TWENTY-SIXTH ANNUAL
PACIFIC NORTHWEST
SOFTWARE QUALITY CONFERENCE

October 14-15, 2008

Oregon Convention Center
Portland, Oregon

TABLE OF CONTENTS

Keynote Address – October 14

Keynote Address – October 15

Collaboration Track I – October 14

Collaboration Track II – October 14

Testing Track – October 14

Assortment Track – October 14

Collaboration Track – October 15

Management Track – October 15

Testing Track – October 15

Assortment Track – October 15

Welcome to the Pacific Northwest Software Quality Conference!

So…why did the 2008 PNSQC board and volunteers select "collaborative quality" as our theme this year? Because we felt collaboration provides a key competitive advantage as it relates to software quality and we want to bring topics to you that are relevant and provide you with skills you can take back and use right away.

This year we are pleased to have Sam Kaner kick-off the technical conference. He is one of the world's leading experts in multi-party collaboration. As our first keynote speaker, he will be sharing with us what it takes to build a collaborative team in a diverse working atmosphere. Also, we have Ron Jeffries and Chet Hendrickson here on Wednesday collaborating as keynote speakers to explore the dynamic behavior of a team working on a software project.

If you have been to our conference in the past, we have kept the format of multiple tracks, giving you plenty of choices throughout the two day conference. We have also maintained the 45-minute presentation format, and this year added 10 minutes between sessions to ensure presenters and attendees have plenty of time to get settled and ready for the next topic.

Tuesday night we have a reception in the exhibit hall and directly afterwards we have the pleasure of collaborating with the Rose City Software Process Improvement Network (SPIN) and with Tamara Sulaiman where she will share her Agile Planning Framework with us.

As we have done in the past, during lunch on Tuesday we have Birds of a Feather table discussions and on Wednesday, we have a distinguished panel gathering to discuss "Collaborative Quality: Does it Help?"

To close the conference, we are asking our attendees to vote for the best paper, we hope you will join us to share your impressions and mark your ballot. Every year we strive to put together the best software conference on the West Coast. We believe we have done that this year, let us know what you think.

Debra Lavell

President, PNSQC

BOARD MEMBERS, OFFICERS, and COMMITTEE CHAIRS

Debra Lavell– Board Member & President
Intel Corporation

Keith Stobie – Board Member & Vice President
Microsoft

David Butt – Board Member & Secretary

Doug Reynolds – Board Member & Treasurer
Tektronix, Inc.

Ian Savage – Board Member, Conference Chair & Program Chair
McAfee, Inc.

Bill Gilmore – Board Member & Operations and Infrastructure Chair

Marilyn Pelerine – Board Member & Audit Chair
Symetra Financial

Chris Blain – Exhibits Chair

David Chandler – Publicity Chair
Nationwide Insurance

Esther Derby – Keynote, Invited Speaker & Workshop Chair
Esther Derby Associates

Ellen Ghiselli – Volunteer Chair
McAfee, Inc.

Shauna Gonzales – Luncheon Program Chair
Nike, Inc.

Patt Thomasson – Communications Chair
McAfee, Inc.

Claire Williams – Networking Chair
Tektronix, Inc.

CONTRIBUTING VOLUNTEERS

Darrel Bonner
Apcon, Inc.

Kris Bugbee
McAfee, Inc.

Robert Cohn

Paul Dittman
McAfee, Inc.

Moss Drake
Daisy Dental

Joshua Eisenberg
McAfee, Inc.

Cynthia Gens
Sabrix, Inc.

Sara Gorsuch
Symetra Financial

Frank Goovaerts
Tektronix, Inc.

Les Grove
Tektronix, Inc.

Brian Hansen
Radar Engineers, Inc.

Kathleen Iberle
Hewlett Packard

Dave Liesse
SS&C Technologies, Inc.

Roslyn Lindquist
Windriver

Launi Mead
Symetra Financial

Jonathan Morris
Kronos, Inc.

Debra Paynter
Web Trends

Ganesh Prabhala
Intel Corporation

Rob Ranslam
Windriver

Mike Roe
Symyx

Rhea Stadick
Intel Corporation

Wolfgang Strigel
Strigel Consulting

Richard Vireday
Intel Corporation

Doug Whitney
McAfee, Inc.

The Art of Building Consensus

Sam Kaner

Making high-quality, high-stake decisions in groups is not easy. And making them in cross-functional groups is even tougher. The diversity in the room breeds misunderstanding, confusion, and frustration. All too often these meetings end with predictably mediocre results – people either accept lowest-common-denominator compromises, or they punt the tough issues to a senior person, so s/he can make the real decisions later. In both cases, one is left wondering, “why call such meetings in the first place?”

