members were ok with this. For a while. For a couple of weeks, even. Until they each, independently, realized just how much overhead they had imposed on themselves. Then all of them, without exception, started fighting back, and saying the process was ludicrous -- we'd never get any product out the door at this rate -- this process is fine for YOUR project, but it doesn't apply to MINE, etc., etc. It then was obvious that the process would not be followed unless there was a way to tailor it to individual projects, ranging from 4-6 weeks development time to three years development time.

4.0 Resolution

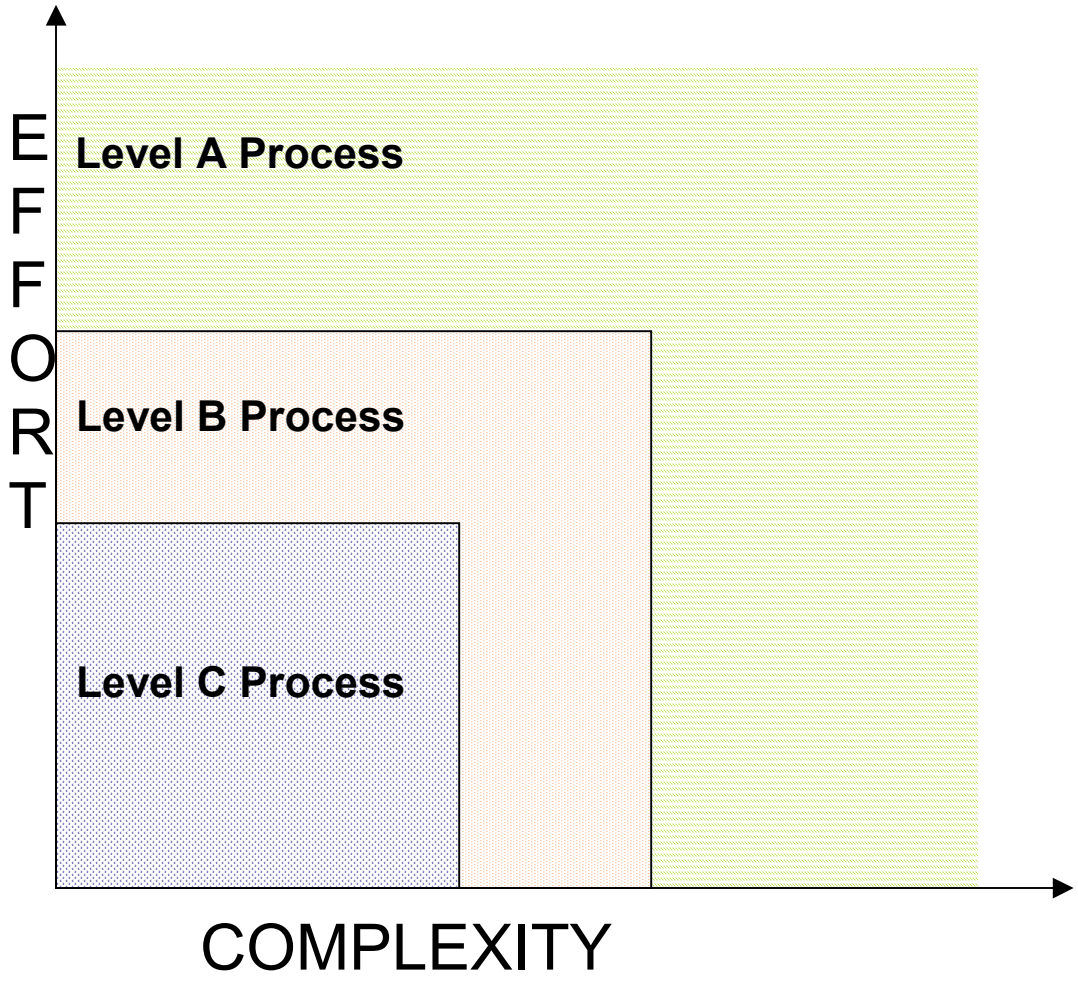
When the task of again rewriting the process specification arose, the steps defined in the cross-divisional exercise were used as a starting point. All members of the team emphatically agreed that sometimes process can actually get in the way of producing a good product in real time. With the pressure of time to market, and particularly the pressure of software support of Silicon that may already be in manufacturing, project leaders were overwhelmed with process requirements. Additionally, it had become very difficult to find qualified project leaders within Cypress, due to the amount of process/documentation overhead required for most projects. The following methodology was devised to determine the appropriate level of process for every product/project, and a way to get the project leader and management to agree to the level of formal process early on in the project life cycle.

The priority of software development is always to support our silicon in a timely manner. However, one core value of Cypress is “Follow the spec or change it.” In order to be in agreement with both of these tenets, the process definition was massaged to meet both needs as well as possible.

Since the complexity of projects varies from very complex to trivial, and the size of the effort varies from very small to very large, I concluded that some way to scale the process was in order. However, we did not want to miss any of the important milestones and/or deliverables along the way, so there had to be a way to incorporate all of these into the process.

The following graph is a very general method for determining which level of process compliance is necessary for each project.

Figure 3



The effort is defined in either people or schedule units, in person weeks, or calendar weeks. If the project ties up a significant amount of people for a short time, or a small amount of people for a long time, it may fall within the same area on the Y axis in the above chart.

Complexity is a relative judgement. A highly technically complex project is generally defined as one that requires new innovation – technology that has never been implemented before. A medium level of complexity may be defined as “routine maintenance” to a legacy code base, which may encompass fixes to the software to support new features. A low level of technical complexity may be a change in a configuration file, or a new operating system port of a well-defined system, or a new family member to an existing silicon family.

At the onset of the project, the project leader states which level of process compliance he intends to follow in the initial project proposal. Upper management must buy off on this proposal, including the level of process commitment. Once that initial level has been agreed upon, the project can proceed.

Level A is the most complex level of process overhead. It is used for the large-scale projects, or projects where the technology is innovative, and possibly patentable. Note: The term “patentable” is used here, because Cypress is frequently developing innovative software solutions to very complex problems, and patent applications are submitted as a result. Although the actual solution may only be a few hundred lines of code, the documentation required to support a patent application is very significant – this level of process compliance protects Cypress in this effort.

Examples of a Level A product would be:

- New User Interface to an existing product - because of large user community buy in, and large up front documentation and review effort, beta test effort, etc.
- New product line - innovative software technology - documentation level required to back up a patent application
- Significant change to product look and feel
- Brand new product – bringing new Silicon and new software to the user for the first time.
- A dot.0 (zero) release

Level B is currently defined as the median level of process. It is used for smaller scale projects than level A, but for projects that still require significant resources or technological innovation.
Examples of a Level B product:

- Significant rework to program functionality
- Change of feature set where users would want a beta test cycle.
- a dot.y release (y>0)

Level C is the process with the least amount of overhead, at least right now. It is defined as the bare minimum.
Examples of Level C include:

- service pack releases
- new devices within a given family
- new options within a feature
- Template update to word processor
- Printer driver updates

Level C is used for those projects that are extremely simple, such as adding new chip packaging options within an existing family, or adding a canned library type element for use by the customer. In other words, those changes and updates that involve very little new intellectual property. Well over half of the software releases at Cypress fall into this category.

When this hierarchical solution was presented to management and project leads, it was met with emphatic acceptance! Finally a way to follow the process AND get product out the door at the same time!

5.0 Implementation

All users of this new process definition have required some level of training. Most of the training was conducted during an all day session, with all project leaders and managers in one room. A chart (Figure 4) was published showing both how the software milestones mapped to silicon, and how the old milestones mapped into the new. It also provided a conversion chart for changing existing project milestones going forward. All project leaders were asked to comply with the new process.

Figure 4

	Old	New	Silicon
SPP	NA	SR0	PR0
Customer Requirements Review	NA	SR1	EROS/PR1
Functional Specification Review	PR1	SR2	IROS
Software Architecture Review	NA	SR3	PR2
Code Complete	NA	SR4	PR3/Tapeout
Alpha Release	PR2	SR5	
Beta Release	PR3	SR6	PR4
Release to Production	PR4	SR7	PR5/PR6
Post-Mortem/Production Review	PR5	SR8	

There are now over 40 checklists helping define this process. For a level A project, there are five required documents, with 21 optional, compared with two required and three optional with the old process. Each of the milestone checklists has a chart published with it, stating which level of process is required to complete which items. This gives the project leads ample notice as to what is required at each milestone review.

The following (Figure 5) is an actual checklist used for SR3 (the Software Architecture Review milestone). This is the same milestone detailed in Figure 2.

The table that follows Figure 6 maps the checklist back to the Level 1 process map.

Figure 5

Software Review Three Checklist

A	B	C	Initials	Owner	Item
_____	_____	_____	_____	Proj. Lead	Overview of design presented
_____	_____	_____	_____	Proj. Lead	Software Arch. Peer Review Held
_____	_____	_N/A_	_____	SWQA	Test Case Design Document Complete
_____	_____	_N/A_	_____	SWQA	Test Case Design Review Complete
_____	_N/A_	_N/A_	_____	Proj. Lead	Usage Model Document Complete
_____	_N/A_	_N/A_	_____	Proj. Lead	Usage Model Review Complete
_____	_____	_____	_____	Proj. Lead	Demo or results of prototyping presented
_____	_____	_____	_____	Proj. Lead	Requirements alloc. to software arch. performed
_____	_____	_____	_____	Proj. Lead	Requirements change review held
_____	_____	_____	_____	Proj. Lead	Performance projections presented
_____	_____	_____	_____	Proj. Lead	Updated schedule presented
_____	_N/A_	_N/A_	_____	SWPE	Alpha Production Plan
_____	_N/A_	_N/A_	_____	Devel.	Internal Training Plan
_____	_N/A_	_N/A_	_____	SWPE	Alpha Kit List
_____	_____	_____	_____	Proj. Lead	Issues presented (include schedule, resources)
_____	_____	_____	_____	Proj. Lead	Third party plan presented, contract reviewed.
_____	_____	_____	_____	Proj. Lead	Top 10 risks presented for review

Figure 6

SR2 to SR3 Process Map	Level A	Level B	Level C
Software Architecture Design	X	X	X
Software Architecture Peer	X	X	X
Test Case Design Document	X	X	
Usage Model Document	X		
Usage Model Review	X		
Test Case Review	X	X	
Alpha Production Plan	X		
Internal Training Plan	X		
Software Architecture Review	X	X	X
Alpha Kit List	X		
Revised Schedule	X	X	X
Requirements Change Review	X	X	

6.0 Conclusion

The first few projects are just wrapping up using this new process. The new process has been very well received so far. Most of the project leaders seem happy with this improvement, and see the value of more reviews more often. There is still some grumbling, but since small and large projects both have their interests served, everyone seems more content to follow the process, and not be so inventive circumventing the requirements. In addition, more project leaders are accepting the role for subsequent projects. This is of great benefit, because we are not spending a lot of time training new project leads.

We are updating schedules more often, and are evaluating risk on a regular basis. There is better visibility into the projects, and more warning when things are going off track. Upper management is very pleased with this benefit.

Because test and requirements reviews are held at each pertinent milestone, development engineering and test engineering have better communication and visibility, and defects are turning up earlier, even with a lack of mandatory code inspections.

In other words, things are just better! It works, because everyone has bought into it from the beginning, and made the commitments necessary to make it work.

Applying Function Point Analysis to Requirements Completeness – A QA Guide to Function Point Sizing

by
Carol Dekkers, Quality Plus Technologies, Inc. and
Mauricio Aguiar, Caixa Economica Federal

Abstract

Requirements issues abound in system development, despite many models and methods intended to verify that requirements are complete. This article highlights how the software sizing technique known as Function Point Analysis (FPA), delivers value as a structured requirements review. While its historical usage has been confined almost exclusively to quantifying software size, FPA is gaining popularity as a useful, structured method for reviewing requirements. When used during software development to verify requirements completeness, FPA delivers more than mere numbers for software size – the FPA documentation reflects the full, known set of functional user requirements.

Introduction

There is a wide diversity of traditional approaches for identifying and gathering software requirements including JAD (Joint Application Design) sessions, requirements management techniques, prototyping, Rapid Application Development (RAD), eXtreme Programming and others. When done properly, these techniques typically deliver some form of documented user requirements. After a series of user and peer reviews, these "formal" requirements are typically assumed to represent the complete set of user requirements. In general, however, the full set of user requirements is not complete until the end of the project and continue to emerge as the project progresses. As a result, the project encounters rework, schedule slippage, and budget overruns, the extent of which depends on the degree of originally unknown requirements. With some DoD projects, this problem is further compounded today with requirements that are outcome or performance based, and functional requirements are developed as part of the design process. When these projects emerge, they challenge our traditional requirements approaches (e.g., how does one document software requirements when the performance requirement is to launch a projectile from a range of 200 miles with an impact of xxx?), and this article is not intended to solve these types of requirement issues.

This article addresses those projects where functional user requirements are articulated (or should be), and outlines how Function Point Analysis can be used as one additional tool that can identify missing requirements, gauge requirements completeness and uncover potential defects. Our experience holds that Function Point Analysis is often more effective than peer or user walkthroughs in identifying the full set of functional user requirements and uncovering potential defects. In fact, the benefits to be gained by applying Function Point Analysis to functional user requirements can be more valuable than the mere function point size of the software.

There are two audiences for this article:

1. Development teams who already use or who are considering using Function Point Analysis on their projects. The information provided here is intended to

increase the cost-effectiveness of FPA and leverage its use as a requirements completeness check; and

2. Development teams who do not use Function Point Analysis, but would like additional tools to increase their requirements effectiveness. The concepts outlined in this article can be applied to any project without the need to complete all steps in the method.

Requirements

Why is it that the "right" (i.e., correctly and accurately stated) set of software requirements is so elusive in our industry? Requirements problems involve getting the requirements right (correct and accurate), getting the right requirements (the complete set of functional user requirements), and involve more than the *specifications of requirements*. One of the biggest problems software developers encounter is being able to judge whether requirements are sufficiently complete before beginning formal design and coding.

Before we discuss how to apply Function Point Analysis to the issue of requirements completeness, it is worthwhile to identify the three major types of software requirements. Together, these form the overall project's user requirements:

1. Functional User Requirements -- These are the logical business or user functions that the software must perform. All software from real-time missile guidance systems to business accounting software has functional requirements that must be performed. These include the elementary processes that must be supported to input, process, manipulate, output and interface data to, from and within the software. The Functional User Requirements are the specific area of requirements that FPA addresses.
2. Non-Functional User Requirements -- These are the technology independent user /business constraints that the software must meet. Non-functional requirements include quality and performance requirements such as portability, usability, security, dependability, reliability, speed, etc.[1] Part of the Function Point Analysis technique can assist with this type of requirement.
3. Technical Requirements -- These are the user requirements for a specific hardware/software configuration or a particular technical configuration that must be delivered. For example, the technical requirements may specify an Oracle database or a multi-tiered hardware solution. While these software specifications are as important as the other two types, Function Point Analysis does not address this type of requirements.

The remainder of this article specifically pertains to the first two types of software user requirements: the Functional and Non-functional requirements.

Please note that when we talk about functional requirements and users, we mean users in the global sense. Often the challenge is exactly that – figuring out whether the specified requirements cover the whole set of stakeholder needs. Sometimes one set of stakeholders may be forgotten when requirements are written. Thus, it is of critical importance to have as many review tools as possible using differing frames of reference.

[1] Quality requirements can be found in the ISO/IEC 9126:2000 suite of standards which address many of the "ility" constraints such as Portability, Security, Usability, Reliability, etc. Contact ISO for further details.

Traditional Ways of Checking Requirements' Completeness

While both the functional and non-functional requirements strive to be unambiguous, correct and complete, it is easy to write down and check business rules for ambiguity and correctness with a user. It is a problem however, to ensure that one has identified the full set of functional user requirements. A frame of reference or two is needed. There are two basic ways such a frame of reference is typically provided using existing techniques:

(1) ***A Theory Based Model*** - A theory based frame of reference may be used such as structured analysis, information engineering or data modeling. The analyst will decompose the problem and look for abstract structures like data flows, processes, data stores, etc. Requirements will be considered complete when the abstract structures *make sense* to the analyst (e.g., data stores have both incoming and outgoing data flows).

(2) ***Personal Experience*** - The analyst may have worked with other business systems, similar to the one being analyzed. In that case, he will possess a subjective frame of reference composed of all the business structures and rules he has encountered previously. The analyst will decompose the problem and look for known structures and rules conformant to his own model of reality. Requirements will be considered complete when the identified structures match the *analyst's model of completeness*, which is subjective (e.g., accounts receivable will have been either received or marked as delinquent).

Ordinarily, the analyst will work with a mixture of the first two frames of reference to increase the quality and clarity of the documented set of known software requirements, and increase the relative percentage of the known to total requirements. Throughout the project, he will integrate his personal experiences with the theoretical knowledge. Increasingly, however, this is insufficient to gain enough completeness coverage. It is in the analyst's interest to use as many frames of reference as possible.

Ideally, frames of reference should be orthogonal, i.e., they should not overlap. Each frame of reference should provide unique information not available in the other models.

While some may argue that orthogonality reduces duplication and this may be desirable, others may state that redundancy in sources is a clear benefit. Whatever your viewpoint, the issue here is to use as many methods as possible to ensure that all bases and perspectives are covered when reviewing requirements. Just because a review method may be complicated doesn't mean it is necessarily thorough—it is the combination of different reviews that will lead to the best results and most complete functional requirements.

What's the Point of Function Point Analysis?

Function Point Analysis (FPA) provides an additional frame of reference for checking the completeness of the functional requirements, together with some of the non-functional requirements. It is different from the first two frames of reference because it provides a unique, user-focused perspective. Function Point Analysis examines the set of functional user requirements in terms of data and its movement / manipulation (transactions) *as understood and expressed by the users*, and on this basis determines the software's functional size. As such, FPA can and should be used in

addition to the Theory Based and Personal Experience models previously mentioned to ensure that the functional user requirements are complete.

Function Point (FP) Basics

Function Points measure the size of a software project's *logical user functionality*, as opposed to the physical implementation of those functions as measured by lines of code (LOC). Function Point Analysis examines the functional user requirements that are to be supported or delivered by the software. It then assigns a weighted number of Function Points to each logical user function as outlined in the Function Point Counting Practices Manual[2] and calculates the software's Function Point size.

In simplest terms, Function Points measure ***what*** the software must do from an external, user perspective, irrespective of how the software is constructed. While analogies from other industries such as building construction and manufacturing *attempt* to describe how function point analysis works with software, none provides a perfect fit. In basic terms, Function Points reflect the functional size of software, independent of the development language and physical implementation. Function points can be likened to the functional area of a building by summing up the size of its rooms. Function points quantify the functional user requirements (the software "floor plan") by summing up the size of its functional "components". As in building construction, one cannot manage a project if only the square foot size is known and system development cannot be managed purely on the basis of FP size. (Note that when matters of software estimating are discussed, many more factors are involved beyond the functional size of software including the type of software, technical requirements, number of users, geographic locations, etc.)

How to Use Function Point Analysis to Gauge Requirements Completeness

For an introductory article on Function Points, refer to the February 1999 issue of *CrossTalk*, The Journal for Defense Software Engineering (free copies can be downloaded from the CrossTalk website at www.stsc.hill.af.mil/crosstalk). When performing a function point (FP) count, all of the *known* functional user requirements for the software are analyzed, weighted and counted using the standard identification method. It is the process of analyzing these functional user requirements that most of the errors and omissions in the requirements are uncovered, as described below. Each bulleted point is a step in the actual FP counting process.

- **Determine the project scope and purpose of the function point count.** (For example, Function Points can be counted to quantify the size of a new development project or an enhancement / renovation project, or to size an existing base application).

In this step it is useful to document the specific name and date of the source document(s) used as a basis for the count (e.g., System ABC Requirements document V1.2 dated March 22, 2000). This provides for traceability of the logical functions included within the functional requirements as of a specific point-in-time and is useful for gauging scope creep during the project. It can also contribute to the historical base for gauging future projects as outlined below.

[2] The Function Point Counting Practices Manual (CPM) is maintained by the International Function Point Users Group (IFPUG) and is currently in Release 4.1 (1999). Contact IFPUG at www.ifpug.org to obtain a copy..

By documenting (even in a few lines of text) the project scope and purpose of the FP count, assumptions about the project are clarified and can identify oversights in requirements. For example, if the purpose of the FP count is to size the amount of customization required for a COTS (Commercial Off The Shelf) package, the scope will include only the customized functions, not the entire package. This provides a delineation of what is included in the project.

- **Identify the application's logical boundary.** This step identifies the functions that the software must perform, together with the interfaces to external users, departments and other applications. The application boundary for FP counting is not the same as a *physical* application boundary. Instead, it is the logical boundary which envelopes self-contained user functions that must exist to deliver the user requirements. This "boundary" separates the software from the user domain (users can be people, things, other software applications, departments, other organizations, etc.). As such, the software may span several physical platforms and include both batch and on-line processes -- all of which are included within the logical application boundary. For example, an Accounts Payable system would typically be considered one application in function point analysis -- even though it may reside across multiple hardware platforms in its physical installation.

Because each "application" or software system has a separate application boundary (e.g., Accounts Payable would typically be one application, Fixed Assets may be another), a project context diagram consisting of several circles to denote the various application boundaries is often drawn as a part of the functional sizing process. In cases where an enhancement project renovates an application for which there is little documentation, this step provides a context diagram that can be later used for communicating with the users about the software system. In a particular client situation, this visual depiction of the various application boundaries and the applications with which they interfaced, provided a basis for discussions about client/server migration of certain applications because our diagrams showed which applications would be affected by the migration of a central application. Because these "context" diagrams are visual in nature and independent of technology, their review with users and developers often leads to the discovery of interfaces that were previously discussed, but which are missing from the written requirements. In addition, this step together with subsequent steps, clearly demarcates the logical boundaries between user applications. By clarifying which functions lie within which applications, there is less likelihood of a set of requirements being overlooked. For example, if a project team assumes that another application will maintain a set of common data, a review of the context diagram showing the interface to the other application may reveal potential oversights.

- **Count the Data Functions:**
  - Identify, weight and count the Internal Logical Files (ILFs). These are the persistent logical entities or data groups to be maintained through a standard function of the software, and
  - Identify, weight and count the External Interface Files (EIFs) which are persistent logical entities that are referenced from other applications, but not maintained. Typically this data is used in editing, validation or reporting processes of the software.

In performing this step of identifying and classifying the persistent **logical** entities as internal (i.e., maintained) and external (i.e., referenced-only) for the software, it is helpful to document the entities by drawing circles around the entities (and their included sub-entities) on a data model or entity-relationship diagram, if there is one. If there isn't a data model or an entity-relationship model, one is essentially created in this step building on the context diagram created in the previous application boundary step. Tables / files that are created only because of the physical or technical implementation needs are not counted, nor are hard-coded tables / files (i.e., not maintained through a standard process of the application). This step also records the number and types of the logical data elements if they are known and if they are not already identified in the requirements. This provides a checklist of data entities against which one can gauge the consistency and completeness of transactional (manipulation of data) functions.

By reviewing the entities, whether on a data model or hand-drawn context diagram, and whether they are inside the application boundary (i.e., to be maintained by the software) or external (i.e., to be referenced only) often gives rise to clarifying comments. For example, such comments might include: "Why is that entity external? I thought we needed to be able to update that entity, which would then lead to a discussion that either confirms the original requirements or reveals an inconsistency in understanding and a change in the diagram. When the review is combined with the transactions outlined in the next step, the majority of (potential) requirements mismatches are identified.