This morning’s keynote is a fascinating tour de force description of what it takes to build consensus in real-world cross-functional environments. Sam Kaner, one of the world’s leading experts in multi-party collaboration, will share models and methods that have been used successfully at HP, Symantec, Electronic Arts, VISA, and hundreds of other organizations. You’ll walk away with powerful new insights and a set of tools you can use right away.

Sam Kaner, Ph.D., has been named as "one of the world's leading experts in collaboration" (Sandor Schuman, Ph.D., founding editor of Journal of Group Facilitation, and co-founder of International Association of Facilitators.) Sam’s classic bestseller, “Facilitator's Guide to Participatory Decision-Making” (Jossey Bass), has gone through 16 printings and is now in its 2nd edition. Sam has been a featured speaker at more than 40 professional conferences, and he has delivered keynote addresses on collaboration and group decision-making at the annual World Congress of Quality, the annual Asia Facilitators' Conference, the annual world conference of the International Facilitators' Association, The Association of Quality & Participation, and the annual Best in the West conference of the Organization Development Network. In 2005, AmericaWest Airlines named Sam as one of America's Best Consultants. Since 1987 he has been Executive Director of Community At Work, a San Francisco-based consulting firm that specializes in designing and facilitating collaborative approaches to complex system change.

Quality Dynamics of Agile SW Development

Ron Jeffries & Chet Hendrickson

As Agile software proponents, we have spent much of our time explaining XP and Agile practices and why they make sense. Generally we talk about these things from a “supply side” viewpoint. We think about software development and how it works best, from the trenches.

Let’s focus on the “demand” side. Let’s look at the needs of those who pay for our software development. They need benefits, profit, information, and flexibility. It turns out that in order to provide what the business side needs, Agile and XP practices are not just helpful - they are almost essential.

Starting from a few simple and commonly held assumptions, we will explore the dynamic behavior of a software project, and will derive both management practices, and technical practices, as the inevitable consequences of setting out to do with what our business-side people need and want.

This keynote is a start at creating a unified theory of team-based software development, deriving the practices that are necessary in order to do software profitably and well. Our presentation will be based around a growing series of graphs and pictures illustrating what happens on a software project. Relationships between practices - what we do -and what happens - will be shown with both static and dynamic charts.

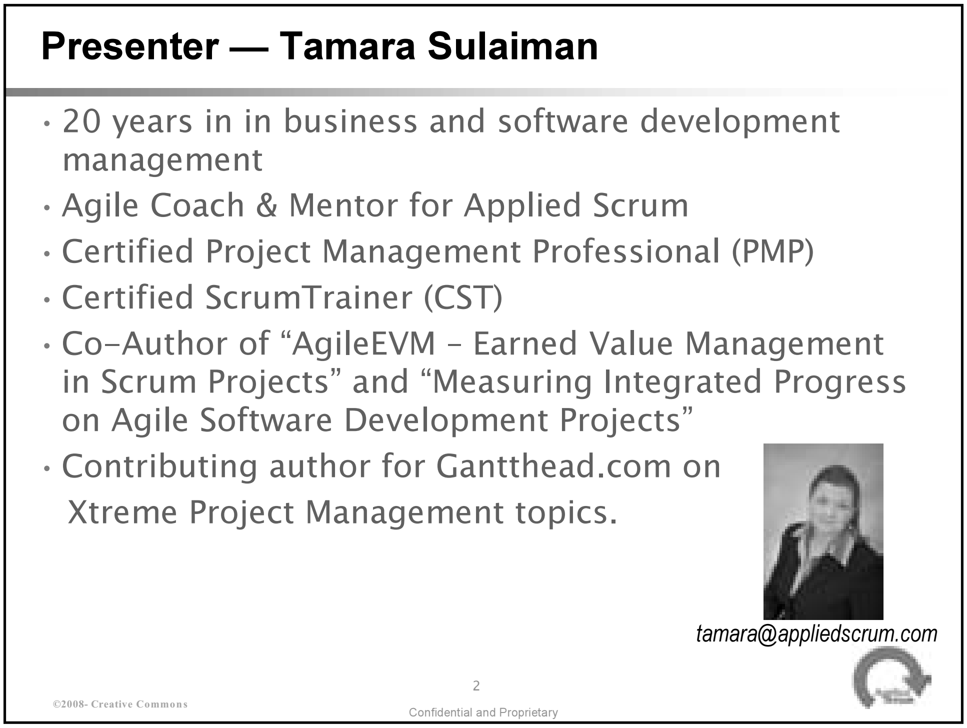
Ron Jeffries *is author of Extreme Programming Adventures in C#, the senior author of Extreme Programming Installed, and was the on-site XP coach for the original Extreme Programming project. Ron has been involved with Extreme Programming for over five years, presenting numerous talks and publishing papers on the topic. He is the proprietor of www.XProgramming.com, a well-known source of XP information. Ron was one of the creators, and a featured instructor in Object Mentor's popular XP Immersion course. He is a well-known independent consultant in XP and Agile methods.*

Ron has advanced degrees in mathematics and computer science, and has been a systems developer for more years than most of you have been alive. His teams have built operating systems, compilers, relational database systems, and a large range of applications. Ron's software products have produced revenue of over half a billion dollars, and he wonders why he didn't get any of it.

Chet Hendrickson *was at the ground zero of Extreme Programming, the Chrysler Comprehensive Compensation (C3) system. As a developer on the pre-XP C3, Chet saw how poor communication, inadequate testing, and an overly complex design can doom a development effort. He helped make the decision to throw away 14 months of work and begin again under the guidance of Kent Beck, Martin Fowler, and Ron Jeffries. Chet, along with Jim Haungs and Rich Garzaniti, in a talk at OOPSLA'97, was the first to report on the "Chrysler Methodology" as the term Extreme Programming had not yet been coined. Chet is an independent consultant, helping software teams improve the software development process by the application of XP's core values of simplicity, communication, feedback, and courage. His clients have ranged from federally charted quasi-public financial institutions to the developers of real-time petroleum exploration equipment. He is an author of Extreme Programming Installed. The book, the second in the Extreme Programming series, consists of a connected collection of essays, presented in the order the practices would actually be implemented during a project. He and Ron Jeffries are the proprietors of agilesoftwaredevelopment.org.*

Expanding Trust and Collaboration with Agile Earned Value Management

Confidential and Proprietary



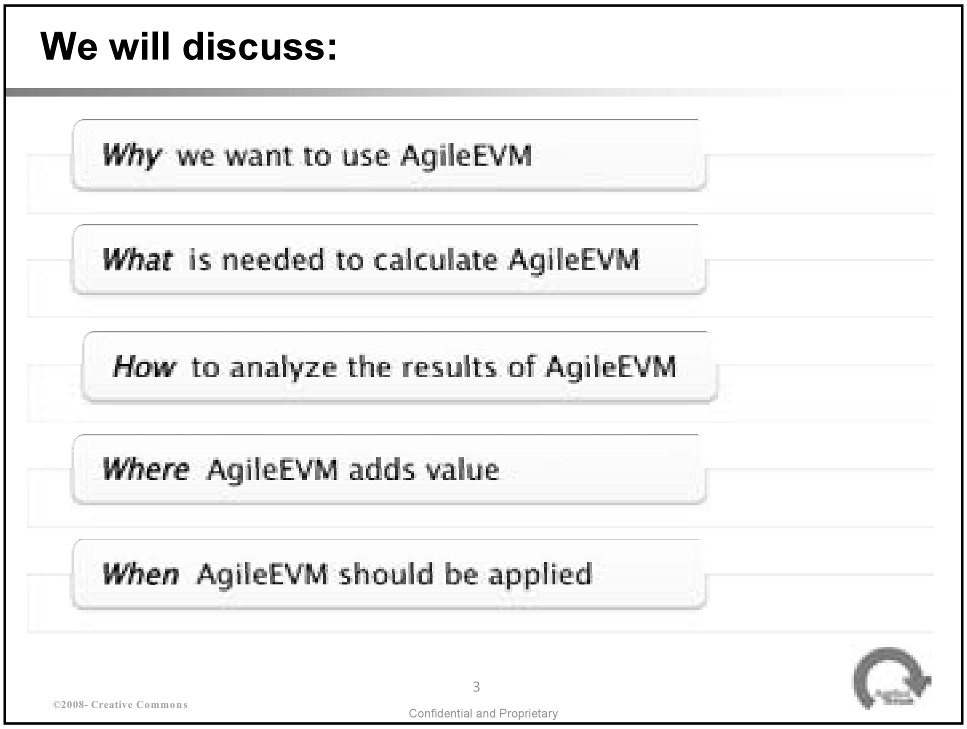
Presenter — Tamara Sulaiman

- 20 years in in business and software development management
- Agile Coach & Mentor for Applied Scrum
- Certified Project Management Professional (PMP)
- Certified ScrumTrainer (CST)
- Co-Author of “AgileEVM – Earned Value Management in Scrum Projects” and “Measuring Integrated Progress on Agile Software Development Projects”
- Contributing author for Gantthead.com on Xtreme Project Management topics.