- **Count the Transactional Functions:**
  - External Inputs (EIs) which are the elementary processes whose primary intent is to maintain the data in one or more persistent logical entities (ILFs) or to control the behavior of the system. (Note that these External Inputs are functional unit *processes* and not physical data flows or data structures);
  - External Outputs (EOs) which are the elementary processes whose primary intent is to deliver data out of the application boundary, and which include at least one of the following: mathematical calculation(s), derive new data elements, update an ILF, or direct the behavior of the system; and
  - External Queries (EQs) which are the elementary whose primary intent is to deliver data out of the application boundary purely by retrieval from one or more of the internal or external logical entities (ILFs/EIFs).

  This step is where the majority of missed, incomplete or inconsistent requirements are identified. The following list provides some examples of the types of discoveries that can be made using Function Point Analysis:
  - If a persistent, logical entity has been identified as an Internal Logical File (i.e., maintained through a standard maintenance function of the application), but there are no associated External Inputs processes, there are one or more mismatched requirements:
    - Either the entity is actually a reference-only entity (in which case it would be an External Interface File), or
    - There is at least one missing requirement to maintain the entity, such as Add entity, Change entity, or Delete entity.
  - If there are data maintenance (or data administration) functions identified for data, but there is no persistent logical entity to house the data (ILF), the data model may be incomplete. This would indicate the need to revise the data requirements of the application.

- If there is a data update function present for an entity identified as reference only (EIF), this would point to the fact that the entity is actually an ILF. The data requirements are inconsistent and need to be reviewed.
- If there are data entities that need to be reference by one or more input, output or query functions and there is no such data source identified on the data model / entity-relationship diagram / context diagram, the data requirements are incomplete and need to be revised.
- If there are output or query functions that specify data fields to be output or displayed for which there is no data source (i.e., no ILF or EIF), and the data is not hard-coded, there is a mismatch between the data model and the user functions. This indicates a need to revisit the data requirements.
- Most maintained entities (ILFs) follow the AUDIO convention rule[3]-- that is, each persistent logical entity typically has an **A**dd, **U**pdate, **D**elete, **I**nquiry and **O**utput function associated with it. Not all entities will follow this pattern, but AUDIO is a good checklist to use with the ILFs..

- **Evaluate the complexity of non-functional user constraints using a value adjustment factor (VAF).** Through an evaluation of the fourteen General Systems Characteristics (GSCs) of Function Point Analysis (e.g., the GSCs include include performance, end user efficiency, transaction volumes, and others), an assessment can be made about the complexity of the software. The impact of user constraints in these areas is often not enunciated or even addressed until late in the software development life cycle, even though their influence can be major on the overall project.

Examining the user requirements with these non-functional, user business constraints in mind can provide the following types of valuable information:

- The non-functional requirement due to transaction rate peak loads may necessitate 24 hour X 7 days a week availability. This will have a critical impact on the resulting project.
- Special protection against data loss may be of critical importance to the users' business and must be specially designed into the system. This must be identified up-front to avoid any unforeseen impact.

FPA provides an objective project size for use in estimating equations (together with other factors) or to normalize measurement ratios. The process itself checks whether the full set of functional user requirements has been identified and can uncover defective and missing requirements. The following table summarizes how to use FPA to uncover requirements defects. The final column illustrates where and what type of potential requirement problem there might be.

Data Function	ILF	EIF	Transaction Function	EI	EO	EQ	Indicator of Requirement Problem	Details of ***Potential*** Requirement

[3] Per personal discussions with John Van Orden, Certified Function Point Specialists, formerly of Gartner Group and a member of the Quality Plus Technologies, Inc. consulting team. John uses the AUDIO acronym as a quick check of requirements completeness during function point counting. Some others use CRUD (create, read, update, delete).

								Problem(s)
Employee entity (maintained)	X	X					X	1.The same entity cannot be both maintained & externally referenced. 2. No maintenance functions.
Monthly sales	X		Add sales, Delete Sales	X X			X	– Can errors be corrected/updated? – Are there no reports that use or query this data?
			Account report			X	X	1. There is no source (no ILF or EIF) in the application that contains account information. Where is data coming from?
% breakdown of FPA components	50%	0		10%		7%		Based on the standard % profile, questions would include: 1.Why are there no external files -- were these overlooked? 2.Why are there no queries in the requirement when the "standard" profile shows 10% of the FP were allocated to browse/query functions. 3. Are all the transactional functions identified? (i.e., the profile for this software does not follow the norm)
Standard % profile of FPA	30%	10%		40%	10%	10%		

components [4]								

Benefits after the Requirements Phase

The value of having a documented set of functional user requirements (and the non-functional requirements that FPA addresses) such as that provided by the FPA process goes far beyond merely the requirements phase. Hill and Tinker Air Force Base's Materiel Systems Groups (MSG) found this to be the case and an example from Hill Air Force Base serves to illustrate this point. MSG at Hill would attach a full listing of the functional requirements (using the FPA documented breakdown of FP components we had counted) to the software project estimate sent in to headquarters. Later, when questions arose about a particular set of functionality and whether or not it had been included, the group would refer to the FP listing to see if that particular functionality was listed. If it was not, it was clear that the functionality had not been included in the estimate, and a decision was then made about whether or not to include it and increase the estimate. This simple set of documented functions minimized the finger pointing and blaming of "who said what and when" and reduced the discussion to whether or not the functions were included in the specifications submitted. Additionally, when scope changes emerged later in the project, as they inevitably do, both groups were in a position to adjust their FPA sizing and quickly assess the impact of scope change on the project.

While other requirements review and tracking techniques can also provide value, FPA is a simple method that delivers both a functional size of the software (useful for estimating) AND can assist with the requirements processes.

Summary

Today's software analyst needs all the assistance he or she can find to help in the quest for complete (and known) user requirements. The framework provided by the structure of the Function Point Analysis (FPA) technique gives the analyst one extra frame of reference against which to gauge the completeness of the known user requirements. Requirements defects will still occur no matter how many frames of reference are used, however, the use of Function Point Analysis to augment the traditional Theory Based and Personal Experience frames of reference will increase the analyst's ability to ensure the software requirements are complete.

Is FPA worthy of consideration from your organization? The answers all depends on where your organization is at and what you want to achieve with software measurement. FPA is one tool that can assist with your requirements processes and also provide a quantitative value to size your software. For those of you who have been using FPA *only* to arrive at a software size, you can gain valuable benefits by applying FPA as a structured review, especially when your requirements are deemed "complete".

==========

About the authors:

Carol Dekkers is a leading authority, presenter, author and consultant in the area of Function Point Analysis and software measurement. She is the President of Quality Plus Technologies, Inc., a management consulting firm specializing in helping DoD and private organizations succeed with function points, invest wisely in software measurement, and improve their bottom line through process improvement. Carol is the

[4] The breakdown of standard percentages here is fictitious and intended to show a sample profile that could be developed using FP counts of a sample size of several similar applications.

Past-president of the International Function Point Users Group (IFPUG), and an ISO project editor on the Functional Size Measurement project (ISO/IEC/JTC1/SC7 WG12) on behalf of the U.S. Recently, Carol was named one of the 21 New Faces of Quality for the 21st Century by the American Society for Quality (ASQ). She is a professional engineer, Certified Function Point Specialist (CFPS) and a Certified Management Consultant (CMC). Send comments or inquiries to Carol at dekkers@qualityplustech.com .

Mauricio Aguiar is a software manager with Caixa Economica Federal, a leading Brazilian government bank with over 2,000 branches. His extensive experience spans 25 years in software management and includes the application of accelerated learning in IT. Mauricio is also the President of the Brazilian Function Point Users Group and serves on the IFPUG Communication and Marketing Committee. A professional engineer and systems analyst with a Master's Degree in Neuro-Linguistic Programming, he is a member of PMI, ASQ and IFPUG. Contact Mauricio by email at mauricioaguiar@yahoo.com .

Driving Quality Improvements with an Effective Software Metrics Program

By

Marsha Holliday, Giora Ben-Yaacov, Pramod Suratkar and Karen Bartleson
Synopsys Inc., Mountain View, CA

==========

ABSTRACT

How do we get management and the product teams to emphasize the importance of quality, and therefore, process improvement?

In the past, factors like fast development schedules, low cost, and high quality were generally considered to be mutually exclusive. Today, products of increasing complexity must be designed and manufactured at reduced costs in seemingly impossible timelines. Under this kind of compounded pressure, something, usually quality, has to give. Or does it? We don't believe so.

This paper describes how a successful software metrics program was deployed at a leading market-driven software company. The deployment of this program has succeeded in increasing quality awareness across the organization and is continuously guiding management and all employees to include quality as an important element of the decision-making process.

The key message of this paper is that the success of any software metric program largely depends on four elements:

- Focus on a few key metrics and indices that provide a view into the quality of the software products, the support services, and the perception and satisfaction of customers.
- Deployment that is light on resources and relies on existing sources of data.
- Effective use of different communication vehicles for consistent sharing of metrics and trends with all levels of management, software product teams, field application engineers, and the business community within the organization.
- Enlisting strong and continuous senior management support.

The success of a software metrics system can be observed in two ways:

- By an increase in quality awareness across all parts of the organization
- The proliferation of the metrics as a primary instrument in making business decisions

BIOGRAPHY

Marsha Holliday

Marsha is a Senior Quality Engineer at Synopsys. She has an extensive software quality background, including software quality improvement program management, product testing, product development and manufacturing process improvement. Before joining Synopsys, Ms. Holliday was the Software Quality Assurance Director at Intermedics, Inc. At that position, she spent 13 years directing software quality assurance for pacemaker design and manufacturing.

Giora Ben-Yaacov

Giora has B.Sc and M.Sc degrees in Engineering from the Technion in Haifa Israel, and a Ph.D degree in Engineering from the University of Cape Town in South Africa. Giora's 30 years of experience in software development and quality improvement programs include a spectrum of results ranging from that of individual contributor, through project leader, program manager, instructor, consultant, and, since August 1998, a Quality Architect with Synopsys Inc.He has authored and presented more than 35 technical papers.

Pramod Suratkar

Pramod has a M.Tech degree from Indian Institute of Technology, Bombay and M.S. and Ph.D degrees in engineering from University Of Wisconsin in Madison. He has 30 years of extensive marketing research, business process reengineering and software quality improvement program management experience. Before Synopsys, he spent 12 years in quality management in telecom, database and energy industries.

Karen Bartleson

Karen Bartleson is director, Quality and Interoperability at Synopsys. With more than 20 years experience in EDA, she is currently responsible for initiatives and programs that further Synopsys' product quality and EDA interoperability. Before joining Synopsys, Ms. Bartleson was CAD manager at United Technologies Microelectronics Center and manager of Logic Analysis at Texas Instruments. Ms. Bartleson is a board member of Accellera and an active member of the EDA Consortium interoperability and quality committees. She holds a BSEE from California Polytechnic University, San Luis Obispo, California.

INTRODUCTION

In this paper we share our real life experience in deployment of a very successful software metrics system at Synopsys Inc., a leading market-driven high tech software company at the heart of the Silicon Valley.

Synopsys, Inc., headquartered in Mountain View, California, creates leading electronic design automation (EDA) software tools for the global electronics market. The company delivers advanced design technologies and solutions to developers of complex integrated circuits, electronic systems, and systems on a chip. The company has more than 3000 employees world-wide, more than 50 software development product teams, and an annual revenue of over $800 million.

Today, leading edge technology, tight time-to-market deadlines, and product & service quality largely drive the business success of our customers. The quality of EDA software tools and processes is therefore essential to our customers' success. Moreover, with fierce competition among tool vendors, quality becomes a critical distinguishing factor. Customers often select EDA vendors based upon the perceived quality of their products and support.

Quality metrics can provide insight into where, when, and what kinds of problems our customers face. Once these problems are understood, the quality issues can be prioritized and resolved systematically.

This paper describes the following topics:

- The challenge
- The metrics
- The multiple reporting schemes
- The results

THE CHALLENGE

Synopsys has grown into the leading EDA tools developer, with over $1 billion in new orders expected this year. The technology and the quality of our products have been highly regarded in our industry. The quality of our products has been the outcome of having excellent people doing excellent work, using many different quality processes. With rapid growth of the company, establishing a more formalized process for evaluating quality has become the key activity for the corporate quality group.

We established a challenging set of goals:

- Increase quality awareness throughout over 50 product groups
- Bring consistency to the internal and external quality view of our products
- Focus on our overall quality perspective, instead of focusing only on the next release, the immediate problems, or the "squeaky wheel"
- Keep and increase the quality awareness at the top levels of management
- Measure results and continuously improve the program

Synopsys has nearly 3000 employees, including over 1000 engineers creating our software products. There are over 50 product teams. When we began our quality improvement project, each team had its own way of viewing and measuring the quality of its products. Each project or business unit created the quality processes and measurements used by their team. Many of the products and teams joined the company as part of acquisitions. Each of these "acquired" teams had their own quality processes and measurements.

We were challenged by the "Not Invented Here" syndrome. We had to choose metrics that all the teams could agree to and use easily, without feeling that the program was an added burden. This was one of the key elements to our program:

Choose metrics that are easy to understand and use

With so many product groups viewing the quality of their products in so many ways, we realized that there was a need for a consistent view of quality. There was always interest in the quality of our "flagship" products, and of course there was interest in a product if it was having quality issues. But, if the program was going to have an effect on the quality culture, this program had to provide quality awareness throughout the company, not just in specific areas.

Metrics are great to have, but do they get used? A lot of times, metrics are paid attention to only when there already is a problem. We wanted these metrics to be part of our culture, reviewed regularly and acted upon by the teams. Then we could identify product quality issues quickly, before bugs and quality became a major issue with our customers.

We needed to get the metrics in front of the product teams, so they could quickly review them. These important issues became another key element of the program:

Provide easy to access metrics for all products on a monthly basis

Many times quality is viewed one bug at a time. The development team is busy with the next release, the immediate problems, the "squeaky wheel." They don't have a chance to see the big picture of how their product is performing. We wanted to view how the product was performing over time.

Our products vary greatly in size, complexity and number of users. You should not compare the detailed metrics for one product to another. But try telling that to the product teams, or even management! Everyone wants to see how their product did compared to the rest of the company. We needed to find a way to provide a reasonable overall metric to meet this need. Now we had another element of the program:

Focus on the overall quality perspective

Quality awareness at the top levels of management is critical to the success of a quality program. Our next goal was to ingrain quality improvement into the corporate culture. We needed management to make quality one of their goals. In order to gain their interest and action, we needed a straightforward metric that summarized the state of our quality.

For management, a picture tells a thousand words. The picture of our quality metrics had to get in front of top management. Quality needed to be reported at the same level as sales and engineering accomplishments were reported. Here's another element of our program:

Report one key metric at the highest levels of management

Resources and funding are usually minimal when it comes to a metrics program. We had to build this program using existing resources. Development groups would not be supportive of another tracking system just to support our program. It was important and cost-effective to find the data for this program in existing tracking systems:

Utilize existing sources of data

There was just one element left. Once you have a metrics program, how do you know if the program is having an effect? We needed to measure the results of our program and be able to report on the results. Showing that the program was effective would promote interest and activity.

Measure and promote success

THE METRICS

The overall objective of this metrics program is to improve our response to customer requests for higher software reliability. Quality metrics can provide insight into where, when, and what kinds of problems our customers face. Issues include product performance as well as our responsiveness to customer needs. Once these problems are understood, the quality issues can be prioritized and resolved systematically.

The process of choosing the metrics for our program went through many revisions. We selected measures, prototyped them, received input from the product teams, and revised again and again. Getting input from the product teams was critical, but getting the product teams to give input was difficult. In order to get this feedback, we went ahead and started the program. Once the metrics were publicly documented and distributed, the product teams were happy to give us feedback!

An interesting facet of the EDA industry is the speed at which the products designed by our customer change. Our tools must keep up with this speed in the marketplace. Customers on the leading edge are willing to accept our products with less-than-perfect quality. But this only is acceptable if our support is outstanding and our responsiveness to their complaints is fast. We needed to ensure these metrics would give us a view into customer satisfaction.

When considering the sources of the data, we realized that no matter what value these metrics were going to add, we needed to ensure that this program did not discourage engineering from reporting bugs. Many bugs are found internally, and we feel that this is an important facet of the product quality. However, if the engineering team feels that they are being measured on this, they could become wary of reporting the bugs that they find. To avoid this situation, we chose some metrics that separate the customer-reported bugs from the internally-found bugs.

To measure both product quality and customer responsiveness, we selected a set of metrics that gives a view to both areas.

Incoming Rate (Customer + Internal)

One measure of product quality is the rate of incoming bugs reported against the product. In order to get a true picture of the product quality, you must track both customer-reported bugs and internally-reported bugs.

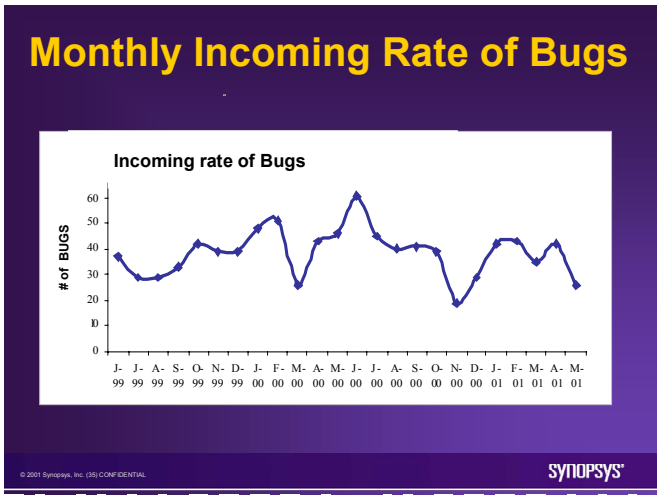


Figure 1: Incoming Rate (customer + internal) Metric for a Product

Backlog (Customer Only)

A view into both the product quality and our customer responsiveness is given by looking at the backlog of open bugs. If the product has a lot of open bugs, it shows that the quality is low. Looking specifically at the backlog of customer-reported bugs, we can get a sense of our responsiveness to our customers. If the backlog is high, our customers are likely to be frustrated and unhappy.

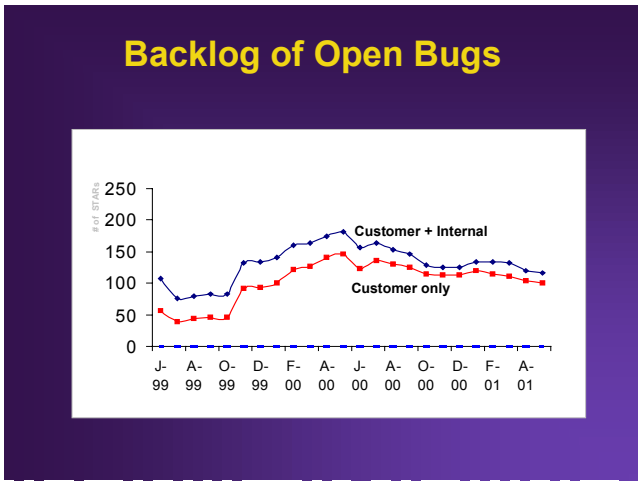


Figure 2: Backlog Metric for a Product (customer only and customer+internal)

Backlog (Customer + Internal)

The focus of maintenance efforts tends to be on customer-reported bugs. The backlog can grow quickly with internal bugs if attention is not given to them. Once again, it is important to include metrics on bugs that are found internally in order to get a true view of product quality.

Weighted Defect Count (Customer Only)

The criticality of open bugs is a good way to view the quality of the product. This metric applies weighting to the bugs in the backlog based on the criticality of the bug. This will quickly focus attention on the product if there are a number of critical bugs sitting in the backlog. By focusing this metric on the customer-reported bugs, we have an excellent view of product quality AND customer responsiveness in one metric.

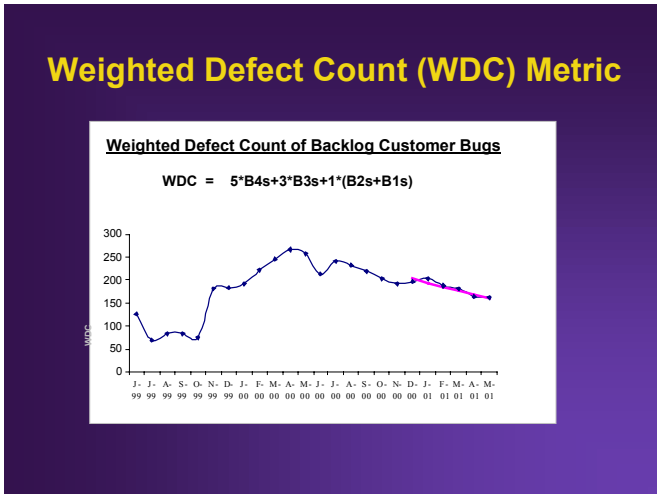


Figure 3: Weighted Defect Count (WDC) Metric for a Product

Average Age of Open Bugs and of Workarounds

Another way of viewing and measuring the backlog of bugs is to consider how long the bugs have been open. The faster we fix bugs that are found, the happier the customers will be. The Average Age of open bugs metric helps us keep the product teams aware of aging bugs. We view the average age in three ways. The first two have been discussed earlier, Customer Only and Customer + Internal.

The third average age metric measures the age of open workarounds. When a customer finds a critical bug, the product may not be of use to them. In order to get them up and running quickly, workarounds are evaluated. If a workaround is available, the information is given to the customer immediately. Work on the long-term fix to the product is later completed. The time that it takes to get a workaround defined and to the customer is tracked in the Average Age of Workarounds metric.

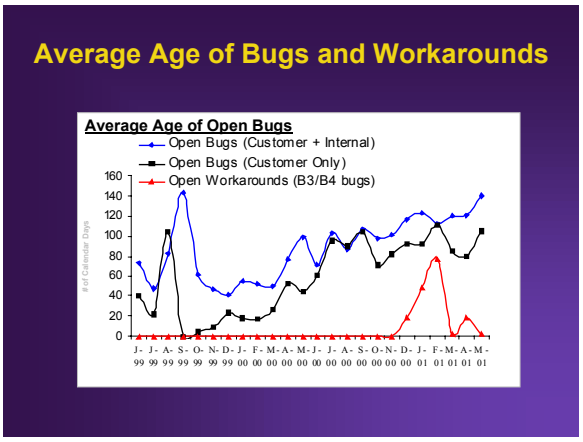


Figure 4: Average Age Metric for a Product

Support Calls

Not all customer calls are due to bugs in the software. Calls are categorized and tracked. Looking at the number of calls each month and the breakdown of those calls (Usage, Installation, Bug, Enhancement) is helpful. This is the last metric that we track.

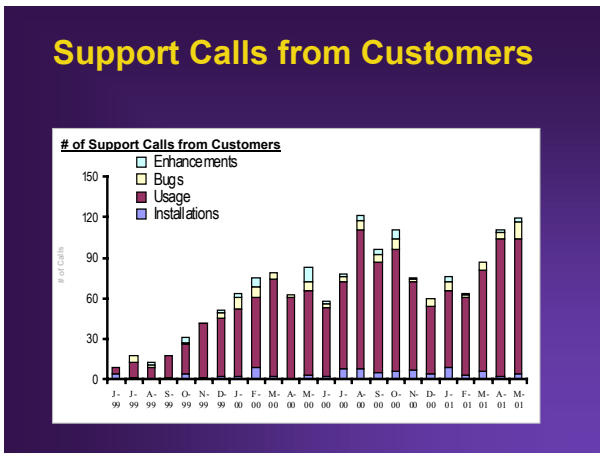


Figure 5: Support Calls Metric for a Product

THE REPORTING

Product teams, management, and sales teams all have different needs when it comes to reporting and reviewing metrics.

The product teams want to review the metrics often. They need to see the details that will lead them to improvements. They need easy access to the metrics by all team members.

To meet these needs, we report on the metrics monthly. We provide all of the metrics that are discussed above. We post the metrics on our internal company website, and send email with a direct link to the metrics to all the product teams. These metrics are used in review meetings with management and team meetings.