tamara@appliedscrum.com

2

Confidential and Proprietary

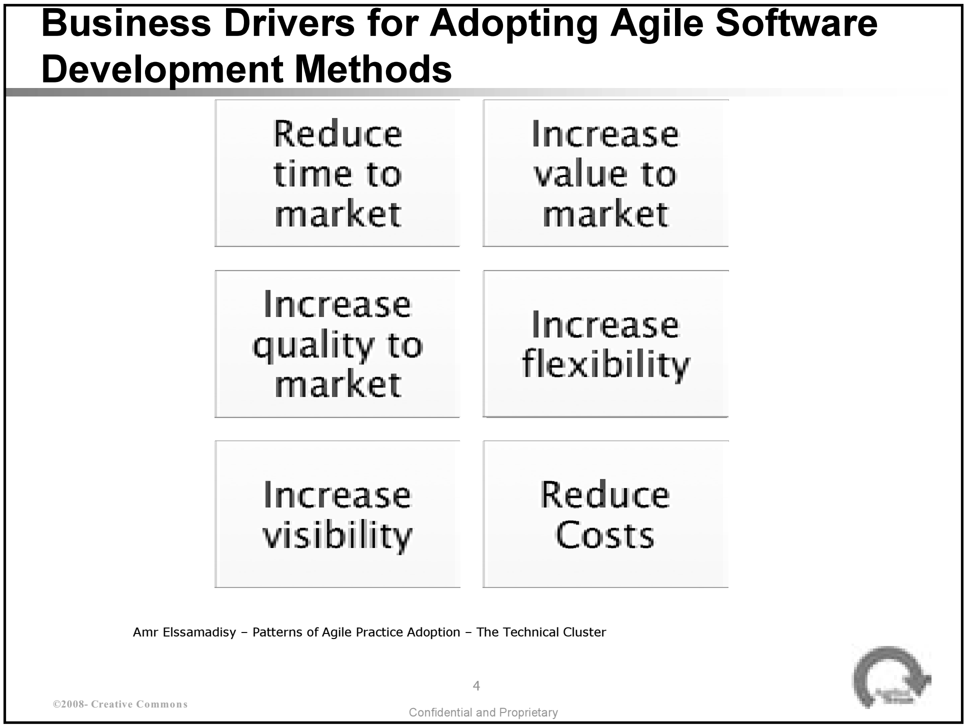


We will discuss:

- *Why* we want to use AgileEVM
- *What* is needed to calculate AgileEVM
- *How* to analyze the results of AgileEVM
- *Where* AgileEVM adds value
- *When* AgileEVM should be applied

3

Confidential and Proprietary



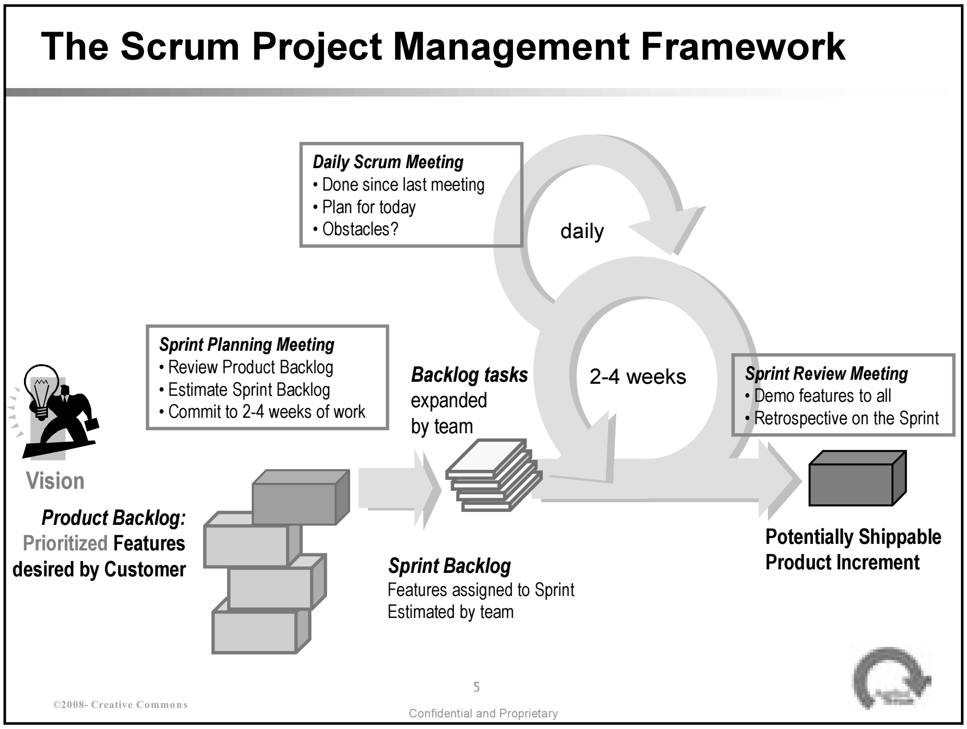
Business Drivers for Adopting Agile Software Development Methods