Executive management wants to oversee the quality efforts, but they don't want all of the details. Instead, they are interested in the bottom line and responsibility. Their need is for a management tool that allows them to quickly evaluate the status and the activities that will improve things.

To meet this need, we chose ONE key metric. The Weighted Defect Count (WDC) of customer-reported bugs provides an excellent view of product quality AND customer responsiveness in one metric (see Figure 3 above).

We prepare a quarterly report on the WDC metrics for the business reviews by senior management. The overall WDC for all of Synopsys and each Business Unit are provided. The WDC for each product is also provided. An editorial for each Business Unit allows explanation of the WDC for the Business Unit and actions that are being taken to meet improvement goals. This becomes part of the highest-level management report in our company.

Sales teams and application engineers have a different set of needs. They need information on what products their customers are happy with and which product their customers are having problems with.

To aid this group we provide a quarterly report on each of our key customers (see Figure 6). We report the WDC of the open bugs that were reported by the customer, and the number of incoming bugs for each product the customer is using. This report is provided on our web for easy access.

When reporting on any metric, it is important to show both the current and the past history of the data. One good or bad month of data does not mean that the product is "good" or "bad". To meet these needs, all of our reports provide the past 24 months of data. In addition, for the key metric (WDC) we also include the trend for the last six months. For example, December tends to be a quiet month, and the incoming rate is usually low. Using the trend smoothes out the effect of one month of unusual data.

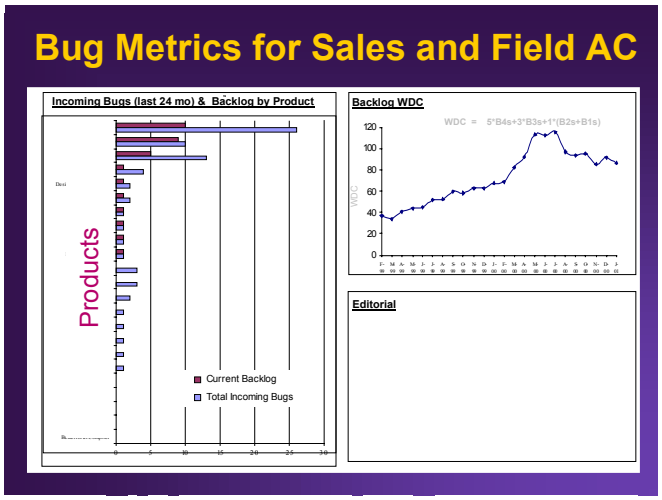


Figure 6: Quarterly Report for Sales and Field Application Engineers

THE RESULTS

This program has exceeded our expectations. Not only are the quality metrics a well used part of the company culture, executive management is focusing attention on these metrics and is supporting software process improvements that are identified.

Executive Management Awareness: Quality is at the core of the company culture, and quality initiatives are driven by top-level corporate goals.

- There is a corporate level MBO for the year: "Improve quality by establishing WDC growth rate to be negative"
- The metrics are reviewed regularly in the Executive Staff meeting.
- Each quarter, products that are showing poor trends are identified. Improvement activities are defined, and results are tracked with quarterly MBOs.

R&D awareness: Product teams use the reports and act on the trends that they see, implementing software development process improvements.

- Product teams review the metrics that are posted each month.
- We get numerous queries for more specific information to help the teams improve their quality each month.
- The metrics are used in formal Product Reviews each quarter.
- Improvement activities are documented and results are tracked quarterly.

Customer feedback: Over and over, in industry surveys, Synopsys' quality is ranked at the top of the industry.

- In the last 3 years, independent EE Times surveys of customers in our industry (the electronic design automation industry – "EDA") demonstrated that Synopsys was ranked in all categories as #1 or #2 leader in the industry.

- For two years in a row (1999, 2000) Synopsys has been awarded the Supplier of the Year award by Infineon, one of our key customers. This award is based on the quality of our products and processes.
- This year we were awarded the AMCC EDA Supplier of the Year award.

Business Growth

- In spite of fierce competition, Synopsys' business has grown to over $1 billion in new orders in the last 12 months.
- Orders have increased 32% in the two years that this program has been in place.
- The customers who have selected us for Supplier of the Year awards have also increased their orders significantly.

SUMMARY

A quality metrics program can be a great asset to your organization. Engineering, sales and the company overall can benefit from this. Consider the key elements when designing your program:

- Choose metrics that are easy to understand and use
- Provide easy to access metrics for all products on a monthly basis
- Focus on the overall quality perspective
- Report one key metric at the highest levels of management
- Utilize existing sources of data
- Measure and promote success

Any high-tech company that wants to stay on top in this highly competitive market must not only lead in technology, it must also make a continuing commitment to quality improvement.

Quality is the fundamental cornerstone of business — not an expendable luxury. By turning the usual process around, beginning with quality rather than concluding with it, the rules of the game can be changed. A foundation of quality makes everything run more smoothly and efficiently, leading to improvements on all fronts.

As Synopsys' company Chairman and CEO noted in his recent message to all employees: “In the eyes of most customers, we are viewed as the most reliable, highest quality, most technically savvy EDA company in the world. This is precisely the brand that we intend to perpetuate.”

The Influence of Process Maturity on Metrics Program Success

Dr. Peter Hantos
Xerox Corporation
701 South Aviation Blvd., MS: ESAE-376
El Segundo, CA 90245 USA
+1 310 333 9038
peter.hantos@usa.xerox.com

Dr. Emanuel R. Baker
Process Strategies, Inc.
10219 Briarwood Drive
Los Angeles, CA 90077
+1 310 278 0856
erbaker@process-strategies.com

ABSTRACT

Organizations often have difficulty establishing a metrics program. Looking at current, industry-wide practices, two approaches can be identified: (1) where the metrics program itself is the catalyst for software process improvement or (2) where the metrics program is an integral part of a model-based process improvement effort. Regardless of the approach, the problems organizations face in establishing a metrics program are similar.

The most common problems identified are:

- Absence of a proper tie-in with the organization's strategic business objectives,
- Misuse of the notion of following industry-wide practices,
- A lack of understanding of the software organization's own process maturity.

To address these issues, we demonstrate the combined use of the Balanced Scorecard and the GQM/GQM-Rx approach to align software process improvement and metrics program objectives with corporate strategic objectives. In analyzing the effects of process maturity, we rely on the experience we have gained using various process improvement models, primarily the CMM and CMMI. Each of these models describes increasing levels of process maturity. Each maturity level represents a distinct, identifiable degree of process capability, which in turn is reflected in various product attributes such as quality and cost, or process attributes, such as productivity.

Finally, we present a case study to demonstrate the development of cycle time and cycle time slip-rate metrics. Our objective is to show that even though the basic computing principles of these metrics are simple and seemingly uniform, different maturity levels in overall program/project planning and tracking yield to cycle time and cycle time slip rate metrics with highly varying effectiveness. Low organizational maturity will most likely enable only the implementation of metrics with low effectiveness, and maturity has to be raised before the metrics' effectiveness can be improved.

BIOGRAPHIES

Dr. Emanuel R. Baker is principal owner of Process Strategies, Inc., an internationally recognized software engineering consulting firm based in Los Angeles, CA and Walpole, ME. He is co-author of the book, "Software Process Quality: Management and Control", published by Marcel Dekker, Inc. in 1999. Dr. Baker has nearly thirty years of experience in software engineering, and he is a certified CBA-IPI Lead Assessor. He has held a number of management positions and was Manager of the Product Assurance Department of Logicon's Strategic and Information Systems Division. He has also played a major role in the development of quality standards for both industry and the Department of Defense. His current interest is to help organizations in their transition to CMMI. He holds an M.S.M.E degree from New York University, and M.S. and Ph.D. degrees in Education from the University of Southern California, Los Angeles.

Dr. Peter Hantos is Manager, Process Technology at Xerox, working on CMM Level-3 implementation in the Office Systems Group. Prior positions included Department Manager for SQA, SPI, System and Reliability Testing in one of the Group's Business Units. Earlier, as Principal Scientist, at the Xerox Corporate Software Engineering Center, he authored the Time To Market Software Subprocess and Software Technology Readiness processes. He started his professional career in the US as Visiting Professor at the Computer Science Department of the University of California, Santa Barbara. Dr. Hantos is a Member of ACM, Senior Member of the IEEE, and holds M.S. and Ph.D. degrees in Electrical Engineering from the Budapest Institute of Technology, Budapest, Hungary.

INTRODUCTION

Paper's Objective

Our primary objective is to help organizations to create a culture in which managers depend on and manage by facts presented via metrics, and developers willingly support their effort. Even if an organization either does not feel the need for a complex framework, or on various grounds refuses following a formal approach, we can provide important insights learned by analyzing various process improvement models and by coaching and managing numerous organizations pursuing Capability Maturity Model™ (CMM)[1]-based improvement. With respect to teams already committed to the CMM [1] framework, we believe we can help with the planning and calibration of measurement activities.

Metrics Programs

Most software practitioners view Metrics Programs as a means to collect measurement data and analyze Software Metrics[2]. Historically, the basic concepts of Software Metrics were set forth by two IEEE Standards: IEEE Standard 1045 Standard for Software Productivity Measurement, and IEEE Standard 982.1 Standard Dictionary of Measures to Produce Reliable Software. These measures are grouped in six categories: Size, Defects, Effort, Duration, Cost, and Customer Satisfaction (See also [2] for more details). The IEEE 982.1 and 1045 standards were published in the 1988-92 timeframe and, while the measures proposed in those standards are still valid, we feel that a more up-to-date definition is needed for Metrics Programs.

Metrics Programs play an important role in the continuous improvement of software development.
Metrics Programs are positioned in two main ways in the context of process improvement efforts:
(1) The Metrics Program is the catalyst of the process improvement effort
(2) The Metrics Program is an integral part of a model-based process improvement effort.

In the first case, specific improvement objectives and strategies are chosen, and the role of the Metrics Program is to provide the measures needed to gain control over the targeted variables. In the second case, the improvement objectives and strategies are part of the model, and the role of the Metrics Program is to determine how well these strategies are succeeding. Some examples of these models include CMM, Capability Maturity Model Integration SM (CMMI)[3] [3], SPICE [4], ISO-15504 [5], ISO 9000 [6], and Six Sigma [7]. Different models require different approaches to determine the necessary measures. For example, the CMM provides a framework to prioritize elements of Process Maturity[4], and describes the primary measures to determine/assess Organizational Maturity[5]. On the other hand, Six Sigma is highly data driven and is aimed at accelerating improvements. For this reason, it is not

---

[1] ***The Capability Maturity Model (CMM)*** is a trademark of the Software Engineering Institute (SEI) at Carnegie Mellon University, Pittsburgh, Pennsylvania.

[2] ***Metrics and Measures*** – We will use Kennett and Baker's book [1] as a source to define these terms. A Measure is a "number that assigns values on a scale. Examples may include number of errors, lines of code, or work effort." A Metric is a "combination of two or more measures or attributes" that enables the computation of useful data for evaluating performance, quality, stability, or other parameters concerning the process or product in question.

[3] ***The Capability Maturity Model Integration (CMMI)*** is a service mark of the SEI

[4] ***Process Maturity*** – The extent to which a specific process is explicitly defined, managed, measured, controlled, effective and efficient, where "Process" refers to Key Process Areas in the CMM, or Process Areas in the CMMI.

[5] ***Organizational Maturity*** – Reflects a composite of the maturity of processes used by the organization. Highly mature organizations use highly mature processes. The term sometimes used interchangeably with Process Maturity. In that case the word "Process" is used in a different, broader sense, and refers to the organization's overall "Software Development Process", which in reality is not a single entity, but a collection of processes focusing on development and management practices such as software product engineering or configuration management.

recommended for organizations beginning the improvement journey. The example of Six Sigma also shows that Organizational Maturity has a fundamental role in selecting the improvement framework as well.

In summary, the expectation of a Metrics Program is not simply to provide a list of recommended measures, but also to ensure that the selected measures closely support the primary improvement strategy.

ESTABLISHING A METRICS PROGRAM

Bad Practice – Copying Metrics from Other Organizations
Organizations often flounder in establishing a metrics program. In the absence of an organized approach, they often try to establish one by attempting to use metrics that other organizations use. This often leads to failure, since metrics appropriated from other organizations are rarely suitable for the organization seeking to establish a metrics program. Contrary to popular belief, plagiarism is NOT the fastest path to productivity. A methodical analysis of the two organizations' culture, history, operational and business context would have to take place before any metric could be adopted. Also, to be used successfully, metrics must be focused on the specific information needed, adapted for the process involved, and customized for the environment from which the individual measures will be derived. Consequently, what works well for one organization, may not work well for another, unless both organizations have nearly identical information needs, processes, and environments.

Recommended Practice - Using GQM/GQM-R$_X$
More enlightened companies may use an approach like Basili and Weiss' GQM (Goal-Question-Metric) model [8], because it is viewed as more effective, and produces more useful metrics than other, unstructured, heuristic approaches. Used properly, that is a correct assumption. Briefly, GQM asserts that the starting point for establishing a metric is to state a goal for making a measurement. A goal might be, for example, to reduce time to market. The next step in GQM is to ask questions focused on determining the information needed. Examples might be, "What is my current time to market from project conception to delivery of the first unit?" and "How long does it take to get a project approved, once a concept has been defined?" A metric that could be derived, based on these questions, is the length of each phase of the development cycle. Another metric might focus on determining the average delay in completing project major milestones. Readers are encouraged to read Baker and Hantos' paper on GQM-R$_X$, a practical extension of Basili's method [9].

Implementation Planning – Metrics Prioritization
Assuming that the organization established a set of metrics, using either a heuristic approach or the GQM model, it still has to decide on the order of efficient and effective implementation. The conventional methods to develop an implementation priority list are based on different management philosophies. They include:

A/ Order the metric implementation tasks by difficulty, in ascending order. The philosophy is to show results early, and gain more, early support to the program.

B/ Order the tasks by difficulty, but in descending order. The idea is to "ride" and exploit the initial momentum, and get the most difficult tasks out of the way.

C/ Order the tasks in descending order by ROI (Return On Investment). The philosophy behind this approach is to "Get the Most Bang for the Buck" right away.

It is important to understand that while one can easily make a compelling case for any of these strategies, the common deficiency of these approaches is that they treat the measurements and their implementation efforts independently. In reality there is a, not so hidden, sophisticated dependency amongst the various measurements. Ignorance of these dependencies and the limitations surrounding the measurement

process can lead to failing or ineffective metrics programs. These limitations fall into two categories: Definitional Constraints, and Process Maturity constraints.

Definitional Constraints
Definitional constraints pertain to how the particular measures are defined. For example to track Defect Density (defects per 1000 lines of code), it is important to precisely define the following:

- What constitutes a line of source code
- How that definition may change as a function of the source language used
- What constitutes a defect
- What comprises the categories of defects
- What point in time is the measurement taken
- The extent to which severity of defect will be included in the measurement, for example, will there be separate determinations at each level of severity?

It is difficult, if not impossible, to give general advice on how to deal with definitional constraints. In the following paragraphs we will focus only on Process Maturity constraints. We have to note though that there are numerous other, primarily cultural factors, which might cause the failure of a Metrics Program, or in fact the failure of any kind of process improvement effort. In the specific case of metrics introduction, addressing these cultural issues is equally important to addressing definitional and maturity constraints, but a detailed discussion of these cultural factors is beyond the scope of this paper.

Process Maturity Constraints
These constraints pertain to the maturity of the organization's development and maintenance processes. An example of a process maturity constraint related to the metric discussed above would be the method by which Defect Density is defined and normalized across the projects. Without consistent definitions, it is difficult to have meaningful measures. After all, if defects per line of code mean one thing on one project, but something different on another, we don't have a useful measure of quality on the organizational level. Once measures are defined uniformly, the infrastructure for making and supporting the measures must exist. The ability to accomplish these objectives is a function of the maturity of the processes used by the organization.

BALANCED SCORECARD

Many organizations pursue software process improvement activities without considering strategic business objectives and how SPI (Software Process Improvement) goals support these objectives. Such organizations tend to have separate corporate programs for process improvement, customer satisfaction, and financial success, and typically have difficulty in communicating the strategic importance of SPI. Even though they systematically use the GQM approach to determine the necessary measures, they might end up with partial success in their efforts and often abandon some or all of their SPI initiatives. The Balanced Scorecard framework was developed at the Harvard Business School [10], and it is an effective way to demonstrate the cause and effect relationship of software process improvement and the critical value propositions needed for customer, shareholder and employee satisfaction. Figure 1 on the next page gives an idea of how the Balanced Scorecard facilitates the translation of Organizational Vision to operational terms, such as objectives, measures, targets, and initiatives. The Balanced Scorecard also demonstrates how long-term strategic objectives can be linked to short-term actions.

Perspective	Relation to Vision	Objectives	Measures	Targets	Initiatives
Financial	"To succeed financially, how should we appear to our shareholders?"				
Customer	"To achieve our vision, how should we appear to our customers?"				
Internal Business Processes	"To satisfy our shareholders and customers, what business processes must we excel at?"				
Learning and Growth	"To achieve our vision, how will we sustain our ability to change and improve?"				

Figure 1. Four Perspectives to Translate Vision and Strategy

Figure 2 shows the specific details of how the Balanced Scorecard is developed step-by-step, and also shows the cause and effect relationships between the Balanced Scorecard elements.

Perspective	Strategic Objectives	Strategic Measurements' Core Outcomes
Financial	Consistently meet or exceed shareholder expectations for revenue growth	❑ Pipeline[6] flow is well managed and uninterrupted
Customer	Consistently meet or exceed customer expectations for on-time delivery	❑ Customer feedback indicates that expectation with respect to on-time delivery was fulfilled
Internal Business Processes	Projects achieve predictable results for schedule within known range of the organization's defined process capability	❑ Products' overall development cycle times are on par or better than industry segment's benchmark ❑ Delivery slip by teams in the phase-gates[7] is less than or equal to the corporate benchmark
Learning and Growth	Investment in people enables achievement of customer and shareholder satisfaction goals	❑ Software engineers demonstrate the necessary understanding of software development life cycle models

Figure 2. Cause and Effect Relationships in the Balanced Scorecard

[6] ***Pipeline*** – The collection of all technology projects, products and value chain capabilities in development. Pipeline management relates to the growth of the orders backlog, effectiveness and efficiency of deliveries to the customer, and timeliness of payments, to mention a few.

[7] ***Phase-Gates*** – Product Development in general is managed in phases, marked by a set of milestones called Phase-Gates, around which projects can be planned, organized, monitored and controlled.

CASE STUDY

In the case study we demonstrate the use of the Balanced Scorecard (Figure 2) to develop delivery cycle time and slip rate metrics. Specifically, we will illustrate the delivery slip approach as it relates to the Strategic Measurements' Core Outcomes for the Internal Business Processes category. The first step is to determine the Product Development Phase Structure, since the Balanced Scorecard implies the use of phases and phase gate reviews as one of the Internal Business Processes.

Sample Product Development Phase Structure

There are different ways to determine how the phase structure will look. For example, in start-up companies the phases are determined by the way they are funded, so the purpose of the activities around the milestones is to secure funding for the next round. In more conventional settings, the purpose of activities around the milestones is basically risk management. A typical review consists of the following steps: Self-assessment by the development team, a review by an outside assessment team, risk mitigation planning by the development team based on the assessment team's feedback, and finally a decision by a decision authority. The decision authority decides (1) if the development can go forward, or (2) development, and consequently the whole program should be stopped, or (3) continuation will depend on the successful completion of certain, pre-determined corrective actions.

In our case study we chose the following, fairly typical 5-step product development phase structure, with planned phase-gate reviews at the end of the phases (Figure 3).

Product Concept	Product Definition	Product Design	Product Demonstration	Delivery & Maintenance
Phase1	*Phase2*	*Phase3*	*Phase4*	*Phase5*

Figure 3. Five-Phase Product Development Process

Cycle Time

Product Development Cycle Time would have to be estimated by program management. The end of the 4th phase, the launch date, would have to be synchronized with market, customer and corporate expectations.

The definitions of product cycle-time and cycle time slip rate (in days) are fairly straightforward:

❶ Planned Product Development Cycle Time = Planned Launch Date – Planned Concept Phase Start + 1

❷ Planned Phase N Duration = Planned Phase N End Date – Planned Phase N Start Date + 1

❸ Actual Phase N Duration = Actual Phase N End Date – Actual Phase N Start Date + 1

❹ Phase N Slip Duration = Actual Phase N End Date – Planned Phase N End Date

❺ Product Slip Duration = Actual Launch Date – Planned Launch Date $\Rightarrow \sum$ Phase N Slip Duration

⊗ Phase N Cycle Time Slip Rate = Phase N Slip Duration ÷ Planned Phase N Duration

❼ Product Cycle Time Slip Rate = Product Slip Duration ÷ Planned Product Development Cycle Time

Tracking Cycle Times and Cycle Time Slip Rate for products certainly satisfies the original requirement we saw in the Balanced Scorecard's Internal Business Process category and will support Product Pipeline management requirements as well. Please note, that for proper metrics implementation Cycle Time must be defined in clear, unambiguous terms. Particularly in global and multi-national companies workweek differences, national and religious holidays, national labor policies, time zone/date issues, etc. can pose a serious challenge for the Metrics Implementer.

Cycle Time Slip Rate

Cycle Time Slip Rate is a derivative of the Cycle Time metric, created for the purpose of providing more diagnostics help for program management. For illustrative purposes we show a data table for four products in Figure 4.

Phases / Products	Product Concept	Product Definition	Product Design	Product Demonstration	Delivery & Maintenance
	Phase1	*Phase2*	*Phase3*	*Phase4*	*Phase5*
Product1	8%	7%	10%	8%	n/a
Product2	1%	3%	9%	2%	n/a
Product3	2%	6%	12%	20%	n/a
Product4	1%	2%	10%	6%	n/a

Figure 4. Phase Slip Data for Products in a Selected Effort Category

When managing a larger portfolio with many different products in the pipeline, products first have to be grouped into effort categories, such as Large-Medium-Small. This grouping is needed to normalize the slip rate data, and make possible the sensible comparison of data elements associated with different products. Due to economies of scale, it is expected that while smaller projects will have shorter cycle times, they might have higher slip rates. A considerably short slip of few days will result in high slip rate when it is compared to the project's short cycle time. Unfortunately, discussing implementation details of effort categorization falls out of the scope of this paper.

The first application of this table can be to identify Product Team related problems. For example, in the case of Product1, the slip rate in all phases is relatively higher than the slip rates of other products in this effort category. Looking at Product3 data, we see a disturbing trend of a radically growing slip rate, even though one would assume that the organization made attempts earlier to control and reduce slip rate and to improve the planning process.

The second possible application of this table is to have a high level assessment of the maturity and performance of the used delivery process. Studying Phase3 data, we see that slips in this phase gate are consistently high for all products in this effort category. This fact suggests that the problems most likely lie with the phase-gate review process itself, and not with the specifics of the organizations delivering the various products.

Product Development Cycle Time vs. Project Cycle Time

To demonstrate these issues, let's consider the following product: During the development of the product concept, a very high level architecture was developed. The system will consist of three main subsystems, a "Server", a "Client" and a "GUI" (Graphical User Interface). The organizational structure will reflect this product structure, and three teams are chartered with development, working parallel. Program management would like to see the development in the following way (Figure 5).

Product Dev. Phases / Teams	Product Concept	Product Definition	Product Design	Product Demonstration	Delivery & Maintenance
	Phase1	***Phase2***	***Phase3***	***Phase4***	***Phase5***
Server Team (ST)	ST phase1	ST phase2	ST phase3	ST phase4	ST phase5
Client Team (CT)	CT phase1	CT phase2	CT phase3	CT phase4	CT phase5
GUI Team (GT)	GUI phase1	GUI phase2	GUI phase3	GUI phase4	GUI phase5

Figure 5. Idealized Breakdown of the Process By Teams

In this idealized view, development is progressing absolutely parallel in all three teams, and the phase gates are nicely lined up. It also seems that measuring team level cycle time and cycle slip-rate is simple, as is the roll-up of team-level cycle time and slip rate data to the product/program level. If we realize that using an overly idealized view of the development process is in fact a manifestation of low organizational maturity, then it becomes clear that a more "capable" process needs to be used.