Reduce time to market	Increase value to market
Increase quality to market	Increase flexibility
Increase visibility	Reduce Costs

Amr Elssamadisy – Patterns of Agile Practice Adoption – The Technical Cluster

4

Confidential and Proprietary



The Scrum Project Management Framework

Daily Scrum Meeting
- Done since last meeting
- Plan for today
- Obstacles?

daily

Sprint Planning Meeting
- Review Product Backlog
- Estimate Sprint Backlog
- Commit to 2-4 weeks of work

Backlog tasks expanded by team

2-4 weeks

Sprint Review Meeting
- Demo features to all
- Retrospective on the Sprint

Vision

Product Backlog: **Prioritized Features desired by Customer**

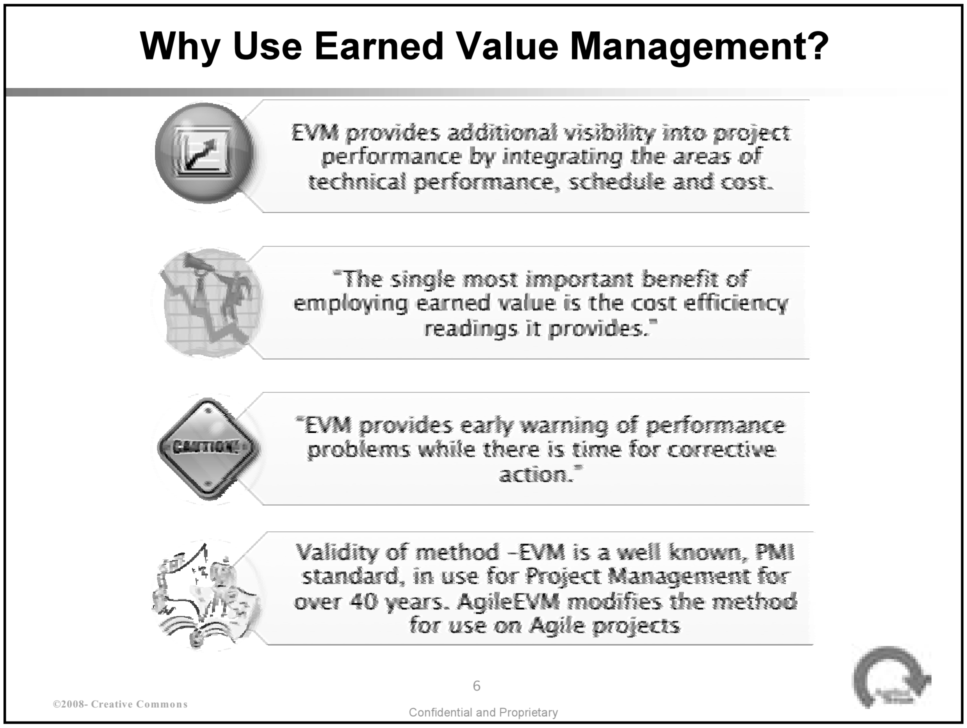
Sprint Backlog
Features assigned to Sprint
Estimated by team

Potentially Shippable Product Increment

5

©2008- Creative Commons

Confidential and Proprietary



Why Use Earned Value Management?

EVM provides additional visibility into project performance by integrating the areas of technical performance, schedule and cost.

"The single most important benefit of employing earned value is the cost efficiency readings it provides."