The following iteration of the phase structure shows a better understanding of how in fact the product development program will be managed (Figure 6):

Product Dev. Phases / Teams	Product Concept	Product Definition	Product Design	Product Demonstration	Delivery & Maintenance
	Phase1	***Phase2***	***Phase3***	***Phase4***	***Phase5***
Program Mgmt. (PM)	PM phase1	PM phase2	PM phase3	PM phase4	Highly limited
Server Team (ST)	*participates*	ST phase2	ST phase3	ST phase4	ST phase5
Client Team (CT)	*participates*	CT phase2	CT phase3	CT phase4	CT phase5
GUI Team (GT)	*participates*	GUI phase2	GUI phase3	GUI phase4	GUI phase5

Figure 6. A More Realistic Phase Structure

The Program Management Team's responsibility is to develop the initial product concept and the overall management processes for the development phases. The Development Teams participate via representatives (domain experts) only in the first phase. While phases 2-4 are straightforward, phase 5 is different. After launch the scope of program management functions is highly limited, and the development teams might be re-directed to other projects. Nevertheless, the people who are experts in various areas of the product will have to keep the responsibility of fixing customer reported defects.

While this is a better phase structure than the previous one, it still has a major flaw: it assumes that there is no difference between the teams' maturity levels, and there is no difference between developing server software vs. client software and a GUI either. People who share the view of "software is software, and it does not matter what it does" will be the first ones to violate one of the basic tenets of process improvement: "One process does not fit all". Here is where the relevance of the specific "Learning and Growing" objective from Figure 2 becomes clear, since the proper coordination of these three teams require an up-front decision about the life-cycle models the development teams will use.

Integrating Project Life Cycles into the Product Development Life Cycle

To make this final iteration, the Program Management Team – in consultation with the Project Managers – decides which software development life cycle models the teams will use. The GUI team feels that a simple waterfall structure will suffice (for a description of the waterfall, see page 136 of Steve McConnell's book [11]. Neither the development of the server or the client software seems to be so straightforward, so the Program Management Team's choice is to use RUP (Rational Unified Process). RUP is an elaboration-based, iterative-incremental software development model (for further details see Phillipe Kruchten's book [12].) If these project life cycle models are also taken into consideration, then the engineering view of the Product Development Process becomes quite complex. Figure 7 shows that not only the number of phases is different, but the nature of the work is different as well, making it very difficult to come up with the desired, homogeneous, top level management view of the product development phases. The diagram also shows the fact that there is a difference in the phase durations as well. Overall, the Client subsystem is less complex than the Server subsystem, so even though they use the same life-cycle model, the duration of the phases will be different. Comparing the waterfall and RUP we must note that in waterfall processes the front-end work tends to be less than the analysis and design activities of a RUP Inception phase. With respect to the back-end, system tests (and fixing of the bugs…) in Waterfall tend to be longer than the activities of the Transition phase in RUP.

Server Team *(RUP)*	Inception	Elaboration		Construction	Transition
Client Team *(RUP)*	Inception	Elaboration	Construction	Transition	
GUI Team *(Waterfall)*	Requirements	High-Level Design	Detailed Design	Code	Integration/Test

Figure 7. Project Life Cycle Phases

Proper alignment of these processes requires an even higher capability process, using state-of-the art approaches, for example, Anchor Points. It is beyond the scope of this paper to discuss the approach, so readers are invited to study Barry Boehm's seminal article on the subject [13].

In reviewing the case study, we have to conclude that the following factors contribute to the complexity of the definition of the metrics:

(1) The Critical Path needs to be computed to determine the total product development cycle-time
(2) Due to the phase differences of the different life-cycle models, product phase objectives become ambiguous, and the product-level phase gate reviews are not effective enough to identify critical risks to the program
(3) Project Cycle Time and Slip Rate Data cannot be simply rolled up to compute the product development program's cycle time and cycle time slip rate.

MEASUREMENTS AND MATURITY

In describing this aspect of our approach, and how it applies to the case study, we rely on the experience we've gained using various process improvement models. Each of these models describes increasing levels of maturity. Each level represents a distinct, identifiable degree of process capability, which in turn is reflected in product attributes, such as cost and quality, or process attributes, such as productivity. By extension, the lower the organization's process maturity, the greater the *inability* to acquire and utilize the measures needed to evaluate process adequacy. For a brief description of the various maturity levels please see the first two columns of Figure 8.

At the lowest level of maturity, processes are not implemented in an organized way, if at all. Consequently, the opportunity to make meaningful measures does not exist. Any measures that can be collected exist by happenstance and are not representative of the organization. If we hypothesize that at the next level of maturity, some processes are *performed*, but are not codified and managed, the organization has achieved a greater level of capability, and it may be possible to obtain some rudimentary measures. These measures may be based on the existence or non-existence of specific work products. For example, if schedules are created, but are not updated to identify actual slips by clearly identifying re-plans, then the ability to measure slip rate is compromised.

The next logical step of maturity is where the processes are not only performed, but they are *managed* as well. That means that the processes are planned, performed, monitored, and controlled, and the organization, as a whole, requires this level of management. Different projects or different organizational groups may perform the same processes differently, but within any *given project or group*, there will be consistency. Some level of capability to improve processes exists, but may be hindered by the fact that the required information may not be reported the same way on each project.

Progressing to the next level of maturity, there would be recognition that there are advantages to standardizing processes across the organization. The processes may be standardized by application domain or on any other basis in which standardization makes sense. Individual projects might tailor the *defined* process to accommodate the unique characteristics of that project, but in general, there would be consistency in the way that processes are performed, as illustrated in Figure 7. Accordingly, a greater ability exists at that level of maturity to acquire and utilize measures to characterize the adequacy of their processes and the quality of the product across projects or organizational groups.

Once that level of maturity exists, the capability exists to begin implementing statistical process control and other numerical methods to *quantitatively manage* individual processes. Corrective action can be implemented, when necessary, to bring an errant process back under control. Progressing to the next level of maturity, the highest level, the organization can begin using statistical process control and other numerical methods to *optimize* the processes through quantitative evaluation of technology or methodological changes to the processes. Furthermore, the organization would also have the capability by applying quantitative means to prevent defects from occurring.

How the Method is Applied

Figure 8 illustrates how the slip rate related metrics evolve at the various levels of organizational maturity.

Level [8]	Characteristics	Metric	Measures	Comments
0 Incomplete	No meaningful process exists	None	None	No ability exists to collect any consistent data
1 Performed	Scheduling and tracking is performed, but the process is not codified	Slip Rate	Actual project end date compared to original planned end date	
			Overall phase end date compared to original planned end date	May not recognize need to track separately for server, GUI, client
2 Managed	Scheduling and tracking is planned, performed, monitored, and controlled, but varies from project to project	Slip Rate	Actual project end date compared to original planned end date	Some projects may show re-plan history
			Actual phase end date compared to original planned end date	Many projects may have data available separately for server, GUI, client
3 Defined	Standardized scheduling and tracking requirements across organization	Slip Rate	Actual project end data compared to original planned end date	Need is recognized for uniqueness in processes/ phase definitions for GUI, client, server
			Actual phase end data compared to original planned end date	Detailed, consistent measures of slip for GUI, client, server
		Slip Rate Change	Trend of slip rate changes	A measure of project management effectiveness
4 Quantitatively Managed	Process quantitatively managed	Slip Rate	See Level 3	With numerical methods, process corrective action can be implemented early if slip or slip rate change are becoming excessive in the organization
5 Optimizing	Process quantitatively managed	Slip Rate	See Level 3	With numerical methods, process improvement can be implemented.

Figure 8. Applying the CMMI Capability Levels to the Case Study

[8] Level naming conventions are adopted from the Software Engineering Institute's description of the continuous representation of the CMMI. The continuous representation calls the maturity levels Capability Levels.

CONCLUSIONS

Process maturity is an essential element in implementing successful metrics programs. By knowing the level of maturity of the various processes, we know the limitations on the kinds of numerical or even qualitative data that we can extract to gain knowledge about the current status of our processes.

As described in the case study, at the very lowest level, we have no coherent project management process whatsoever, and project management is performed in a very ad-hoc manner, if at all. We might suspect that we are not managing our projects very effectively (because common knowledge indicates that many projects slip), but we would have no way to quantify the slip rate in any coherent manner. If we wanted to improve the process, we would have no way of determining if we had succeeded in implementing effective improvements.

At the next level, Level 1, we can implement some gross measures of slip rate, but nothing that can be used to improve internal business processes. It's not until Level 2 that we begin to gain some understanding of how well projects are being managed; however, the process definitions (and corresponding measures) are still too immature to give us the complete understanding needed.

At Level 3, we have the processes adequately defined such that the measures we select give us the understanding needed of the project management practices currently in use. Level 4 enables us to start implementing numerical methods for process corrective action, if the data indicates such a need. For example, histograms can help to determine if the GUI, client, or server projects were the projects that were causing the most problems. Continuing to Level 5, our data enables us to optimize our processes and techniques to help us find ways of reducing slip rates even further than what the current nominal level is.

By applying the maturity concepts of established process improvement frameworks, we demonstrated that an organization's process maturity has a critical impact on the effectiveness of the organization's Metrics Program. To "jump start" the difficult task of Metrics Program initiation, we introduced two methodologies. To ensure the alignment of the Measurement Program with corporate strategic objectives, we recommended the use of the Balanced Scorecard method. After the strategic measures were established in conjunction with the strategic objectives, the GQM/GQM-R_X approach can be used to derive the necessary operational metrics and measures. Finally, even if no formal process improvement model is used, a very thorough evaluation of the organization's process maturity is needed to verify the viability and expected effectiveness of the planned metrics.

REFERENCES

[1] M.C. Paulk, B. Curtis, M. B. Chrissis, C. V. Weber, Capability Maturity Model for Software, Version 1.1, Technical Report CMU/SEI-93-TR-24, Software Engineering Institute, Carnegie Mellon University, 1993

[2] R.S. Kennett, and E.R. Baker, Software Process Quality Management and Control, Marcel Dekker, 1999

[3] D.M. Ahern, R. Turner, and A. Clouse, CMMI(SM) Distilled: A Practical Introduction to Integrated Process Improvement, Addison Wesley, 2001

[4] K.E. Emam (editor) at al, Spice: The Theory and Practice of Software Process Improvement and Capability Determination, IEEE Computer Society, 1998

[5] ISO/IEC TR 15504-1:1998 Part 1: Concepts and Introductory Guide, ISO Store, 1998

[6] J.J. Tsiakals, J. Joseph, C.A. Cianfrani, and J.E. West, ISO 9001:2000 Explained, Quality Press, 2001

[7] F.W. Breyfogle III, Implementing Six Sigma: Smarter Solutions Using Statistical Methods, Second Edition, John Wiley and Sons, 1999

[8] V.R. Basili, and D.M. Weiss, "A Methodology for Collecting Valid Software Engineering Data", IEEE Transactions on Software Engineering, pp. 728-738, November 1984

[9] E.R. Baker and P. Hantos, "GQM-R$_X$ – A Prescription to Prevent Metrics Headaches", Fifth Multi-Conference on Systemics, Cybernetics and Informatics, Orlando, Florida, USA, July 22 - 25, 2001.

[10] R.S. Kaplan, and D.P. Norton, "Using the Balanced Scorecard as a Strategic Management System", Harvard Business Review, pp. 75-85, January – February 1996

[11] S. McConnell, Rapid Development – Taming Wild Schedules, Microsoft Press, 1996

[12] P. Kruchten, The Rational Unified Process, An Introduction, Second Edition, Addison Wesley, 2000

[13] B. Boehm, "Anchoring the Software Process", IEEE Software, July 1996

Implementing Software Quality Release Criteria

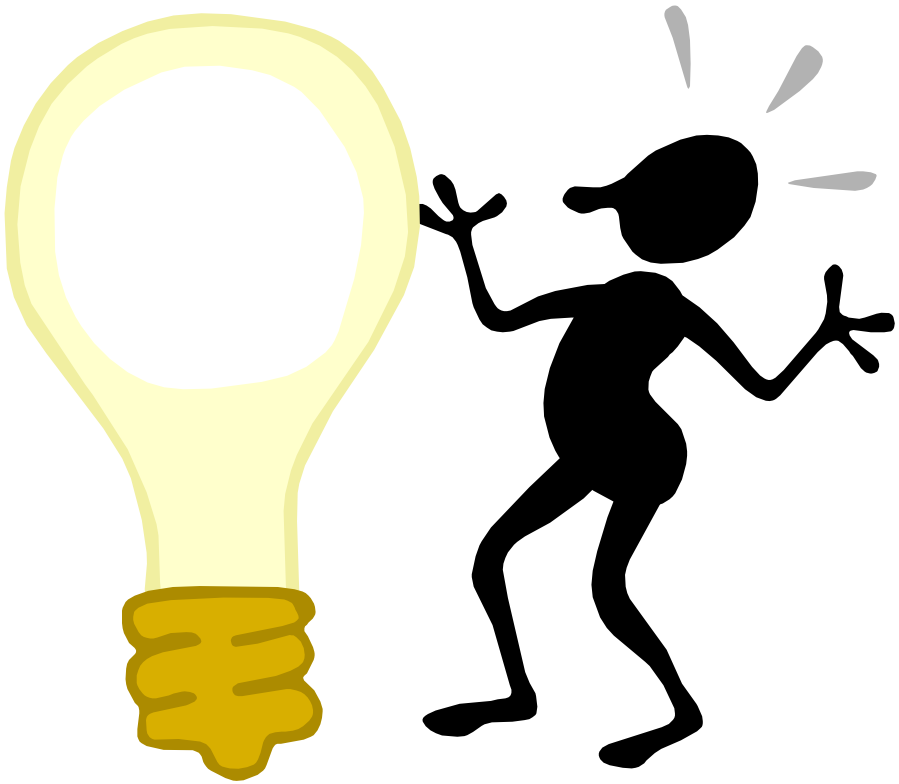
Or, How we solved a corporate-wide software quality problem without going completely bonkers

Manny Gatlin

Mike Anderson

Greg Hannon

Concept



"Daring ideas are like chessmen moved forward; they may be beaten, but they may start a winning game."

- Johann Wolfgang von Goethe

Introduction

- What if…
  - Software quality wasn't a surprise!
  - Expected quality was planned and the quality activities were quantified!
  - Quality goals were defined and measurable!

A Software Quality Framework

- What if you had a Software Quality Framework?
  - Software Quality Release Criteria
    - A quality framework for software
    - Establishes minimum standards for quality
    - Provides an extensible framework for software quality

What to Measure | How to Measure | When to Measure

Software Quality Release Criteria

- Functionality
- Usability
- Reliability
- Performance
- Service
- Compliance
- Stability
- Installability
- Maintainability
- Extensibility
- Reproducibility
- Compatibility
- Documentation

Standard

- Functional Testing
- Defect Data
- Product Reproducibility
- Install Testing
- Customer Documentation
- Compatibility Testing
- Legal Compliance
- Continuous Operation

Extended

- Regression Testing
- Customer Acceptance Verification
- Performance Measurement
- Standards Compliance

Custom

- Custom 1

SW QRC Scope

- Targeted groups
  - Applies to all Intel software products and components released externally
- Actual Use
  - Product development teams
    - Deployment path teams
    - Other teams that have seen the SW QRC have grabbed it and run with it
  - Fab software teams have started using the SW QRC

SW QRC Bill of Materials

- Content
  - Quality scorecard template
  - Milestone checklist templates (Alpha, Beta, Gold)
  - Reference Guide
  - Training (Introduction and Workshop)
- Infrastructure
  - Templates implemented as MS Excel worksheets
  - Internal website created with links to all collateral
- Maturity
  - Targeted at “typical” software product development teams
  - Criteria selected based on actual team practices

SW QRC Bill of Materials (con't)

SW Quality Release Criteria Scorecard
Product: Widgit 2.1
Owner: B. Lee Gured
Last Update: WW 30 2000
Current Software Milestone Objective: Alpha RC 1.0

Risk Legend:
1: Showstopper
2: High Risk
3: Medium Risk
4: Low Risk
5: Negligible Risk

ID	QUALITY ASSESSMENT	DESCRIPTION	RISK CURRENT	RISK PREVIOUS	OWNER	NOTES/COMMENTS
	Summary	Overall product health.				
SWFT	Functional Testing	Evaluation of the testing of features and functions to determine whether the testing is adequate for ensuring quality.				
SWDD	Defect Data	Evaluation of product defect information/verification of correct product operation.	4	3	Smith	Defect data now being tracked weekly
SWPR	Product Reproducibility	Evaluation of the ability to archive, accurately and consistently reproduce/rebuild the software product, and ensure the consistency of the product package with the content expectations for the released product package.				
SWIT	Install Testing	Evaluation of install/uninstall testing to verify correct install and uninstall of product.				
SWCD	Customer Documentation	Evaluation of product user documentation for accuracy and consistency with product operation.				
SWCT	Compatibility Testing	Evaluation of the product under test to perform its required functions while sharing the same environment with one or more systems or components. This includes the categories of binary sharing, data sharing, HW/OS/Environments, and backward compatibility.				
SWLC	Legal Compliance	Evaluation of product compliance with legal requirements.				

Quality Scorecard, used to roll up the results from the Checklists to help communicate the risk to software product quality.

SW Quality Release Criteria - Alpha Checklist
Product: Widgit 2.1
Owner: B. Lee Gured
Last Update: WW 39 2000

ID	QUALITY ASSESSMENT	CRITERIA	ALPHA TARGET	Date Done	OWNER	STATUS/COMMENTS
SWPR1	Product Reproducibility	The release candidate can be demonstrated to be reproducible.	Alpha Release is reproducible on server hosting PDT's build environment.	WW 37 '00	Smith	
SWPR2	Product Reproducibility	The Software Bill of Materials (BOM) and product Stock Keeping Units (SKUs) are documented, current, and have been reviewed by the PDT and other stakeholders.	Software product BOM's and SKU's are defined	WW 39 '00	Smith	
SWPR3	Product Reproducibility	The product Package Checklist is documented, current and has been reviewed by the PDT and other stakeholders	Package Checklist is defined		Smith	
SWPR4	Product Reproducibility	Release image and media have been scanned for viruses.	Release candidate is scanned before release with 0 viruses detected		Jones	
SWPR5	Product Reproducibility	The software source and build environment has been successfully archived according to plan and is verified to be retrievable and rebuildable into a product binary that matches the product approved for release	Archive plan exists and has been reviewed		Davis	

α

SWDD3	Defect Data	Cumulative submitted defects trending flat.	Trending flat for 3 consecutive weeks prior to release		Smith	
SWDD4	Defect Data	All appropriate open defects at release decision date reviewed by CCB or equivalent.	Review by CCB complete by WW 29 '99	WW 29 '99	Smith	
SWPR1	Product Reproducibility	The release candidate can be demonstrated to be reproducible.				

ß

SWDD4	Defect Data	All appropriate open defects at release decision date reviewed by CCB or equivalent.	Review by CCB complete by WW 29 '99	WW 29 '99	Smith	
SWPR1	Product Reproducibility	The release candidate can be demonstrated to be reproducible.				

Au

Milestone Checklists, used at and prior to each development milestone to communicate product health using the SW QRC.

Creation

"If we want to solve a problem that we have never solved before, we must leave the door to the unknown ajar."

- Richard P. Feynman

How Everything Started

- The Problem
  - No common framework for software quality existed at Intel
  - Some isolated pockets of excellence
  - Software was shipping with serious problems
- The Request
  - Intel's Corporate Quality Network wanted a software quality solution
    - Must integrate with existing hardware quality processes and corporate guidelines
    - Must address common software problems

How We Approached the Problem

- The Goal
  - Support data-driven product release decisions by management
- The Approach: Provide
  - A framework for software quality planning and monitoring
  - A consistent method for assessing software health prior to ship
  - A means to make software quality predictable

Initial Reactions to the Concept

Development Guidelines

- Don’t re-invent the wheel
  - Build on industry practices
- Don’t forget tribal practices
  - Leverage Best Known Methods from Intel Product Development Teams
  - Why do initiatives fail? Learn from the past
- Use the language of the tribe
  - Must translate into Intel terminology
- Integrate industry learning with tribal knowledge
  - Result: *New **AND** Improved practices!*

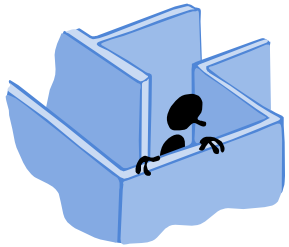
A Brief History of the Program

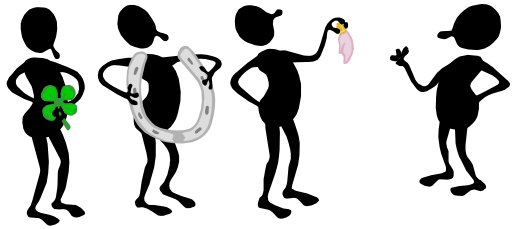
Pre-1999

Several unsuccessful attempts to address software quality issues across Intel

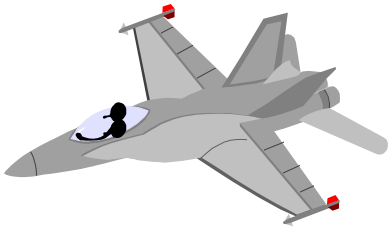
1999

Expert Interviews, Discovery, and Concept Development





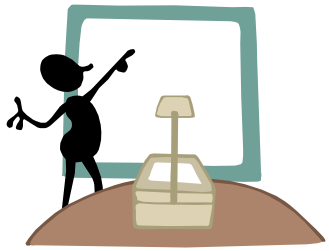
2000



Pilot Program, Training Development, and Phased Deployment

2001

Continued Deployment, Training, and Refinement



Cost of Development

Early 1999

1 engineer

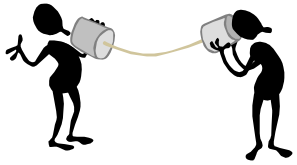
- Drove initial concept
- Collected Best Known Methods and expert input

Late 1999 – Early 2000

3 engineers

- Completed the initial revision of the collaterals
- Drove a pilot deployment to prove the concepts and usability

Mid 2000 – Early 2001



2 engineers

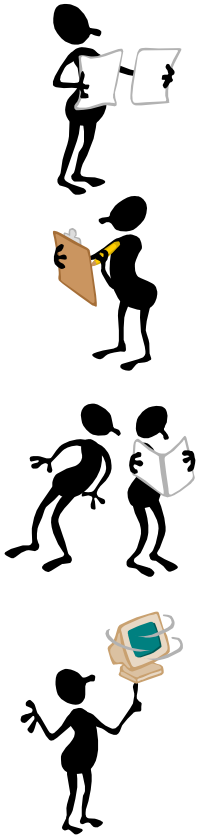
- Refinement of the training
- Phased deployment started

Present

1 engineer

- Refinement of the training
- Global deployment started

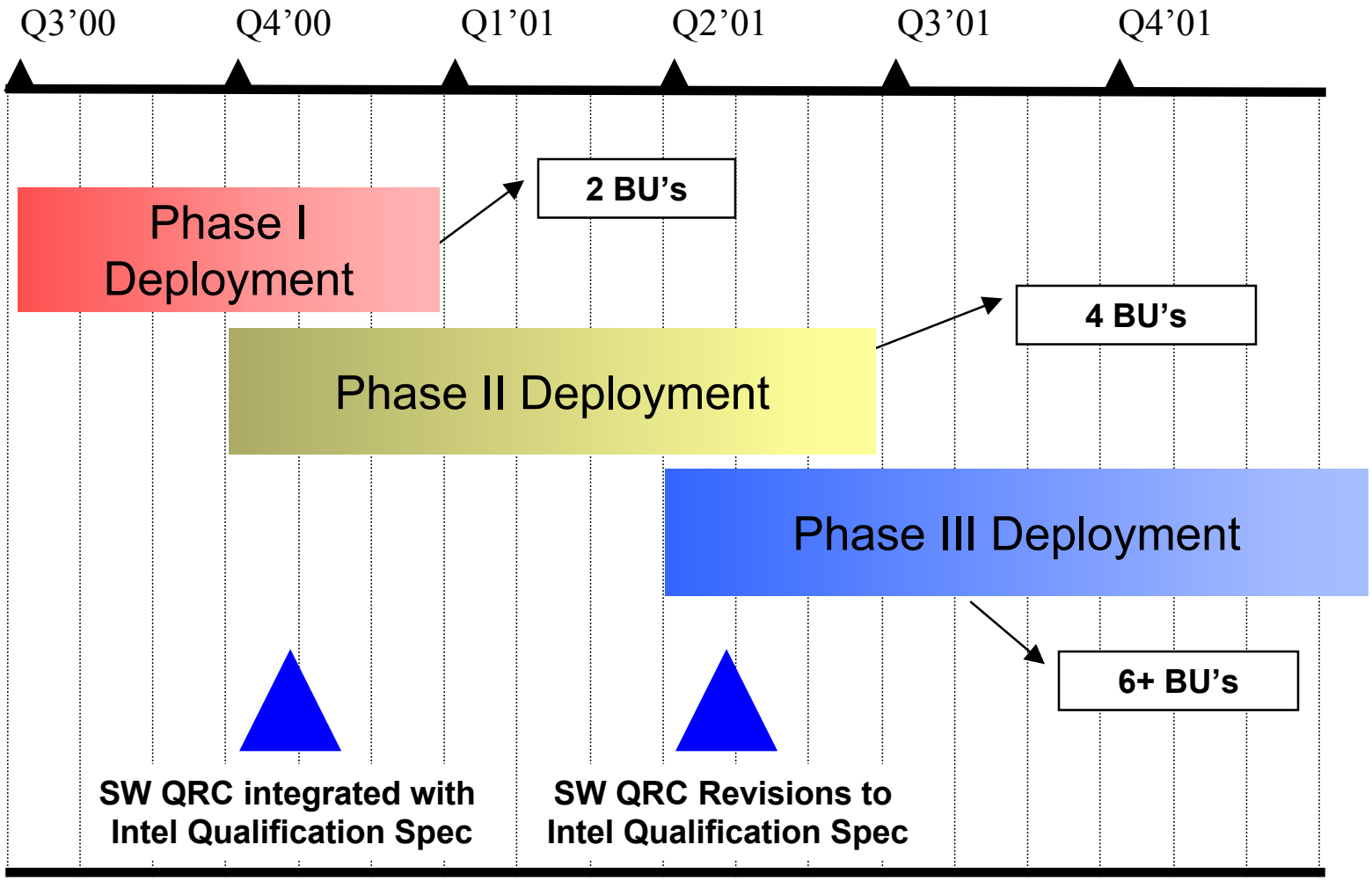
Intensive reviews with relevant company experts



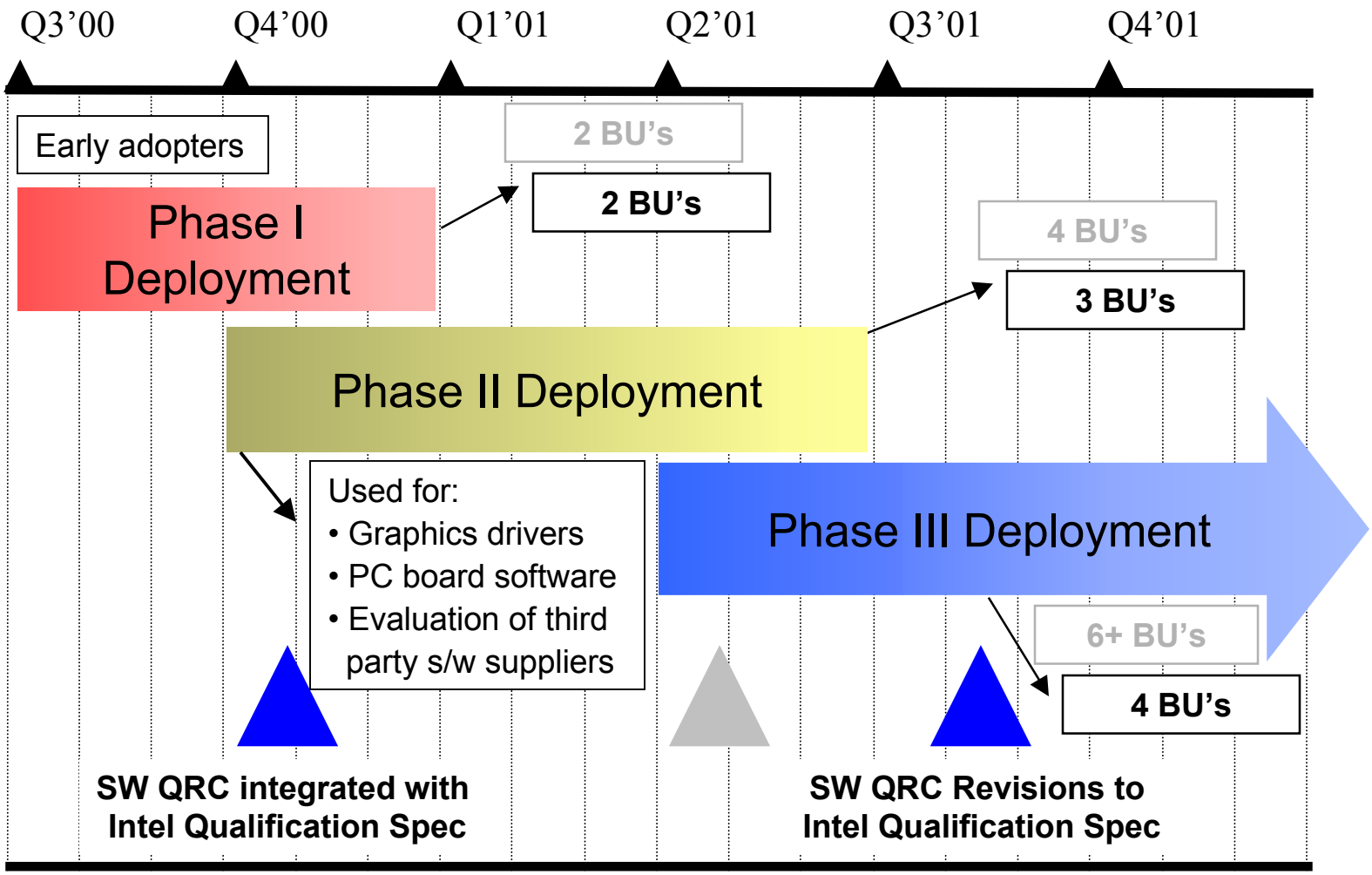
Sales Technique

- Pilot Program
  - Deployed to a small number of teams
  - Presented as a ***Framework***
    - Used to manage work they were already doing
    - Built upon and expanded their quality activities
  - Presented as something ***Tailorable/Adaptable***
    - Not “set in stone” – use as a starting point
    - Our motto: “It’s a framework, not a straightjacket!”
- Unsolicited Testimonials
  - Many people sold the idea for us, without our knowledge!

Planned Deployment



Actual Deployment



Reflection

"There is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than to take the lead in the introduction of a new order of things."

- Niccolo' Machiavelli

Barriers to Success

- Corporate Quality Network
  - No history of addressing software quality issues
  - Pervasive belief that software quality could be addressed just like hardware quality
- SW Process Engineers
  - Had competing interests for their Business Units
  - Driven by frequently changing quality priorities
- Product Development Teams
  - Not Invented Here (NIH) Syndrome
  - Resistance to Corporate involvement in engineering/quality
  - Breadth of Assessments stresses Test/SQA organizations

Surprises/Enablers to Success

- Corporate Quality Network
  - Enabled cross-divisional communication
    - Elicitation of Best Known Methods from many projects
    - Understanding of current (tribal) practices at Intel
- SW Process Engineers
  - Generated key stakeholder buy-in for their Business Units
  - Provided insight into specific project development issues in their Business Units
  - Provided tremendous help in defining and refining the criteria
- Product Development Teams
  - Some projects have “self-adopted” successfully
  - Developing metrics infrastructure helps adoption

Impact

- Enabled some Project Management best practices
  - Many projects have either slipped schedule or re-allocated resources based on SW QRC data

Customer quote: The SW QRC is "***the greatest thing since sliced bread***" – mentioned after the data in a scorecard had provided the justification for taking a 2 week schedule slip.

- Encouraged intra-team communication
  - It forced several miscommunication issues into the open
- Forced detailed planning up-front for quality
- Prevented issues & tasks from being dropped during project changes (project re-orgs, re-planning efforts, etc.)
- Provided a common software quality framework
  - Minimized impact when engineers move to new projects

Project Performance Analysis

- What went well
  - Pilot Program acceptance was enthusiastic
  - General acceptance of the framework
- What didn't happen as planned
  - Teams wanted more criteria than provided!
    - Desired a "Canonical" set of criteria
  - Deployment Program slower than planned
- Pleasant surprises
  - Many teams are adopting the SW QRC without any push
  - Teams are adding to the basic framework

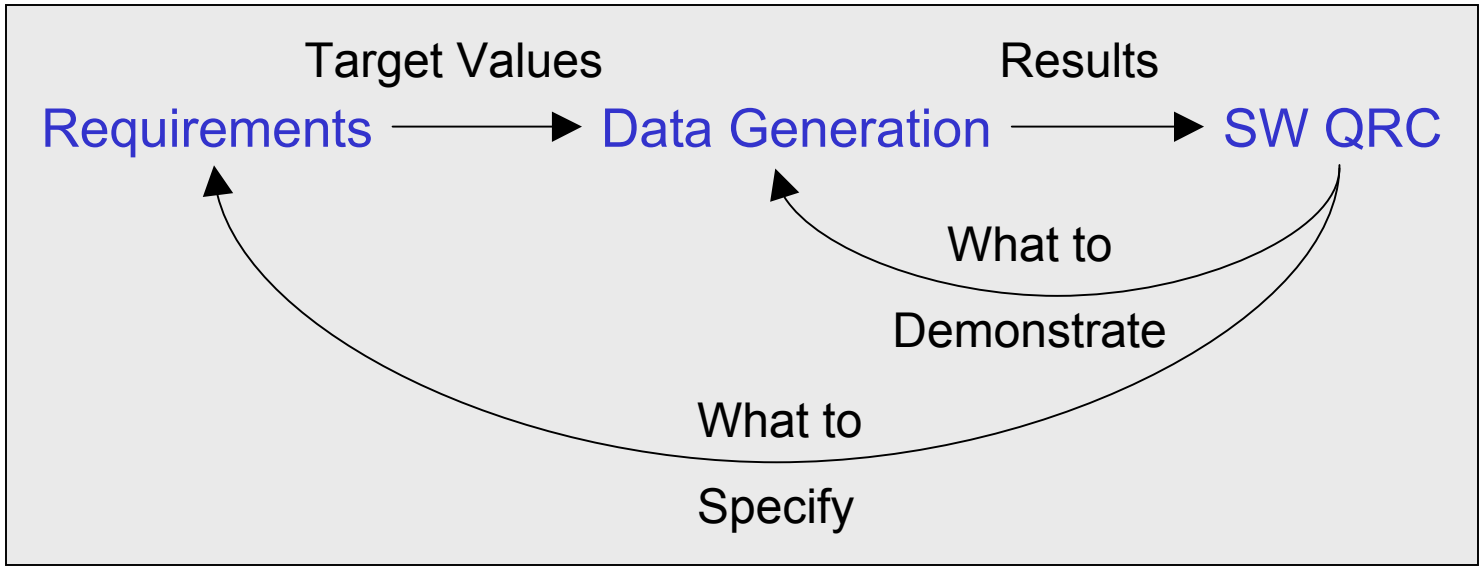
Next Steps

- Continued Deployment
  - Remaining Software Quality Engineers to be trained by end of 2001
  - Service Level Agreements being developed for adopting divisions
- Continuous Improvement
  - Add criteria targets by software type (e.g. BIOS, drivers, etc.)
  - Include feedback from SW QRC Working Group

Future Directions

- Requirements to Quality Criteria

Specification and Planning *Demonstration* *Monitoring and Reporting*



Integrate the quality planning, tracking,and reporting process more tightly with the requirements process

Lessons Learned

- Software Quality Initiatives are Hard
  - Corporate sponsorship is not enough
  - Grass-roots involvement is needed
- Plans Change
  - Discovery of how things are actually done can change emphasis
- Success Criteria
  - Keep the focus on solving the problems, not pushing an agenda

Secrets Exposed: Improving Schedule Accuracy

By Rick Anderson (rick.d.anderson@tektronix.com)

Key Words:

Scheduling, Software Project Tracking and Oversight, Estimation, Project Management

Abstract:

Welcome to my world. I'm the date the software for this product was going to ship (the so-called "End Date"). Note the words "was going to ship". Everyone knows the software won't really ship on this date. I was chosen from 364 other dates because I was viewed as the most optimistic date a super hero could meet to deliver the product. When they first chose me, I was lauded with cheers of excitement and I was put in documents and on white boards everywhere. But now I'm nothing but cursed. They selected me pretty much at random. They didn't look at project history nor do any formal estimation methods. And of course, I was the only one chosen. They didn't have the foresight to give a range of delivery dates. As the project progressed, they really didn't look much at me, instead focusing on getting more features in and fixing all those bugs they introduced. I knew I was in trouble early in the project when the requirements document was published and it was only two paragraphs. And now they are saying some date halfway across the calendar is the new delivery date. I was doomed from the beginning…

Does any of this sound similar to your product development end dates? Far too many of our software projects are late. As software engineers and software quality engineers, we must do a better job meeting our end dates. This paper will highlight some of the areas where we go wrong and give you some insight into how perform and influence better project scheduling, tracking and oversight, estimating and overall project management.

Author Biography:

Rick Anderson has over 15 years experience with embedded, real-time product development, software quality assurance and software process improvement. He is currently a Senior Software Engineering Manager at Tektronix, Inc. in Beaverton, Oregon in the Oscilloscope Product Line. Tektronix is a leading provider of test and measurement equipment for the telecom, computer and semiconductor industries. Rick leads both product development and software quality engineering teams in his current role. He has degrees in Computer Science and Business Management from Oregon State University. Rick last presented at PNSQC in 1997.

Rick can be reached via email at *rick.d.anderson@tektronix.com*, phone at 503-627-2630 or postal mail at:

Tektronix, Inc.
Rick Anderson, M/S 39-732
P.O. Box 500
Beaverton, OR 97077.

Secrets Exposed: Improving Schedule Accuracy

By Rick Anderson (rick.d.anderson@tektronix.com)

Introduction

Historically and industry-wide, software teams have shown they rarely deliver on scheduled end dates (and often intermediate milestones are no better). Estimation is lackluster, proactive management of the software schedule is weak and there are several mid-project actions which reduce the ability to meet the forecast milestone dates. Time and time again, software projects are late.

According to research by the Standish Group, only 16.2% of all software projects are completed on time and on budget. They also found that the average project schedule overrun is a whopping 222% (and this says nothing about the reduced feature sets which go along with these overruns). Looking at some recent projects within Tektronix Instruments Business Unit, 16 of 22 projects were late by more than a month (comparing actual end date against the end date proposed at the beginning of the project). Clearly our track record with schedule accuracy is not good and we must do better!

The title of this paper indicates we are going to expose secrets to improving schedule accuracy. These so-called "secrets" are really common sense things that every good software project should already be doing. But like many things that we should do, we often don't. Understanding what causes schedule inaccuracy and having some tools in our bag of tricks to improve it should greatly improve your chances of making both mid-project and end-of-project dates. Most of this paper will focus on the software end date, but the techniques discussed apply equally to mid-project milestones as well.

Consider this non-exhaustive list of causes of schedule inaccuracy:

- Reliance on externally set dates (including only top-down ones) instead of internally generated, bottom-up dates
- Added features (feature/requirements creep and/or gold plating)
- Lack of resources (including attrition, absenteeism, changes in organization structure, resources pulled to other projects)
- New engineer learning curves
- Missing tasks on project schedule
- Daily interruptions that engineers typically experience
- Multitasking engineers on multiple projects and not accounting for task switching
- Under estimating
- Low quality at earlier stages of the lifecycle
- Bad communication, including not having full team involvement in task and schedule definition
- Development environment problems (including lack of prototypes, tools, etc.)
- Third party problems
- Late hardware / accessories / platforms

Listed below are some recommendations for individuals and teams on how to improve their overall schedule accuracy. These have been grouped into estimation, scheduling and project lifecycle practices. The recommendations are presented in no particular order. The key to success is to incorporate multiple approaches to achieve better results.

Estimation

Consider this not uncommon definition of estimation: "An estimate is the most optimistic prediction of when the software can be done assuming everything goes right." This definition dooms us to failure before we start! Research has shown developer estimates tend to be optimistic by 20-30% (van Genuchten, 1991).

Here are some recommendations on how to do a better job in the area of estimation:

- **Combine top-down estimates with bottom-up ones.** Often schedules are passed down from above (project is needed by end of year to match competitor offering, product release is tied to upcoming conference, etc.). These so-called top-down "estimates" are based on a business or market need. The software team must also estimate the project delivery dates given a particular feature set. This should be compared to the business or market need date and the various controllable aspects of the project should be negotiated (schedule, resources, features and in some cases, manufacturing cost).

- **Schedule targets.** Instead of a single end date, consider the use of best case, expected case and worse case delivery dates. Best case targets should represent "pull-in" dates that can be achieved if everything goes completely right - a stretch goal the software team can work towards. Worst case targets include time for unplanned hardware chip or board turns, third party contractor delays, additional resource training, unplanned activity to improve quality (fix defects, rewrite code), etc. It is recognized that worse case delivery could be "never", so this should be estimated with 95% probability of meeting the worse case date. All three target dates should be shared with management to better understand risks and schedule variability.

  One suggestion is to incorporate this into the Software Product Plan document. You can outline the three delivery dates, and then comment on those items that could be improved on (schedule compression possibilities or best case schedules) and those items that might adversely affect the schedule (or worst case dates).

- **Schedule probability.** As an attempt to put a stake in the ground, software teams should consider the following guidelines when publishing "expected" schedules. Clearly probability is difficult to judge, however a common "confidence factor" will help to normalize estimates across projects as well as within the team. Note that as the project progresses, the schedule should be more accurate.

- Initial schedules (at first draft of Software Project Plan) should have a 50% probability of being accurate.
- Mid-project schedules should have a 70% probability of being accurate. It is best to tie this to some standard mid-project milestone so "mid-project" is defined.
- The Software complete milestone (all coding done, but not yet fully tested and debugged) should have increased probability, with >80% accuracy.

- **Multiple estimation techniques.** Consider using multiple estimation techniques when trying to nail down the end date. The more techniques you utilize, the better your estimates will probably be. Some estimation techniques include:

  - Estimation technique #1: The software project leader might estimate the project alone and given their experience, this is the proposed end date. This estimation technique is often used very early in the project when much is unknown about the product. By the way, research has shown that the initial estimate can be off by a factor of 16!

  - Estimation technique #2: Bottom-up estimates, where each Software Engineer does his or her own estimate of their tasks, and then these are then aggregated to come up with the software team end date is another method. Areas not included in primary feature development, such as infrastructure and system integration tasks are then added. It is important to engage the complete software team in schedule estimation on an on-going basis.

  - Estimation technique #3: Compare your upcoming project to similar past projects. Keeping past project history metrics like project size, duration and resources can be one of the most helpful estimation resources at your disposal.

  - Estimation technique #4: Use software estimation tools such as SLIM-Estimate or COCOMO. SLIM (Software Lifecycle Management) is a good tool to tell you if your initial estimate (in lines of code) is "in the ballpark". The tool can also give you some insight into manpower loading curves. SLIM has a vast project history database and you can even enter your own project history and compare it to your current project.

  - Estimation technique #5: The Delphi technique asks various experts in the company to estimate the same task. These experts might or might not be on the project. After each individual estimates the task independently, the results are shared between all the estimators and individuals comment on their estimates. After discussion, another round of estimation then commences and this continues until there is rough consensus.

- **Re-estimate often.** Estimation should not be a one-time event. As more becomes known about the project, the project end date should be re-estimated and affirmed or changed. On most long projects, it makes sense to re-estimate once a quarter (see *Schedule Scrubs* below).

Scheduling

Another aspect to successfully meeting end dates is to utilize proper project management techniques when scheduling. Like most of the topics herein, entire books have been written on this topic. Listed below are some of the more common areas of project scheduling, that if done properly, will result in a better chance of meeting your software milestone dates.

- **Use a scheduling program.** There are many software programs that will ease the burden of a detailed project schedule. MS-Project seems to be one of the more popular ones. Regardless of which tool you use, use something. These tools can do the basics, like keeping track of tasks, duration, dependencies and resources. This will result in a detailed work breakdown structure. Once a week, project teams can review and update their schedules. Schedule pull-in activities can then be considered. Companies like Lateral Work Systems have complete fast-time-to-market methodologies (and tools), which although not perfect, do have some merit.

- **Schedule all known tasks.** This includes vacations, holidays, unit/integration/system testing, defect fixes, technical reviews, sales training, documentation reviews (including non-software engineering documentation such as users guides, programmer guides, software quality engineering test cases, etc.). You should even schedule time to create the schedule and then manage it over time! Missing tasks often account for 20-30% of the overall schedule (van Genuchten, 1991). For near term tasks, they should be broken down into 3-5 day tasks. Long term tasks can have less granularity. Dependencies between tasks should also be clearly enumerated.

- **Schedule some rework time.** Rarely has a software project gone from beginning to end without some amount of documentation, code and test rework. This might be so we can support future reuse, or because a particular subsystem has gotten too complex or because a particular feature is not performing to specification. We rarely schedule time to re-architect or re-design a portion of the system, yet this often happens. Think back on past projects and consider some of those unscheduled areas and consider putting in scheduled time at appropriate places for some of the unknowns.

- **Loading.** Software engineers get involved in many different things, including unscheduled tasks for other projects, recruiting, meetings, customer support or sustaining activities, etc. We should estimate tasks with these “interruptions” in mind. A task that would take 8 working days of uninterrupted work potentially should be scheduled for 10 days. Said another way, either add 20% on top of the estimate for day-to-day tasks or load the schedule at 80%.

- **Project milestones and mini-milestones.** Most projects consist of more than just software. There are often project-wide milestones. These major milestones revolve around the completion of major deliverables (product requirements complete, prototype complete, etc.). Generally all projects within your company will have the same set of milestones. Besides these standard project milestones, consider adding several software-only milestones in the middle of the project. These milestones should be tailored to the particular project and software development effort. These are often called mini-milestones. Example mini-milestones might include software requirements complete, incremental phase 1 complete and software code complete. It is recommended that the mini-milestones be scheduled every two months for longer projects. This gives the software team a short-term goal to work towards (which will seem more achievable than trying to hit an end date a year or more away). Track results against these milestones and factor trend data into later project schedule activities. Hitting mini-milestones on schedule will increase the teams confidence they can hit the end date. And if you miss a mini-milestone, it probably means your end date should be pushed, as lost time usually cannot be made up.