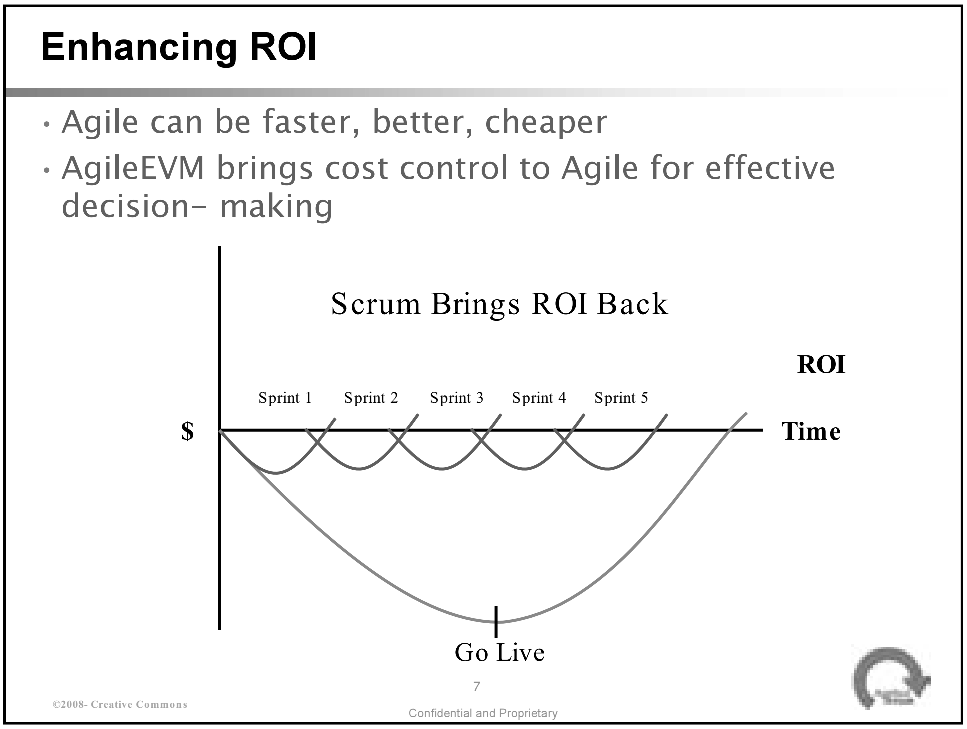
"EVM provides early warning of performance problems while there is time for corrective action."

Validity of method –EVM is a well known, PMI standard, in use for Project Management for over 40 years. AgileEVM modifies the method for use on Agile projects

6

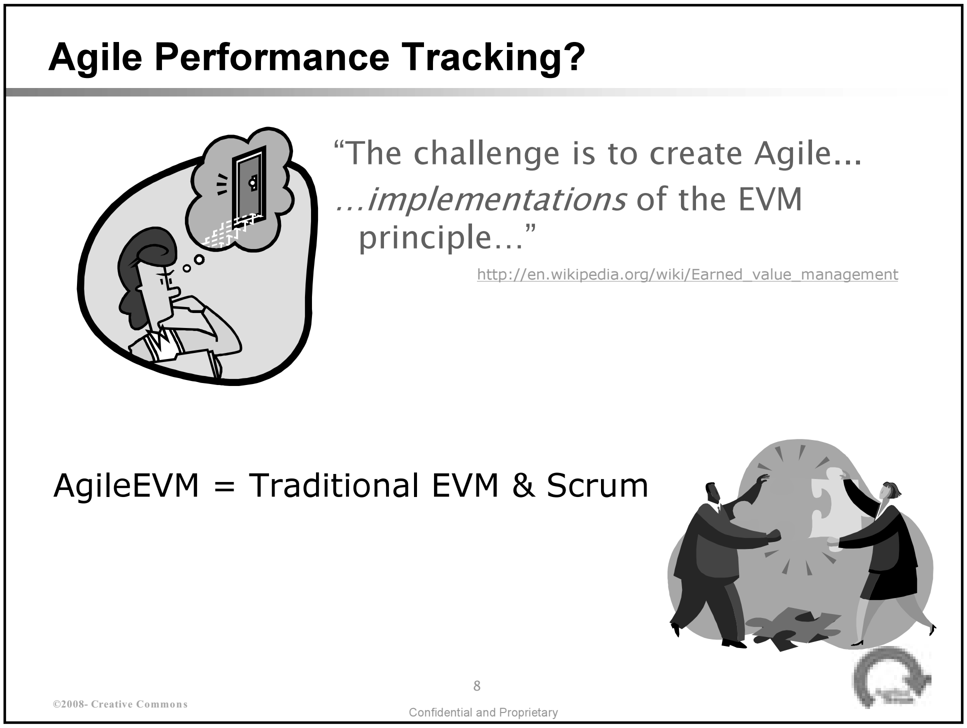
©2008- Creative Commons

Confidential and Proprietary



Enhancing ROI

- Agile can be faster, better, cheaper
- AgileEVM brings cost control to Agile for effective decision- making