- **Weekly schedule reviews.** Review the schedule often with the project (and technical) leaders on the project. Like estimation, scheduling is not a one-time event. The schedule should be reviewed and updated every week, marking % complete for all tasks and modifying other tasks as appropriate. These reviews should take 15-60 minutes per project leader. The focus here should be slanted towards upcoming milestones, although the long term should not be totally dismissed. Any opportunities to potentially expedite or pull-in tasks should be discussed.

- **Schedule scrubs.** Every one to three months a detailed "schedule scrub" should be completed. These schedule scrubs focus on the longer-term schedule, hopefully to uncover new tasks, incorrect task duration, new dependencies, etc. Note that the focus is on risks, the end date and the critical path, versus the short-term focus of the weekly schedule reviews. Potentially these scrubs can occur at the same time the major milestones or mini-milestones occur. A detailed schedule scrub might take 4-8 hours and involve everyone on the project, management and even other scheduling experts in the company. The scrubs might be accomplished much like a Formal Technical Review (hand-out schedule, review, meet to discuss), an Informal Technical Review (hand-out schedule, review, send in comments) or on a group-by-group basis (each group meets for 30-60 minutes in front of the schedule).

- **Minimize Dependencies.** By doing intelligent planning and task scheduling, you can often create schedules that have a better chance of succeeding. For example, if there are certain people in your organization who do not do well under stress, try to keep them off the critical path. Another technique is to prioritize tasks so as to limit dependencies. You can do this with your most risky activities. This way if a particular task is delayed, it has less chance of affecting the upcoming milestones.

Project Lifecycle Factors

The term "project lifecycle factors" refers to those things that, if done well at the beginning of the project, should result in savings (time, cost, etc.) later in the project. And the inverse, if not done well, will result in more time (missed schedules!), cost, etc. Some recommendations here include:

- **Requirements.** It probably goes without saying, but many projects still do not do them. Having documented requirements or specifications that are complete and reviewed will go a long way towards improving schedule accuracy. The same could be said of architecture, design and test plan documents.

- **Quality of specifications.** Directly related to schedule accuracy is the quality of our design specifications. Complete, reviewed and up-to-date requirements, architecture, designs and test plans can greatly reduce the risk of a missed end date by increasing quality. Do not sacrifice design time just to meet schedule. If designs are not complete, quality will suffer. Much research has proven this is a pay-me-now or pay-me-later scenario. Cutting corners early in the project will result in low quality and more time spent at the end fixing defects. Some specific recommendations in this area include:

    - Review everything. All software documents (project plans, requirements, architecture, design, test plans, etc.) should be reviewed by appropriate individuals in the organization. These reviews should be planned, scheduled and communicated. The goal is to find defects early in the project. Code reviews are recommended for code that is complex and/or critical.

    - Consider past defects. Special emphasis should be placed on areas where defects often occur, including start-up, shutdown, system performance, error handling, public interfaces between subsystems or systems and late requirements.

    - Include stretch goals in your requirement specifications. Each requirement should be uniquely identifiable, complete, testable and should have a priority next to it (such as Essential, Desired and Optional). This allows the software team some flexibility (implement all the E's, do D's and O's as time permits).

    - Some companies that have been successful in the area of schedule accuracy recommend doing more complete designs. They do both "high level designs" and "detailed designs", the latter getting to the algorithm and pseudo-code level. While this step takes additional time, this time is saved in the end because the code can be implemented faster and defects are exposed earlier. It is also felt that this helps expose other open issues that have yet to be resolved.

- **Account for requirements creep and requirements gold plating.** New features have a way of showing up and existing features have a way of becoming bigger. Software teams need to do a better job of resisting this, especially after requirements have been frozen and end dates given. Whenever new features are added in the middle of the project, the software team should immediately rescrub the schedule to account for the feature. We know that adding features late in the project is more expensive then if they were added during the requirements phase. These same features are also more prone to error because they most often do not go through the normal development process (left out of architecture, design, test plans, etc.) Having a solid, cross-functional change control board, which controls new features, can help with this issue.

- **Integrate often.** Frequent code integration cycles are important for success. Getting a small core of the system working solidly and then building on this in a series of layers is a solid recipe for success. Integration cycles should include full team product evaluation and bug fixing when appropriate. Use evaluation checklists (Software Engineer created list of manual tests that take into account white box understanding of code), and rotate assignments to encourage team ownership of product quality. Begin regression and stress testing as early in the project as practical to uncover fundamental system problems that often require long system fix times.

- **Mid-project correction.** For projects over 1.5 years, a mid-project review should be held (Anderson, 1997). One of the issues that should be discussed is schedule accuracy to date, state of requirements, architecture and design, resources and other schedule related items. This can serve as an excellent mechanism to air potential schedule issues with the entire software team so the issues can be addressed at a time in the project when something can be done about it.

- **Get the software team involved.** Design discussion meetings help reach consensus, educate and help solve difficult problems by having "team mindshare" to solve the problem. These meetings can be adhoc or scheduled weekly or biweekly, but they must happen to be successful. Team alignment is a requirement for schedule accuracy.

- **Use metrics.** Many organizations could do a much better job using metrics to help understand where they stand in the middle of a project. Metrics such as schedule accuracy to date, defect curve as compared to historic averages and others can be a great indicator to a software project leader and program manager that the project is falling behind a scheduled end date. SLIM Control (another Quantitative Software Management tool) is a metric-oriented tool that might be able to help in this area. Track success against a plan you create early in the project.

Miscellaneous

In addition to the above, here are some other general guidelines that often apply:

- Schedule pull-in opportunities exist early in the project, not towards the end. If a task goes late, generally this time cannot be made up. Research has shown that the maximum schedules can be compressed is 25% (Boehm 1981, Jones 1994) through adding resources and requiring overtime. Usually when we miss something, we assume it can be made up. Instead we should consider adding the missed time to the end, or even more radically, consider that future tasks will also slip by similar amounts.

- While it is obvious, open communication and visibility between the software project leader, software team, software functional manager, software quality and the project manager as the schedule starts to slip is necessary.

- Splitting an individual between two (or more) projects should be avoided. If required, the individual should only be scheduled at 80% total capacity, as context-switching time between projects is required.

- Similarly, concurrent development (common code bases, platforms, and applications) has overhead associated with it and this should be scheduled.

- Recognize that code integration takes time and schedule it appropriately.

- Avoid the philosophy of under estimating to get approval to move forward on a feature. By definition, this is schedule inaccuracy!

- Make sure the software team clearly understands the customer and has the appropriate domain knowledge. If they do not, schedule time to obtain it.

- If new ASICs are being introduced, consider the amount of driver development and testing these chips have undergone. Understanding this helps schedule the time to implement features on top of the chip.

- Setting public goals (hitting mini-milestones, reducing defects to x by a certain date) are good for motivation and communication.

- The sharing of status reports can be a good mechanism to keep each other informed. The same could be said of "change alerts", a mechanism whereby when code is checked-in, everyone else on the team is informed of the change. At Tektronix, we handle this through a combination of email and tool support.

- The software project leader should recognize that some individuals are good at task estimation and others are not. Understanding this will help make minor adjustments to compensate for differing skill levels and inaccuracies.

- Manage and clearly understand those tasks that are the highest risk, the longest duration or the most unknown. Spend more time on these tasks.

- Allow enough quality time for estimation and scheduling to occur.

- Parkinson's Law: If estimates are too high, the work will expand to fill the amount of available time.

Conclusions

The above list is a good start at some things software teams can do to improve schedule accuracy. The list, however, is not meant to be complete. Reflect on your own thoughts and experiences in this area. Through continued focus, open team communication, hard work and schedule reviews, schedule accuracy will improve and you will improve your chances of being one of the successful 16% of software projects.

Bibliography

Anderson, Rick D., 1997. *"Maximizing Lessons Learned: A Complete Post-Project Review Process"* in the *Pacific Northwest Software Quality Conference 1997 Proceedings.*
.

Boehm, Barry W. 1981. *Software Engineering Economics*. Englewood Cliffs, N.J.: Prentice Hall.

COCOMO Software Estimation Tool (*http://sunset.usc.edu/research/COCOMOII/index.html)*

Jones, Capers, 1994. *Assessment and Control of Software Risks.* Englewood Cliffs, N.J.: Yourdon Press.

Lateral Work Systems. *(http://www.lateralworks.com)*.

SLIM-Estimate and SLIM-Control Software Tools (*http://www.qsm.com*).

The Standish Group, 1995. "Chaos." (*http://standishgroup.com/visitor/chaos.htm*).

van Genuchten, Michiel, 1991. "Why is Software Late? An empirical Study of Reasons for Delay in Software Development." *IEEE Transactions on Software Engineering*, vol. 17, no. 6 (June): 582-590.

The Tale of Two Projects

Does following 'best' practices really make a difference?

Doug Reynolds
Tektronix, Inc.
P.O. Box 500, M/S 39-732
Beaverton, Oregon 97077
douglas.f.reynolds@tek.com

Abstract

It is not often that you have the luxury of comparing two projects that are so closely related yet utilize so many different practices to get to the same goal (to ship a viable product). Both projects used the same development platform, tools, policy, and started and ended at approximately the same time, however the underlying results were quite different. Comparisons of common 'best' practices like peer reviews, requirements engineering, and configuration management were observed during the life of the projects. The results of these observations will be presented in a paper and presentation as lessons learned.

Biography

Doug Reynolds is a Software Quality Engineer at Tektronix with the Instrument Business Unit. He has worked for Tektronix for the past 9 years. He is currently working on a Master of Computer Science and Engineering degree at Oregon Graduate Institute (OGI).

Keywords

Best Practices, Change Control, Defect Tracking, Configuration Management, Lesson Learned, Peer Reviews, Post Project Reviews, Requirements Engineering

1.0 Introduction

In this paper I compare how two similar but different projects implemented different 'best' practices, observe the effectiveness of these 'best' practices, and then present what lessons have been learned. Some of the 'best' practices analyzed are Requirements Engineering, Peer Reviews, and Configuration Management. In many cases common software metrics (defect counts, requirements collected, etc) can be used to compare the effectiveness of different practices as utilized by the two project teams.

To help distinguish between these two projects I will refer to them as Project A and Project B. Both projects include embedded hardware, embedded software, a real-time operating system (VXWorks), and a Windows type user interface. Both projects worked under many of the same constraints, which included the organizational development policy, development platform, and development tools. The following table helps illustrates the size and complexity of the projects.

Table 1 – Project Similarities

Item	Project A	Project B
Average Full-time Software Engineers	20	15
Average Full-time Software Quality Engineers*	3	3
Duration (months)	24	24
Lines of Code (LOC)	1,700,000	1,400,000

*Software Quality Engineers perform both validation and verification activities

The tale of two projects is a look back and a comparison of how these two similar but different, projects achieved the same goal (to ship a viable product) under similar constraints. Both projects were successful in their own ways (one project has won prestigious industry awards, the other project has overtaken market share from its competitors). Both projects had room for improvement in many areas such as project planning, scheduling, and risk management. Both projects followed the intent of the organizational policies; however, each selected their own way of implementing the policy requirements. I will compare the effectiveness of these decisions.

2.0 Background

Both projects worked within an ISO-9000 certified organization that had the goal to 'consistently meet or exceed customers total expectations'. The ISO-9000 series of standards and the organizational development policy specified what, at a minimum, must be done; it did not specify how things were to be done[1]. For a project team to fulfill the requirements of the development policy, the project team must document through the project plan how they are going to perform the required tasks (policy requirements). The project team must then gain approval for how they are going to perform these tasks (management reviews), and then execute what they have documented. A brief example is that the policy states that a project must use a defect tracking system, the policy does not state how the project will track defects – this is left up to the project. In summary, the project teams have the flexibility to do what makes sense, all they have to do is 'document what they are going to do', gain management approval, and then 'do what they have documented'. Both of these projects followed the intent of this policy, which

provided a fair amount of latitude. Because of the project similarities I am able to compare the effects of the decisions made by these projects.

The following sections are broken down into the areas to be compared. In each case an introduction is provided, a discussion on how the project teams executed (or did not execute), and a summary of what lessons were learned. In many instances the implementation (or lack there-of) of different policy requirements attributed to the success and/or failures of the project team. In many cases these differences in implementation of best practices can be equated to lessons learned!

3.0 Requirements Engineering

"Software requirements engineering is the science and discipline concerned with establishing and documenting the software requirements". [2] This process consists of software requirements elicitation, software requirements analysis, software requirements specification(s), software requirements verification, and software requirements management. The development policy used by both projects requires the collection, analysis, verification, and management of the software requirements, which includes the generation of a software requirements specification(s) before software planning can take place.

Both projects followed the intent of the development policy by performing some degree of requirements engineering and requirements management. Both teams collected requirements and generated a software requirements specification. Both teams had the initial software requirements reviewed and placed under version control, and both teams defined a change control board. Neither team did a very good job of managing requirements creep, prioritizing the tradeoffs, and scheduling the amount of effort for each of the requirements. These items became apparent when the projects started discussing which requirements could/would be dropped to help meet the project schedules.

There were some differences between the levels of requirements management. Following the initial review of the software requirements, which were then used to generate the initial project schedule, project A spent an additional three months analyzing, prototyping, refining, and re-reviewing all of their software requirements. Their refined requirements would probably be rated as 'B' quality requirements (on a scale of A, B, C, D, and F, where A is the best). Project A treated the requirements specifications as living documents tracking changes to the specifications using a defect tracking system, updating the requirements specifications, and then having the changes re-reviewed by a change control board. During this period Project A was able to identify missing functionality, missing product constraints, and break more complex requirements into easier to understand requirements. What Project A failed to do through this process was to re-scrub the schedule (update the work breakdown structure and use this to update the schedule) to reflect the revised requirements.

Project A also worked with their Software Quality Engineering team to implement requirements traceability. This enabled them to ensure that the requirements had a stable and unique identifier which could be used to trace the software requirements between the software requirement specification, test plans, and test cases. At Abbott Laboratories, where traceability

was instituted in 1987, they like to say, “You can’t manage what you can’t trace”. [3] Even though Project A did a good job of updating their requirements Project A did not prioritize their requirements (i.e. Essential, Desired, or Optional).

Project B on the other-hand rushed through the requirements engineering process and proceeded on with loosely defined requirements. These requirements would probably be rated as ‘D’ quality requirements. The requirements were done as a one time shot and never updated.

Since Project A had more thoroughly completed requirements, the team was able to educate each other through peer reviews/inspections and get other organizations involved early in the process (Customer Documentation for writing manuals, Software Quality Engineering in developing test plans, etc). Besides the defects that were removed through the entire review process, the software team also benefited from the other organizations combing through the requirements to understand the product behaviors.

Project B had a little more difficult time. Even though the team jumped ahead into design and implementation, later the team had to spend a fair amount of time educating other groups. For example Customer Documentation did not understand the behavior of the product, which made it difficult to write manuals. Software Quality Engineers did not fully understand the behaviors that needed to be tested (especially as requirements changed and were not documented). The Change Control Board had a difficult time trying to reduce features since the requirements were so out of date that the initial requirements prioritization (Essential, Desired, and Optional) became useless. To resolve some of these issues, Project B ended up holding special weekly meetings later in the project to disseminate the information to other groups.

The following table is used to summarize some of the differences between the two projects. One important note is that the requirements count for both of these projects should intuitively have been about the same (not a difference of 5X). Even though Project B did prioritize requirements initially, these priorities were of no benefit because the requirements were not kept up to date.

Table 2 - Summary of Requirements Engineering

Item	Project A	Project B
Requirements Count*	2223	462
Implemented Requirements Traceability	YES	NO
Prioritized Requirements	NO	YES
Requirements treated as living documents	YES	NO

*Intuitively both of these numbers should be at least 2000 based upon product features

Some lessons learned about Requirements Engineering:

1) Software Requirements Specifications should be treated as living documents by updating them as often as necessary. If the software requirements are treated as living documents then other organizations like Customer Documentation and Software Quality Engineering can take advantage of them by completing their tasks earlier and with less re-work. In the end, Project B added two Software Quality Engineers to write and re-work test cases due to

out of date requirements. Project B also spent more time entering, investigating, and rejecting these defects. During the evaluation of these defects it turns out that many of the rejected defects were due to non-reported changes in the Software Requirements Specifications.

2) Software Requirements should be prioritized (Essential, Desired, Optional; High, Medium, Low; 1, 2, 3; etc.) and used to maintain the work breakdown structure and schedule. Both projects had a difficult time evaluating the effects on the schedule while trying to remove non-essential requirements at the end of the project. Project A never prioritized their requirements, Project B prioritized their requirements but did not treat their requirements as living or keep them tied to the work breakdown structure.

The process of requirements engineering is initially one of the most important processes that we go through. Through this process we not only define the product requirements but we should use these requirements to make our commitments through project plans, scheduling, and risk reduction activities. The amount of commitment and discipline that we provide to this process will pay significant dividends in the final cost and quality of the product. A helpful resource in refining the requirements process and provides a step by step approach to requirements management is a book by Suzanne Robertson and James Robertson, "*Mastering the Requirements Process*". [5]

4.0 Peer Reviews

Peer reviews is a process by which a work product is examined by peers with the primary goal of finding defects. [6] Peer reviews are a form of verification for these projects, based upon the policy 'design verification'. The development policy calls out that the Software Project Leader is to ensure that peer reviews are held for certain documents such as software project plans, software quality plans, software requirements specifications, and software design specifications. Other documents do not require reviews but reviews could be held for documents such as source code, test plans, test cases, etc. Even though the software project leader is 'responsible' for making sure that the reviews are done, it is usually left up to the document author to setup the review and select the participants. The development policy also states that the Software Quality Leader is to include in the Software Quality Plan what peer review philosophies will be used on each project.

Looking into the Software Quality Plan for Project A it was determined that Project A planned to "train, encourage, and facilitate effective reviews on requirements, designs, and code implementations for the Project software team". For the Peer Review process Project A had defined the following: the purpose, goals, techniques and methodologies, completion criteria, metrics, and resources. On the other hand, looking into the Software Quality Plan for Project B, the only reference to Peer Reviews dealt with what the team would do with the results or the reviews (track items through the defect tracking system, keep records as verifiable objective evidence that the review took place, etc.). The following table shows a summary of key documents (not all documents are shown) each project produced, reviewed, and maintained. Note that Project B only reviewed their documents 'once' at the very beginning of the project. For many projects you may be able to get away with updating project documents, but the nature

of these projects requires the Software Requirements Specifications to evolve. The paper *IEEE Recommended Practice for Software Requirements Specifications*, describes how Software Requirement Specification evolution may be needed as the development of the software product progresses. This paper points out that it may be impossible to specify some details at the time the project is initiated and that additional changes may ensure as deficiencies, shortcomings, and inaccuracies are discovered in the SRS. [11]

Table 3 – Summary of Document Management (Peer Reviews)

Item	Project A		Project B	
	Reviewed	Living	Reviewed	Living
Software Project Plan	YES	SOME	YES	NO
Software Quality Plan	YES	YES	YES	NO
Software Requirements Specifications	YES	YES	YES	NO
Software Design Specifications*	YES	YES	N/A	N/A
Software Test Plans	YES	SOME	> 5%	NO

SOME- implies that the document(s) were kept up to date for over half the project (i.e. just not a one time shot)
* Project B did not produce any Software Design Specifications

The Peer Review process for Project A deserves further discussion. Early on, Project A was headed down the same path as Project B with regards to Peer Reviews (even though the Software Quality Plan had stated goals). Team members acknowledged that peer reviews were a good thing, but by the same token the team members grumbled about holding Formal Technical Reviews (just like Project B). Several items helped encourage the software team to perform peer reviews.

One, Project A's Software Quality Leader worked with the Software Project Leader to sell the process. Peer reviews find from 60-90 percent of the defects in a program, which is considerably better than walkthroughs or testing. Because they can be used early in the development cycle, peer reviews have been found to produce net schedule savings of from 10 to 30 percent. [7] One study of large programs even found that each hour spent on peer reviews avoided an average of 33 hours of maintenance, and peer reviews were up to 20 times more efficient than testing. [8]

Two, the Software Quality Leader worked with the Software Project Leader to train the software team. An internal training session was provided to the software team called, "An accelerated introduction to the Software Review Process". This training session presented the following:

1) Why do we do peer reviews and how effective have they been within software engineering (sell the team on reviews)
2) Basic principles behind what peer reviews really provide (defect removal, deciminate product/domain knowledge, help people reach agreement, etc.)
3) Different review types used within the organizations (Inspections, Formal Technical Reviews, and Informal Technical Reviews – desk reviews)
4) The Roles played by those involved in the peer review process (Author, Scribe, Moderator, Reader, Participants)

5) The Phases of the peer review process (Planning, Preparation, Review Meeting, Rework, and Feedback).

Three, the Software Project Leader worked from a Review Planning Matrix that was reviewed during the weekly team meeting. The Review Planning Matrix (RPM) was a chart with each line representing a document that needed to be generated (or updated). Each line contained items like who was to generate (or update) the document, who were expected to be the required reviewers, and who were the optional reviewers. Since the RPM was updated during each of the team meetings, the team knew who was reviewing what document. This helped prevent one expert from getting stuck performing all the reviews – the reviews were equally divided among the team. The RPM was color-coded to show the progress of the team. Each line of the RPM started out red to show that the documents had not been generated/updated. As the documents were generated/updated the line representing the document turned yellow, and finally the line representing the document turned green after the document had been reviewed, updated, and approved. For each group of documents, the completion of the documents (RPM had only green lines) was a mini-milestone and the team celebrated with a team lunch.

Last, the Software Quality Leader kept metrics on the review process (the number of documents reviewed, the time spent reviewing documents, the total number of pages reviewed, the number of action items found, and the number of major issues and/or defects found). Grammar, spelling errors, and minor issues were not recorded. The metrics were summarized and presented to the team on a regular basis. For example the results of the first three documents were presented, which had a total of 15 reviewers, totaled 89 pages, took 31 hours and 45 minutes, were two action items and 98 unique issues/defects were identified. The documented results from these first three documents helped motivate the team. The collected metrics also proved useful to determine if a document should be re-reviewed by comparing it to the results of other documents. For example, was too little time spent reviewing a document, were too many defects found for the time spent reviewing the document, etc.

Though Project A initially planned on using Formal Technical Reviews, the peer reviews actually performed by Project A were (what is called internally) Informal Technical Reviews (ITR's). This type of review is really a desk review where reviewers perform the peer review at their desk and provide feedback via Email (for action items, defects and major issues) and via handwritten comments (for spelling, grammar, and minor issues). This review method may not have been as successful as using Formal Technical Reviews (i.e. lack of discussion in meeting, having to explain everything via Email, etc) but at least the reviews were performed, defects were found, and domain knowledge disseminated. The ITR process worked as follows:

1) The author filled out a form called the review meeting notice which provided the information about the review
2) The reviewers then provided comments back via Email using a review comment form (which contained verification of approval)
3) The author compiled the results (fixed defects, resolved issues, etc) using a form called review meeting minutes.

Some lessons learned about the Peer Review Process:

1) Peer reviews are more effective if you achieve buy-in from the Software Project Leader (Project B was never able to achieve this buy-in), train the software team on the peer review process, and tailor the review process to the software team (provide the framework for success).

2) Use a Review Planning Matrix to plan track the progress of peer reviews through milestones. The Software Project Leader used the Review Planning Matrix to encourage the team to write the documents and perform the peer reviews in a timely manor because nobody likes to see their name in Red.

3) Collect metrics from Peer Reviews. The metrics can be used to motivate the software engineering team and to determine if the work products have been adequately reviewed. The Software Quality Leader from Project A was able to show the software team the results of their Peer Reviews (it is important that these numbers are not used to beat-up the team but to motivate the team).

5.0 Configuration Management

Configuration Management (CM) is a multi-part discipline for applying technical and administrative direction, control and surveillance for 1) configuration identification, 2) configuration and change control, 3) configuration status accounting, and 4) configuration audit. [9] Configuration identification is used to identify baseline and revision levels. Configuration control (also known as change control) is used to ensure that each change is known, authorized, and documented through a change control board. Configuration status accounting is used to record and track problem reports, change requests, and change orders. Configuration audit(s) are unscheduled audit(s) for standards compliance.

Configuration management is a key for managing and controlling software projects. The development policy requires that each project set up a configuration management system during initial project planning. Because of the nature of these projects, both project A and B actually used multiple CM systems. Project A used ClearCase for their embedded VxWorks development, Visual Source Safe for their GUI development, and RCS for their Kernel development. Project B used CVS for their embedded VxWorks development and Visual Source Safe for their GUI development. For both projects about 75% of all the resources assigned to the project performed embedded VxWorks development.

During both projects, developers found different strengths and weaknesses of the configuration management tools they were using. Each team could classify these into one of three areas: 1) who supported the tools, 2) how easy were the tools to learn, and 3) how easy were the tools to use on a daily basis. Project A was able to use the company's Central Engineering tools support group to support ClearCase because ClearCase is a corporate standard configuration management tool. Project B found that they had to have one of the project team members support CVS. Project A found that it took several days for team members to get acquainted with ClearCase enough to start using it proficiently whereas Project B found that CVS only took a few hours. Project A found that after the ramp-up period, ClearCase was relatively easy to use with 'sand boxes' provided by the different ClearCase views. Project B had to copy files from

CVS to their own 'sand box' area and then copy the files back when it came time to integrate the files into the project. The manual process of copying files between CVS and the personnel sandbox introduced problems for Project B. When files were mis-copied it took time for Information Systems to restore he file.

In addition to the items already mentioned, the two projects found other interrelated team issues dealing with the way that they had setup their configuration management systems. One item was that Project A and Project B found that they wanted to share some common code for a particular portion of their respective projects. It turns out that since both teams were using different configuration management systems for source code, they had to port the changes back and forth as they tried to maintain a similar code base. Over the course of the projects it is estimated that this cost both projects several man-months. A second item, even though both projects were very similar, was that Project A and Project B found they had a difficult time sharing software-engineering resources because neither team fully understood the other team's environment (configuration management tools, build environment, etc).

Configuration management also affected the way both teams performed integration builds. To perform a full integration build, Project A would have to start the build in one environment and then when it was completed pass an Email (baton) to a system integrator in the next environment. Once the Email had been passed through all the environments the build would be completed. To say the least this process was not automated and builds usually occurred twice a week. The problem was exacerbated when the build process broke or when someone would get in a hurry and make a simple mistake at the second or third level and the build process would have to be started from the beginning.

Team members from Project B worked for a period of time and were able to perform daily-automated builds even though the code resided in multiple CM systems. Project B setup their build process so when it completed in one environment the files needed for the next environment would be copied to a pre-determined location (accessible by both environments) and a flag would be set. This flag was used to continue the build process in the next environment. This process worked quite successfully for Project B, as they were able to see integration problems earlier. In some cases Project B saw problems days earlier than Project A, since Project A only built twice per week. By performing the automated daily builds team members were forced to perform more rigorous testing prior to checking-in their code to ensure they did not break the automated process (no team member wanted to suffer the teasing from other team members). Project B also took advantage of the daily builds by running daily smoke tests after the build process completed which helped exercise the new code and started the testing process earlier. The smoke tests were developed by the Software Quality Engineering team and were designed to run for a short period of time (less than 20 minutes) and touch as many subsystems as possible to verify that the system met 'some' baseline expectations.

Although Project B did not actually penalize software engineers for breaking a build, Project B benefited in many other ways that could not be measured. Steve McConnell points out that Daily Builds and Smoke Tests help minimize integration risk, reduce the risk of low quality, support easier defect analysis, support progress monitoring, improve morale, and improve

customer relations. This process can provide critical control for projects in recovery mode. Daily builds and Smoke Tests can be used effectively on projects of virtually any size and complexity. [4]

The lessons learned about configuration management:

1) Use a common (and only one if possible) configuration management tool across similar projects. As both projects wanted to share common data (and in some cases resources) and perform builds across multiple configuration management systems they found it difficult to use multiple CM systems. It has been estimated that the additional effort for coordination and managing multiple CM systems cost both projects several man months. Both projects would have liked to share resources during different times of the project since both projects had high and low peaks for resources. This did not happen as effectively as it could have because of the different environments.

2) Performing automated daily builds followed by a daily smoke test is very beneficial. Daily-automated builds help the project by forcing the engineers to perform more rigorous integration testing and this helped to identify integration problems earlier. Daily smoke tests after each build help the new code base to be exercised and verify that base functionality was not broken. Even though it is not measurable you could see that Project B rejoiced when the build completed on a regular basis, by the same token you could tell when the build broke.

6.0 Change Control (Defect Tracking System)

Change control as defined by ISO-9000 is the procedure for controlling and managing changes to the entities and parts being developed and to the project. Change control can be considered a subset of configuration management. [1] For both of these projects change control was used to track and manage issues, defects, and enhancement requests for the product. A change control system could be performed using a pencil and paper, a spreadsheet, or a software application designed just for tracking issues and defects. Simply put, a change control system is a database of all the issues, defects, and enhancement requested recorded for a particular project or activity. But if used effectively a change control system is more than a system used to track defects. A change control system can be used to help predict project status and to help in the prevention of defects.

For these projects the development policy states that each project must have an on-line change control process for handling configuration changes and that this process will be documented in the project plan. The change control process shall have the capability of tracking change requests from the time they are submitted to the time they are resolved, have the ability to evaluate and review the proposed changes, and be able to document and communicate the approval/disapproval of changes. The development policy also calls out a Change Control Board which is responsible for evaluating and approving or disapproving proposed changes to configuration items and for ensuring implementation of approved items.

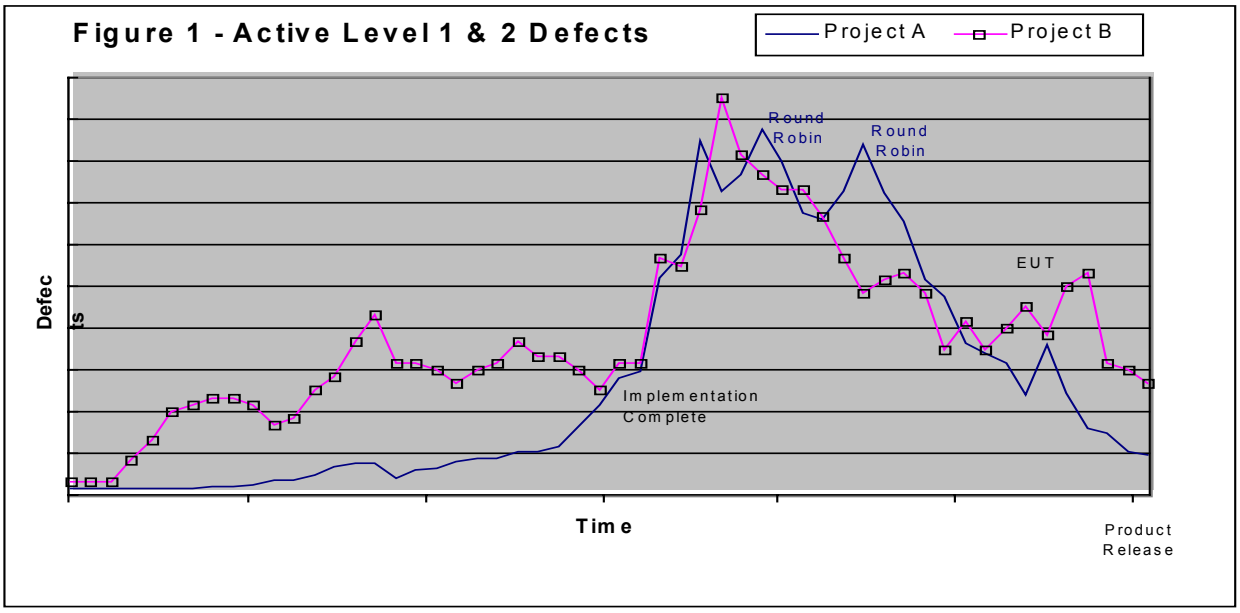
Both of these projects used Rational Software's ClearDDTS (Distributed Defect Tracking System). ClearDDTS meets all the requirements for the development policy, but the way that the tool was used differed between the two projects. ClearDDTS allowed for each project to

define the type of defect, the severity of the defect (severity 1-Fatal to severity 5-Suggestion), track which version the defect was found in, track what subsystem the defect was found in, and enter a headline and description of the defect. Once the defect was entered, the defect would be verified as an actual defect. After verification, the Software Project Leader would assign the defect to an engineer to be resolved. This is where the projects differed. When the defect was being assigned to an engineer, the Software Project Leader for Project B also entered a priority level for the defect. The priorities levels ranged from 1-Urgent to 4-Low based on how important it was for the defect to be fixed. Project A did not use priorities. The priority gave the Software Engineers for Project B a clear direction on which defects they should work on first (i.e. they could work on a severity 3 priority 1 defect before they worked on a severity 1 priority 3 defect).

Both project teams used their respective Change Control Boards (CCB) very effectively. These change control boards were required by the development policy and were used to help evaluate changing requirements and determining the impact of the defect on a customer. Marketing for both projects helped by keeping a broad market focus rather than zeroing in on one customer as the CCB reviewed the various enhancement requests. During the first part of the projects, the change control boards met periodically to review enhancement requests. Toward the end of each project the Change Control Boards met weekly to review the status of each of the defects.

Both projects used the output of the ClearDDTS to track and monitor the project's progress in resolving level 1 and level 2 defects. After all the software was fully implemented this graph was used to estimate the project's ship date. Based upon previous projects, a graph of this data should look like a bell curve. Figure 1 shows how the trend of active defects, those currently being worked on by software engineering, through the life of the project(s). Note that figure 1 has been annotated to show when software implementation was completed, when round robin testing was performed, and when expert user testing was performed.

Since figure 1 was being used to estimate the projects product release date, both projects found that early detection of defects was valuable. Early detection of defects was made possible through smoke tests (Project B), automated functional tests (Project A), and round robin testing (Project A). Round Robin testing is where each software engineer switches subsystems with another software engineer and the software engineers test one and others code. Expert User Testing is where experts from around the company are called in to use the product like actual customers. It is important to note that each time concentrated testing efforts were performed there was a significant rise in the amount of defects found verses the amount of defects resolved.



ClearDDTS also provides some of the information needed for defect prevention. In the ClearDDTS database, when a software engineer opens or resolves a defect the date and time are recorded and the software engineer is required to log in which phase the defect occurred and in which phase the defect was resolved. Some selections for "phase" include requirements, design, implementation, and not reproducible. In the future this data may prove to be very useful at determining the causes of some of the defects. For now, this data is going to require more analysis and may require some training for the software engineers so the software engineers use it in a standardized way. For example, Project B had only 4 total defects that were logged against the requirements (which seems extremely low knowing the state of Project B's requirements) while Project A had 7.9% of its defects logged against requirements. For this data to be useful we need it to make use it is accurate!

Another item that is not provided but would be useful is how long an engineer actually spent on resolving a defect. Currently, the date the defect was opened and resolved is recorded, but the actual time that the engineer spent on fixing the defect is not recorded. It would be interesting to see how much time was actually spent fixing a documentation defect that had propagated through design and implementation verses a defect that was inserted during implementation.

Some lessons learned while analyzing the Change Control process:

1) The use of a cross-functional change control board worked effectively to help balance the risk of creeping requirements, schedule, and customer needs. As enhancement requests were generated, the change control board evaluated the need based upon the customer requirements and project resources. The change control board composed of marketing, engineering and program management which provided a balance that helped minimize the effect of creeping requirements.

2) Train the project team to use the full capability of the defect tracking system. For example, use priorities as a technique to manage defects, use data to track project progress, and use the data to perform defect analysis.

7.0 Software Release Criteria

Release procedures are called out by ISO-9000. The development policy requires that the software release criteria within the Software Project Plans be met prior to the release of a product to manufacturing. Initially both projects had a loosely defined set of software release criteria documented in their software project plans.

Project B found that they wanted to perform an external software release for Beta testing. At this time project B determined that they had loosely defined their software release criteria. To perform this Beta release Project B reviewed and updated their software release criteria. Shortly after Project B completed their software release criteria, Project A also updated their software release criteria. When completed the software release criteria was the same for both Project A and Project B.

The revised software release criteria for both projects was used as the baseline to determine if the product was customer shippable. Both projects benefited from this activity by having a well-defined set of software release criteria that was agreed upon by Software Engineering, Software Quality Engineering, Marketing, and Program Management. The software release criteria included what level of quality was acceptable for shipment (or for external releases like Beta testing) and by eventually shipping a higher quality product (even if it had to have a reduced feature set).

During the late stages of these projects, the attitudes of both projects showed as both project teams were having problems meeting the release criteria. Project A's engineers were not happy with the products performance and continued to work to meet the product release criteria (this showed commitment to quality). Project B's engineers were looking at how they could reduce the product release criteria and still feel good about shipping the product (they were willing to compromise the quality of the product just to ship the product). Even though Project B did beat Project A to market (by one week), Project B had to reduce its feature set (removed broken features with agreement from the change control board) for the initial product shipment. Both products did meet the revised software release criteria.

A lesson learned about the importance of Software Release Criteria

1) Generate agreed-upon software release criteria for the project. This helps prevent (note it does not eliminate) the software engineering team from trying to ship a product before it is ready, thereby compromising the product quality. Though not mentioned, early agreement off the software release criteria helps remove most of the emotion surrounding the decision to ship the product.

Steve McConnell provides the following statement about morale killers, teamwork, and low quality, "Developers derive some of their self-esteem from the products they develop. If they develop high-quality products, they feel good. If they develop a low-quality product, they don't

feel as good. For a developer to be motivated by pride of ownership, there has to be ownership of the work, and there has to be pride in the work. A few developers can take pride in their response to the challenge of developing a low-quality product in the shorted possible time. But most developers are motivated more by quality than by sheer output."[4]

8.0 Corrective Action (Post Project Reviews)

Corrective action is a requirement of ISO-9000 which specifies that procedures must exist for non-conforming products, for identifying weaknesses in the quality system, and to show evidence that corrective actions have been identified and if appropriate, implemented. The development policy for these organizations has several items identified that meet all the ISO-9000 criteria, one of these items is the Post-Project Review (i.e. post-mortem). The Post-Project Review (PPR) as defined by the development policy is to be performed to establish corrective and/or preventive actions to improve processes, practices, and/or tools.

Both projects used the same process for conducting the post-project review. The process is a six-step process that involves planning, orientation, preparation, a project review meeting, documenting the results, and panning and implementing process improvement. [10] Even though this was the process used by both projects, the results varied.

Project A's PPR was upbeat even through the discussion of what did not work well. The team picked three issues to work on which included; upgrading tools, resolving some architectural issues, and working toward a daily build process. Project A spent a fair amount of time brainstorming how some of these problems might be addressed and then several team members volunteered to drive the issues to completion.

Project B's PPR was not quite as upbeat. A quote from the PPR was "it was hard to keep the discussion focused on issues that were representative of the team". Project B never did drive any of their software issues to completion. Instead, Project B focused on everything they should have done and did not do. Project B was not able to drive any of their issues to completion. Some of the areas they discussed were: 1) lack of written requirements was a problem (i.e. more time needed to be spent on developing requirements) 2) the Software Process was not followed (i.e. features were the first priority, no design reviews, no code reviews, no process model was evident), and 3) the software had no performance targets. Just the tone of these comments says it all!

A lesson learned about Post Project Reviews

1) Post project reviews need to not only identify what corrective actions need to be taken but also a list of actions that will be taken. Project B did an excellent job of identifying things that went wrong with the project but was not able to achieve closure on how they were going to correct them for the future.

9.0 Summary

In summary, hindsight is always 20-20. This would have been fun to run as a controlled experiment, one project doomed for success another project doomed for failure. My hope is that we can learn

from our experiences and that we can start treating best practices as project requirements when they make sense. How many times have we said that something as simple as peer reviews is a good thing but then we fail to schedule the time to do them or we just don't want to do them because they are not as fun as some of our other project activities.

Many of the lessons learned and industry best practices are actually inter-related. For example project documentation and peer reviews are inter-related. Doing the project documentation and then not doing the peer reviews does not provide the same effect as if the peer reviews had been done. By the same token, the peer review is useless without project documentation – there is nothing to review. The same could be pointed out in other areas, for example daily builds and smoke tests.

In retrospect for these two projects, the difference may not have always been measured in faster time to market or customer satisfaction, but in the number of defects released in the initial product release or in the satisfaction of the engineering team at their accomplishments. Though many of these items have been measured internally, not all of them can be measured objectively. For example how team members felt about their accomplishments is not an objective measurement. Other measurements have been made but can not be correlated. For example, project schedule slips and project costs can not be directly correlated but are most likely inter-related. We can not make a direct correlation but one fact is that Project A seemed to do a better job at following best practices and only slipped its schedule 27% as compared to the Project B's slippage of 67%.

The lessons learned throughout these projects in many cases could be classified as 'why didn't we just do it'. For these, the hope is that the lesson is learned and the organization will go off and not repeat the same thing, one such example treating our documents as living documents. Other lessons will cause the organization to extend itself beyond what they are currently doing – these lessons could be incremental steps to processes that are already working. For these, the hope is that the lessons can be incorporated into processes whether it be through planning ahead or getting the organization to update the policy to make the lesson a requirement instead of a suggestion.

So does following 'best' practices really make a difference? The best practices used by these projects are worth doing. Even though the implementation may have differed, doing the right thing seemed to always pay off in the long run.

10.0 References

[1] Charles H Schmauch, "*ISO 9000 for software developers*", Milwaukee, WI, ASQC Quality Press 1994, pp1, 13, 17.

[2] Richard H Thayer, Merline Dorfman, and Sidney C Bailin, "*Software Requirements Engineering*", Los Alamitos, CA, IEEE Computer Society Press 1997, pp1.

[3] R. Watkins, and M. Neal, "*Why and How of Requirements Tracing*", IEEE Software, July 1994, pp 104-106.

[4] Steve McConnell, "*RAPID Development, Taming Wild Software Schedules*", Microsoft Press, 1996.

[5] Suzanne Robertson and James Robertson, "*Mastering the Requirements Process*", Addison-Wesley 1999

[6] Watts S. Humphrey, "*Managing the Software Process*", Reading, MA: Addison-Wesley 1989, pp 171-178

[7] Tom Gilb and Dorothy Graham, "*Software Inspection"*, Workingham, England: Addison-Wesley 1993

[8] Glenn W Russell, "*Experience with Inspection in Ultralarge-Scale Developments*", IEEE Software, Vol. 8, No. 1, January 1991, pp 25-31

[9] Edward Kit, "*Software Testing in the Real World*", Addison-Wesley 1995, pp 42-43

[10] Rick D. Anderson and Bill Gilmore, "*Maximizing Lessons Learned – A Complete Post-Project Review Process*", Fifteen Annual Pacific Northwest Software Quality Conference, 1997.

[11] American Society for Quality Control Certified Software Quality Engineer Certification Brochure, 1996.

11.0 Glossary

CCB	Change Control Board	A designated person or group of people responsible for evaluating and approving or disapproving proposed changes to configuration items, and for ensuring implementation of approved items.
CM	Configuration Management	The discipline for applying technical and administrative direction, control and surveillance for 1) configuration identification, 2) configuration and change control, 3) configuration status accounting, and 4) configuration audit. [9]
PPR	Post Project Review	A review held at the end of a project that methodically and critically examines the development processes, practices, and tools used during the project
SQE	Software Quality Engineering	A group that performs verification and validation activities.
SQP	Software Quality Plan.	A document that identifies reviews, validation, and verification activities for a software project.

21 Project Management Success Tips[1]

Karl E. Wiegers
Process Impact
www.processimpact.com

Managing software projects is difficult under the best circumstances. Unfortunately, many new project managers receive virtually no job training. Sometimes you must rely on coaching and survival tips from people who have already done their tour of duty in the project management trenches. Here are 20 such tips for success, which I've learned from both well-managed and challenged projects. Keep these suggestions in mind during your next project, recognizing that none of them is a silver bullet for your project management problems.

Laying the Groundwork

Tip #1: Define project success criteria. At the beginning of the project, make sure the stakeholders share a common understanding of how they will determine whether this project is successful. Too often, meeting a predetermined schedule is the only apparent success factor, but there are certainly others. Some examples are increasing market share by a certain amount, reaching a specified sales volume or revenue, achieving specific customer satisfaction measures, saving money by retiring a high-maintenance legacy system, and achieving a particular transaction processing volume and correctness. Also keep your eye on team member job satisfaction, sometimes indicated by staff turnover rate and the willingness of team members to do what it takes to make the project succeed.

Remember that not all of these can be your top priority. You'll generally have to make some thoughtful tradeoff decisions to ensure that you satisfy your most important priorities. If you don't define clear priorities for your success criteria, team members can wind up working at cross purposes, leading to frustration, stress, and reduced teamwork effectiveness.

Tip #2: Identify project drivers, constraints, and degrees of freedom. Every project needs to balance its functionality, staffing, budget, schedule, and quality objectives. Define each of these five project dimensions as either a constraint within which you must operate, a driver aligned with project success, or a degree of freedom that you can adjust within some stated bounds to succeed. There's bad news: not all factors can be constraints and not all can be drivers. The project manager must have some flexibility to react to slippages, demands for increased functionality, staff turnover, or other realities. For more details about this, see Chapter 2 of my book Creating a Software Engineering Culture (Dorset House, 1996).

Tip #3: Define product release criteria. Early in the project, decide what criteria will determine whether or not the product is ready for release. You might base release criteria on:

- the number of high-priority defects still open
- performance goals being met
- specific functionality being fully operational
- customer acceptance criteria being satisfied

---

[1] This paper was originally published in *Software Development*, November 1999. It is reprinted (with modifications) with permission from *Software Development* magazine.

or other indicators that the project has met its goals. Whatever criteria you choose should be realistic, objectively measurable, documented, and aligned with what "quality" means to your customers. Decide early on how you will tell when you're done, visibly track progress toward your goals, and stick to your guns when confronted with pressure to ship before the product is ready for prime time.

Tip #4: Negotiate commitments. Despite pressure to promise the impossible, never make a commitment you know you can't keep. I practice this as a personal philosophy and it has worked well for me. Engage in good-faith negotiations with customers and managers about what is realistically achievable. Negotiation is required whenever there's a gap between the schedule or functionality the key project stakeholders demand and your best prediction of the future as embodied in project estimates. Any data you have from previous projects will help you make persuasive arguments, although there is no real defense against truly unreasonable people. Plan to renegotiate commitments when project realities (such as staff, budget, or deadlines) change, unanticipated problems arise, risks materialize, or new requirements are added. No one likes to have to modify their commitments. However, if the reality is that the initial commitments won't be achieved, let's not pretend that they will up until the moment of unfortunate truth.

Planning the Work

Tip #5: Write a plan. Some people believe the time spent writing a plan could be better spent writing code, but I don't agree. The hard part isn't writing the plan. The hard part is actually doing the planning —- thinking, negotiating, balancing, talking, asking, and listening. Actually writing the plan is mostly transcription at that point. The time you spend analyzing what it will take to solve the problem will reduce the number of surprises you have to cope with later in the project. Today's multi-site and cross-cultural development projects demand even more careful planning and tracking than do traditional projects undertaken by a co-located team.