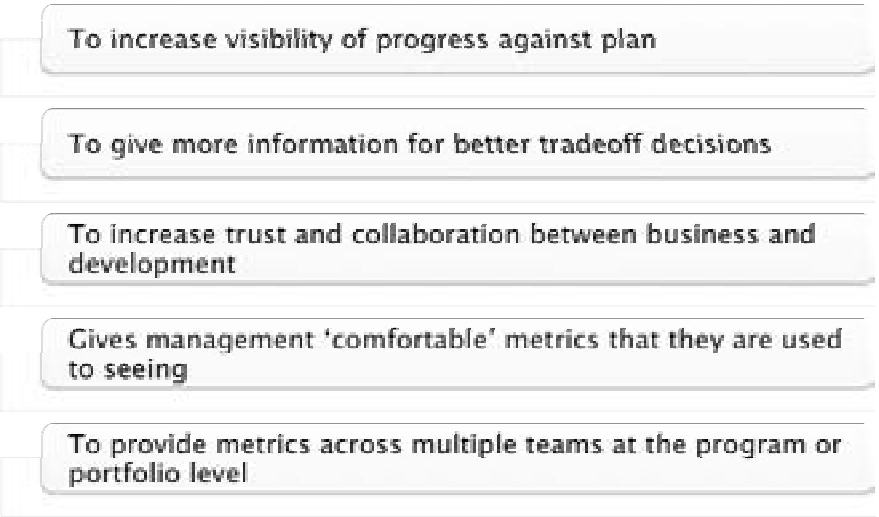
Agile Performance Tracking?

"The challenge is to create Agile...
...implementations of the EVM principle..."

http://en.wikipedia.org/wiki/Earned_value_management

AgileEVM = Traditional EVM & Scrum

Why Use AgileEVM?



To increase visibility of progress against plan

To give more information for better tradeoff decisions

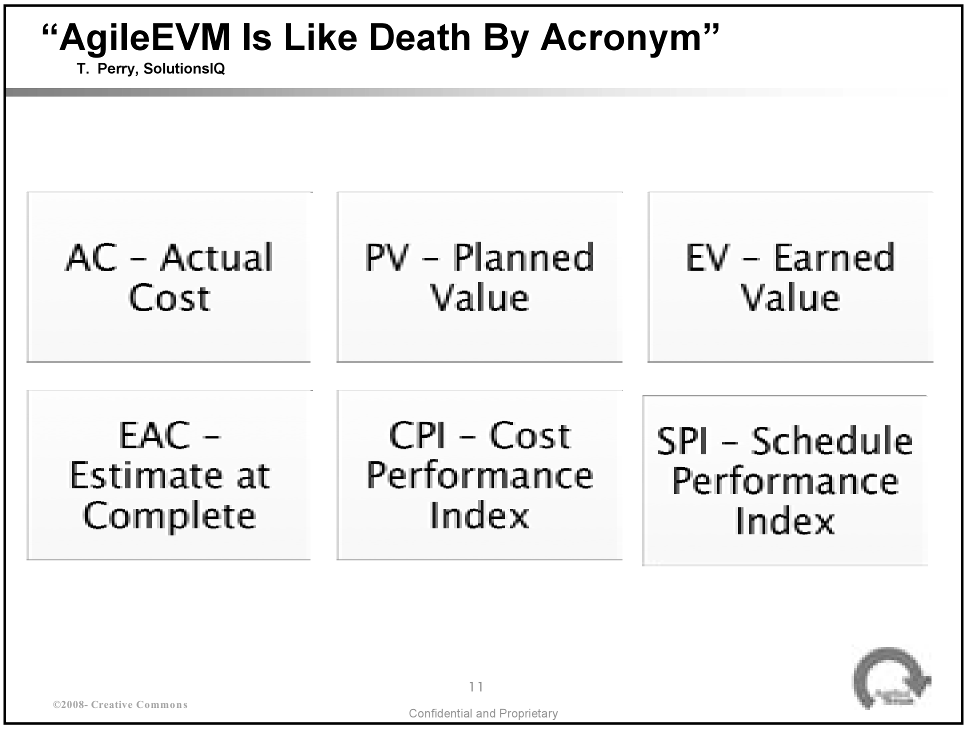
To increase trust and collaboration between business and development

Gives management 'comfortable' metrics that they are used to seeing

To provide metrics across multiple teams at the program or portfolio level

A Couple of Notes:

- Earned Value Metrics track progress against plan; they do not track earned business value.
- AgileEVM does not replace Agile Burn-down, Burn-up charts, or any other metric you are currently using.