A useful plan is much more than a schedule or work breakdown structure of tasks to perform. It also includes:

- staff, budget, and other resource estimates and plans
- project staffing, roles, and responsibilities
- assumptions, dependencies, and risks
- schedules of major milestones
- descriptions of major deliverables
- identification of the software development life cycle that you'll follow
- how the project will be tracked and monitored
- metrics that you'll collect
- how you will manage any subcontractor relationships
- how you will acquire and train the necessary staff

Your organization should adopt a standard software project management plan template, which can be tailored for various kinds of projects. An excellent starting point is IEEE Std 1058-1998, the "IEEE Standard for Software Project Management Plans", which is found in the IEEE Software Engineering Standards Collection, 1999 Edition. If you commonly tackle different kinds of projects, such as new product development and small enhancements, you might want to adopt a separate project plan template for each, rather than overburdening small projects with excessive documentation that adds little value. The project plan shouldn't be any longer or more elaborate than necessary to make sure you can successfully execute the project, but you should always take the time to write a plan.

Tip #6: Decompose tasks to inch-pebble granularity. Inch-pebbles are miniature milestones. Breaking large tasks into multiple small tasks helps you estimate them more accurately, reveals

work activities you might not have thought of otherwise, and permits more accurate, fine-grained status tracking. You can track progress based on the number of inch-pebbles that have been completed at any given time, compared to those you planned to complete by that time. Select inch-pebbles of a size that you feel you can estimate accurately. I feel most comfortable with inch-pebbles that represent tasks of about 5 to 15 labor-hours, or about one to three days in duration.

Tip #7: Develop planning worksheets for common large tasks. If your team frequently undertakes certain common tasks, such as implementing a new object class, executing a system test cycle, or performing a product build, develop activity checklists and planning worksheets for these tasks. Each checklist should include all of the steps the large task might need. These checklists and worksheets will help each team member identify and estimate the effort associated with each instance of the large task he or she must tackle. Have multiple team members participate in developing the worksheets, because people work in different ways and no one person will think of all the tasks that a group of people can collectively come up with. Using standard worksheets will help the team members adopt common processes that they can tune up as they gain experience. Tailor the worksheets to meet the specific needs of individual projects.

Tip #8: Plan to do rework after a quality control activity. I've seen project task lists in which the author seemed to assume that every testing experience will be a success that lets you proceed directly into the next development activity. However, almost all quality control activities, such as testing and technical reviews, find defects or other improvement opportunities. Your project schedule or work breakdown structure should include rework as a discrete task after every quality control activity. Base your estimates of rework time on previous experience. For example, you might have historical inspection data indicating that, on average, your developers find 30 defects per thousand lines of code by inspection and that it costs an average of 20 minutes to rework each code defect. You can crunch these kinds of numbers to come up with average expected rework effort for various types of work products. If you don't actually have to do any rework, great; you're ahead of schedule on that task. But don't count on it.

Tip #9: Manage project risks. If you don't identify and control project risks, they will control you. A risk is a potential problem that could affect the success of your project, a problem that hasn't happened yet -- and you'd like to keep it that way. Risk management has been identified as one of the most significant best practices for software development. Simply identifying the possible risk factors isn't enough. You also have to evaluate the relative threat each one poses so you can focus your risk management energy where it will do the most good. Risk exposure is a combination of the probability that a specific risk could materialize into a problem and the negative consequences for the project if it does.

To manage each risk, you can select mitigation actions to reduce either the probability or the impact. You might also want to identify contingency plans that will kick in if your risk control activities are not effective. A risk list isn't the same thing as a plan for how you will identify, prioritize, control, and track risks. Incorporate risk tracking into your routine project status tracking. Keep track of which risks materialized and which mitigation actions were effective, so the next project has smoother sailing. For a concise tutorial on software risk management, see my article "Know Your Enemy: Software Risk Management" (*Software Development*, Oct. 1998).

Tip #10: Plan time for process improvement. Your team members are already swamped with their current project assignments, but if you want the group to rise to a higher plane of software engineering capability, you'll have to invest some time in process improvement. Set aside some time from your project schedule, because software project activities should include making process changes that will help your next project be even more successful. Don't allocate 100 percent of your team members' available time to project tasks and then wonder why they don't make any progress on the improvement initiatives. Some process improvements can begin to pay off immediately, while you won't get the full return on your investment in other improvements

until the next project. View process improvement as a strategic investment in the sustained effectiveness of your development organization. I liken process improvement to highway construction: it slows everyone down a little bit for a time, but after the work is done, the road is a lot smoother and the throughput greater.

Tip #11: Respect the learning curve. The time and money you spend on training, reading and self-study, consultants, and developing improved processes are part of the investment your organization makes in project success. Recognize that you'll pay a price in terms of a short-term productivity loss when you first try to apply new processes, tools, or technologies. Don't expect to get fabulous benefits from new software engineering approaches on the first try, no matter what the tool vendor's literature or the methodology consultant's brochure claims. Instead, build extra time into the schedule to account for the inevitable learning curve. Make sure your managers and customers understand about the learning curve and accept it as an inescapable consequence of working in a rapidly changing, high-technology field.

Estimating the Project

Tip #12: Estimate based on effort, not calendar time. People generally provide estimates in units of calendar time, but I prefer to estimate the amount of effort (in labor-hours) associated with a task, then translate the effort into a calendar-time estimate. A 20-hour task might take 2.5 calendar days of nominal full-time effort, or 2 long days. However, it could also take a week, if you have to wait for critical information from a customer, or if you have to stay home with a sick child for a couple of days. The translation of effort into calendar time is based on estimates of how many effective hours I can spend on project tasks per day, any interruptions or emergency bug fix requests I might get, meetings, and all the other places into which time disappears. If you keep track of how you actually spend your time at work, you'll know how many effective weekly project hours you have on average. Typically, this is only about 50 to 60 percent of the nominal time spent at work.

Tip #13: Don't schedule people for more than 80 percent of their time. Tracking the average weekly hours that your team members actually spend working on their project assignments is a real eye-opener. The task-switching overhead associated with the many activities we are all asked to do reduces our effectiveness significantly. Excessive multitasking introduces communication and thought process inefficiencies that reduce individual productivity. I once heard a manager say that someone on his team had spent an average of ten hours per week on a particular activity, so she could do four of them at once. In reality, she'll be lucky if she can handle three such tasks. Some people multitask more efficiently than others. If some of your team members thrash when working on too many tasks at once, set clear priorities and help them succeed by focusing on just one or two objectives at a time.

Tip #14: Build training time into the schedule. Determine how much time your team members typically spend on training activities annually, and subtract that from the time available for them to work on project tasks. You probably already subtract out average values for vacation time, sick time, and other assignments; treat training time the same way. Recognize that the high-tech field of software development demands that all practitioners devote time to ongoing education, both on their own time and on the company's time. Try to arrange just-in-time training when you can schedule it, as the half-life of new technical knowledge is short unless the knowledge is put to use quickly. Training can be a team-building experience, as project team members and other stakeholders hear the same story about how to apply improved practices to their common challenges.

Tip #15: Record estimates and how you derived them. When you prepare estimates for your work, write down those estimates and document how you arrived at each of them. Understanding the assumptions and approaches used to create an estimate will make them easier to defend and

adjust when necessary, and it will help you improve your estimation process. Train the team in estimation methods, rather than simply assuming that every software developer and project leader is instinctively skilled at accurately predicting the future. Develop estimation procedures and checklists that people throughout your organization can use. An effective group estimation technique is the Wideband Delphi method (see my article "Stop Promising Miracles", *Software Development*, February 2000).

Tip #16: Use estimation tools. Many commercial tools are available to help you estimate entire projects. Based on large databases of actual project experience, these tools can give you a spectrum of possible schedule and staff allocation options. They'll also help you avoid the "impossible region," combinations of product size, team size, and schedule where no known project has been successful. The tools incorporate a number of "cost drivers" you can adjust to try to make the tool more accurately model your project, based on the technologies used, the team's experience, and other factors. Over time, you can calibrate the tool with your own project data to make it an even more reliable predictor of the future. A good tool to try is Estimate Pro from the Software Productivity Center (www.spc.ca). Others include KnowledgePlan (www.spr.com), SLIM (www.qsm.com), and Cost Xpert (www.marotz.com). You can compare the estimates from the tools with the bottom-up estimates generated from a work breakdown structure; you need to understand any major disconnects so you can arrive at the most realistic overall estimate.

Tip #17: Plan contingency buffers. Things never go precisely as you plan on a project, so your budget and schedule should include contingency buffers at the end of major phases to accommodate the unforeseen. You can use your project risk analysis to estimate the possible schedule impact if several of the risks materialize and build that projected risk exposure into your schedule as a contingency buffer.

Unfortunately, your manager or customer may view these contingency buffers as padding, rather than as the sensible acknowledgement of reality that they are. To help persuade skeptics, point to unpleasant surprises on previous projects as a rationale for your foresight. If a manager elects to discard contingency buffers, he or she has tacitly chosen to absorb all the risks that fed into the buffer and to assume that all estimates are accurate and that no unexpected events will take place. The reality on most projects is quite different. I'd rather see us deal with reality, however unattractive, than to live in Fantasyland, which leads to chronic disappointments.

Tracking Your Progress

Tip #18: Record actuals and estimates. Someone once asked me where to get historical data to improve her ability to estimate future work. My answer was, "If you write down what actually happened today, that becomes historical data tomorrow." Unless you record the actual effort or time spent on each task and compare them to your estimates, you'll never improve your estimating approach. Your estimates will forever remain guesses. Each individual can begin recording estimates and actuals, and the project manager should track these important data items on a project task or milestone basis. In addition to effort and schedule, you could estimate and track the size of the product, in units of lines of code, function points, classes and methods, GUI screens, or other units that make sense for your project.

Tip #19: Count tasks as complete only when they're 100% complete. We have a tendency to give ourselves a lot of partial credit for tasks we've begun but not fully completed: "I thought about the algorithm for that module in the shower this morning, and the algorithm is the hard part, so I'm probably about 60% done." It's difficult to accurately assess what fraction of a sizable task has actually been done at a given moment. One benefit of using inch-pebbles for task planning is that you can break a large activity into a number of small tasks and classify each small task as either done or not done -- nothing in between. Your project status tracking is then

based on the fraction of the tasks that are completed, not the fraction of each task that is completed. If someone asks you whether a specific task is complete and your reply is, "It's all done except...", then it's not done! Don't let people "round up" their task completion status; use explicit criteria to tell whether a step truly is completed.

Tip #20: Track project status openly and honestly. Create a climate in which team members feel it is safe for them to report project status accurately. Strive to run the project from a foundation of accurate, data-based facts, rather than from the misleading optimism that sometimes arises from fear of reporting bad news. Use project status information to take corrective actions when necessary and to celebrate when you can. You can only manage a project successfully when you really know what's done and what isn't, what tasks are falling behind their estimates and why, and what problems and issues remain to be tackled.

An old riddle asks, "How does a software project become six months late?" The answer is, "One day at a time." The painful problems arise when you don't know just how far behind (or, occasionally, ahead) of plan the project really is. Use project status information and metrics data to take corrective actions when necessary and to celebrate when you can. Remember the cardinal rule of software metrics: metrics data must never be used to either punish or reward individuals for their performance.

Planning for the Future

Tip #21: Conduct post-project reviews. Post-project reviews (also called postmortems or retrospectives) provide an opportunity for the team to reflect on how the last project or the previous phase went and to capture lessons learned that will help enhance your future performance. During such a review, try to identify the things that went well, so you can create an environment that enables you to repeat those success contributors. Also look for things that didn't go so well, so you can change your approaches and prevent those problems in the future. In addition, think of things that happened that surprised you, as these might be risk factors for which you should be alert on the next project.

Post-project reviews should be conducted in a constructive, honest atmosphere, rather than being an opportunity to place blame for previous problems. Capture the lessons learned from each review and share them with the entire team and organization, so all can benefit from your painfully gained experience. I like to write lessons learned in a neutral way, such that it isn't obvious whether we learned the lesson because we did something right or because we made a mistake.

These tips won't guarantee success, but they will help you get a solid handle on your project and ensure that you're doing all you can to make it succeed in a crazy and unpredictable world.

21 Project Management Success Tips

Karl E. Wiegers

Process Impact

www.processimpact.com

21 Project Management Success Tips

Agenda

- 4 tips for Laying the Foundation
- 7 tips for Planning the Project
- 6 tips for Estimating the Work
- 3 Tips for Tracking Your Progress
- 1 Tip for Learning for the Future

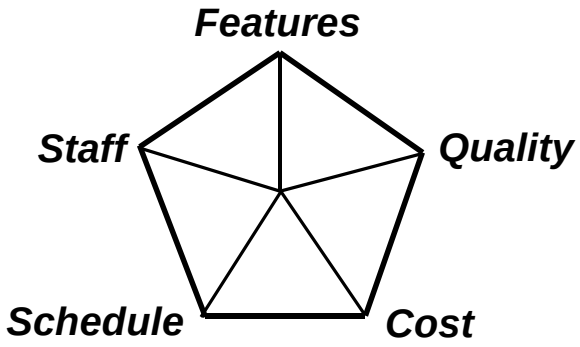
Laying the Foundation: Tip #1

- **Define project success criteria.**
  - ✓ Agree on how stakeholders will determine success.
    - meeting schedule or budget
    - customer satisfaction
    - team member satisfaction
    - market share
    - sales revenue
    - cost savings

  - ✓ Set priorities that lead to success.
  - ✓ Not everything can be top priority!

Laying the Foundation: Tip #2

- **Identify project drivers, constraints, and degrees of freedom.**



[Wiegers, Karl E. *Creating a Software Engineering Culture*. Dorset House, 1996]

Laying the Foundation: Tip #3

- **Define product release criteria.**
  - ✓ How do you know when you're done?
  - ✓ Should be objective and measurable.
  - ✓ Decide early, track progress, and don't abandon
  - ✓ Some possibilities:

- number of open high-priority defects
- performance measurements
- specific functionality fully operational
- customer acceptance criteria satisfied
- reliability goals satisfied
- legal or regulatory compliance

Laying the Foundation: Tip #4

- **Negotiate commitments.**

  - ✓ Negotiate in good faith with customers, managers, and team members.
  - ✓ Base plans on what is realistically achievable.
  - ✓ Use data to support your case.
  - ✓ Renegotiate when requirements or realities change.

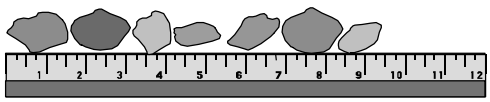
Never make a commitment you know you can't keep!

Planning the Project: Tip #1

- **Write a plan.**
  - ✓ A plan is more than a schedule.
  - ✓ The hard part is thinking, asking, listening, and thinking -- not writing.
  - ✓ Planning helps avoid the surprise factor.
  - ✓ Record assumptions, constraints, dependencies.
  - ✓ Cross-cultural and cross-time zone projects need even better planning.
  - ✓ Adapt a standard project plan template to each project.

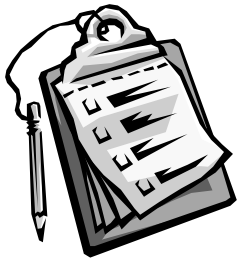
Planning the Project: Tip #2

- **Decompose tasks to inch-pebble granularity.**
  - ✓ Inch-pebbles are miniature milestones.
  - ✓ Estimates are often low because of overlooked tasks.
  - ✓ It's easier to estimate and track small tasks.
  - ✓ Inch-pebbles are typically 1-3 days in duration.
  - ✓ Track progress by completion of inch-pebbles.



Planning the Project: Tip #3

- **Develop planning worksheets for common large tasks.**
  - ✓ List the steps you might perform for common tasks.
    - implementing a class
    - conducting a system test
    - building a release
  - ✓ Develop worksheets collaboratively.
    - no one person will think of all the steps
    - people work in different ways
  - ✓ Use worksheets to plan and estimate future tasks.



Planning the Project: Tip #4

- **Plan for rework after a quality control task.**
  - ✓ Tests and reviews almost always lead to rework.
  - ✓ Put “Rework” tasks in your work breakdown structure and schedule.
  - ✓ If you don’t plan for rework, your schedule will slip.
  - ✓ If no rework is needed, you’re ahead of schedule!
  - ✓ Base rework estimates on previous experience

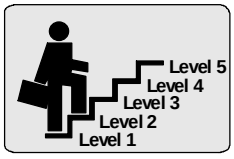
Planning the Project: Tip #5

- **Manage project risks.**
  - ✓ During project planning:
    - brainstorm project risks
    - analyze and prioritize
    - mitigate, prevent, or avoid greatest risks
  - ✓ During project tracking:
    - monitor mitigation action implementation
    - monitor mitigation effectiveness
    - re-evaluate top 10 risk list periodically
    - look for new risks

[Wiegers, Karl. "Know Your Enemy: Software Risk Management," *Software Development*, 11/98]

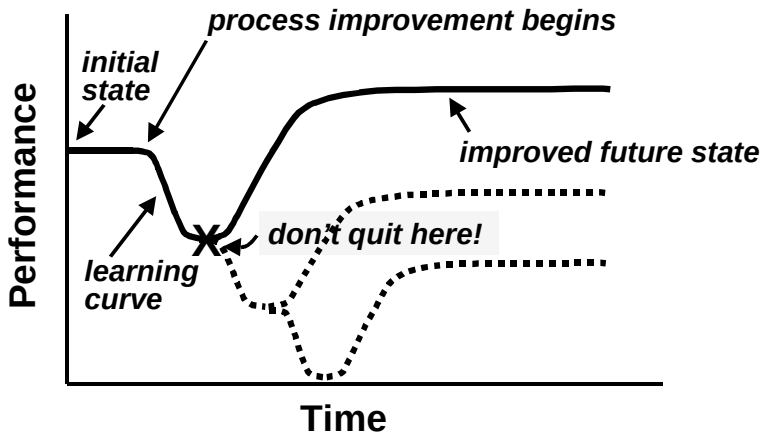
Planning the Project: Tip #6

- **Plan time for process improvement.**
  - ✓ Process improvement is part of project overhead.
  - ✓ Process improvement is a strategic investment.
  - ✓ Decide how much time and $$ you want to invest.
  - ✓ Don't allocate project work at 100% and hope process improvement happens.
  - ✓ Track process improvement effort and progress.



Planning the Project: Tip #7

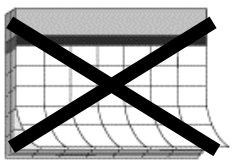
■ **Respect the learning curve.**



Estimating the Work: Tip #1

■ **Estimate based on effort, not calendar time.**

- ✓ Estimate effort for uninterrupted work by 1 person.
- ✓ Account for average effective project hours/week.
  - typically 50-60% of nominal time
  - measure your own results for best estimates
- ✓ Account for vacation, sickness, training, other planned activities.
- ✓ Translate effort estimates into calendar time.

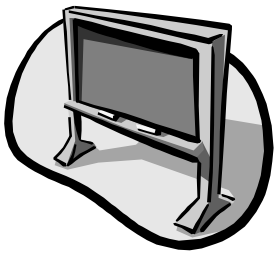


Estimating the Work: Tip #2

- **Don't schedule people for more than 80 percent of their time.**
  - ✓ Task-switching overhead reduces efficiency.
  - ✓ Some people task-switch better than others.
    - understand how each team member works best
    - prioritize tasks to finish first things first
  - ✓ More human interfaces consume more communication effort.

Estimating the Work: Tip #3

- **Build training time into the schedule.**
  - ✓ Training the team is part of project overhead.
  - ✓ Try for just-in-time training.
  - ✓ Subtract training time from available resources.
  - ✓ Don't forget the learning curve!

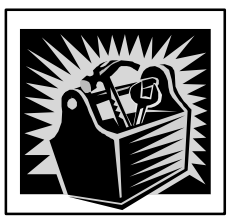


Estimating the Work: Tip #4

- **Record estimates and how you derived them.**
  - ✓ If you don't record estimates, what do you track against?
  - ✓ Adopt estimation procedures and heuristics.
  - ✓ Train the team in estimation.
  - ✓ Document your estimating method.
  - ✓ Compare estimates from more than one method.
  - ✓ Record your assumptions.
  - ✓ Get reality check from other team members.
  - ✓ Use Wideband Delphi for group estimation.

Estimating the Work: Tip #5

- **Use estimation tools.**
  - ✓ Tools relate product size, effort, team size, schedule.
  - ✓ Models are based on data from completed projects.
  - ✓ Cost drivers permit project-specific adjustments.
  - ✓ Tools can keep you out of the ***Impossible Region***.
  - ✓ Some tools:
    - Estimate (www.construx.com)
    - Estimate Professional (www.spc.ca)
    - KnowledgePlan (www.spr.com)
    - SLIM (www.qsm.com)
    - Cost Xpert (www.marotz.com)



Estimating the Work: Tip #6

- **Plan contingency buffers.**
  - ✓ Life never goes exactly as planned.
  - ✓ Build contingency time into each phase's schedule.
  - ✓ Managers might scrap those buffers.
    - point out surprises from previous projects
    - ask why this time will be different
    - create defensible buffers based on experience, data, or risk analysis
  - ✓ Try critical chain project management.

E, Schragenheim, and H. W. Dettmer, "Does Your Internal Management Meet Expectations?", *CrossTalk,* April 2001.

Tracking Your Progress: Tip #1

- **Record estimates and actuals.**
  - ✓ Data lets you improve estimation accuracy.
  - ✓ Record estimated and actual effort and schedule.
    - individual tasks
    - project major milestones
    - overall product cycle time
  - ✓ Improve your estimates with experience, not faith.
  - ✓ Estimates that aren't based on data are guesses.

Tracking Your Progress: Tip #2

- **Count tasks as complete only when they're 100% complete.**
  - ✓ We give ourselves a lot of "partial credit."
  - ✓ It's hard to accurately assess percent completion.
  - ✓ Large tasks are 90% complete for a long time!
  - ✓ Inch-pebbles support accurate completion tracking.
  - ✓ Use objective completion criteria.
  - ✓ Use earned value to track what you've really done.

Tracking Your Progress: Tip #3

- **Track project status openly and honestly.**
  - ✓ Commit to accurate status tracking at all times.
  - ✓ Create a safe environment for being honest.
  - ✓ Blame the process, not the people.
  - ✓ Learn about problems as soon as you can.
  - ✓ Take corrective actions as soon as you can.
  - ✓ Be straight with customers and managers.
  - ✓ Base status changes on exit criteria.

Learning for the Future: Tip #1

- **Conduct post-project reviews.**
  - ✓ Could also hold post-phase reviews.
  - ✓ Look for:
    - what went well (repeat it!)
    - what didn't go so well (change it!)
    - what happened that surprised you (manage risks!)
  - ✓ Record lessons learned for future reference.
    - write lessons neutrally
    - make lessons available to the whole organization

[Karl Wiegers and Johanna Rothman. "Looking Back, Looking Ahead," *Software Development*, 2/01]

21 Project Management Success Tips

- These aren't silver bullets.
- Success comes from good practices and processes.
- Permeate everything with communication.
- Learn from past experience, co-workers, and industry literature.
- Commit to 2 personal improvement areas on each project or phase.

Project Management References

Brown, Norm. “Industrial-Strength Management Strategies.” *IEEE Software* (July 1996), pp. 94-103.

Ensworth, Patricia. *The Accidental Project Manager: Surviving the Transition from Techie to Manager*. New York: John Wiley & Sons, 2001.

Paulk, Mark, et al. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, Mass.: Addison-Wesley, 1995.

Kerth, Norman L. *Project Retrospectives*. New York: Dorset House Publishing, 2001.

McConnell, Steve. *Software Project Survival Guide*. Redmond, Wash.: Microsoft Press, 1998.

Putnam, Lawrence H., and Ware Myers. *Industrial Strength Software: Effective Management Using Measurement*. Los Alamitos, Calif.: IEEE Computer Society Press, 1997.

Whitten, Neal. *Managing Software Development Projects: Formula for Success, 2nd Ed.* New York: John Wiley & Sons, 1995.

Wiegers, Karl E. *Creating a Software Engineering Culture*. New York: Dorset House Publishing, 1996.