“AgileEVM Is Like Death By Acronym”

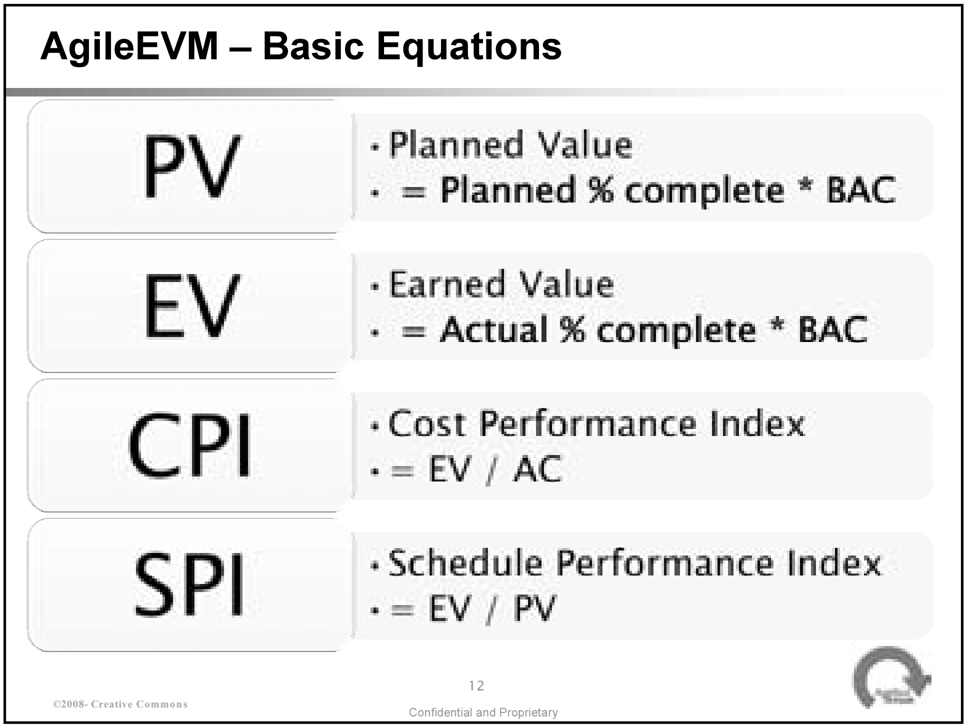
T. Perry, SolutionsIQ

AC – Actual Cost	PV – Planned Value	EV – Earned Value
EAC – Estimate at Complete	CPI – Cost Performance Index	SPI – Schedule Performance Index

©2008- Creative Commons

11

Confidential and Proprietary



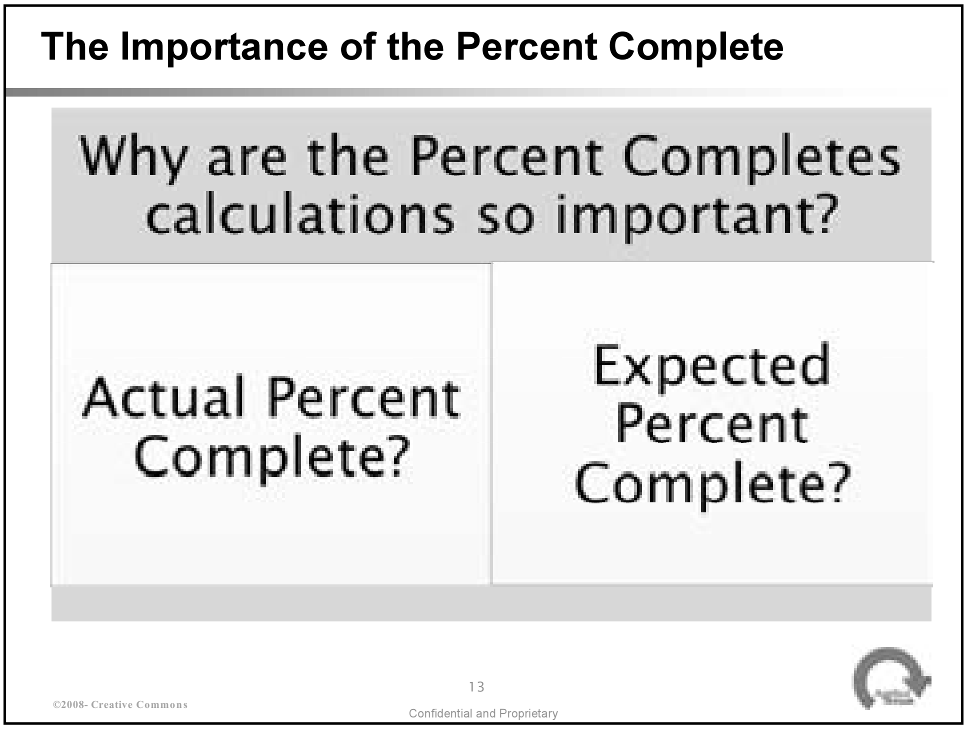
AgileEVM – Basic Equations

PV	• Planned Value • = Planned % complete * BAC
EV	• Earned Value • = Actual % complete * BAC
CPI	• Cost Performance Index • = EV / AC
SPI	• Schedule Performance Index • = EV / PV

©2008- Creative Commons

12

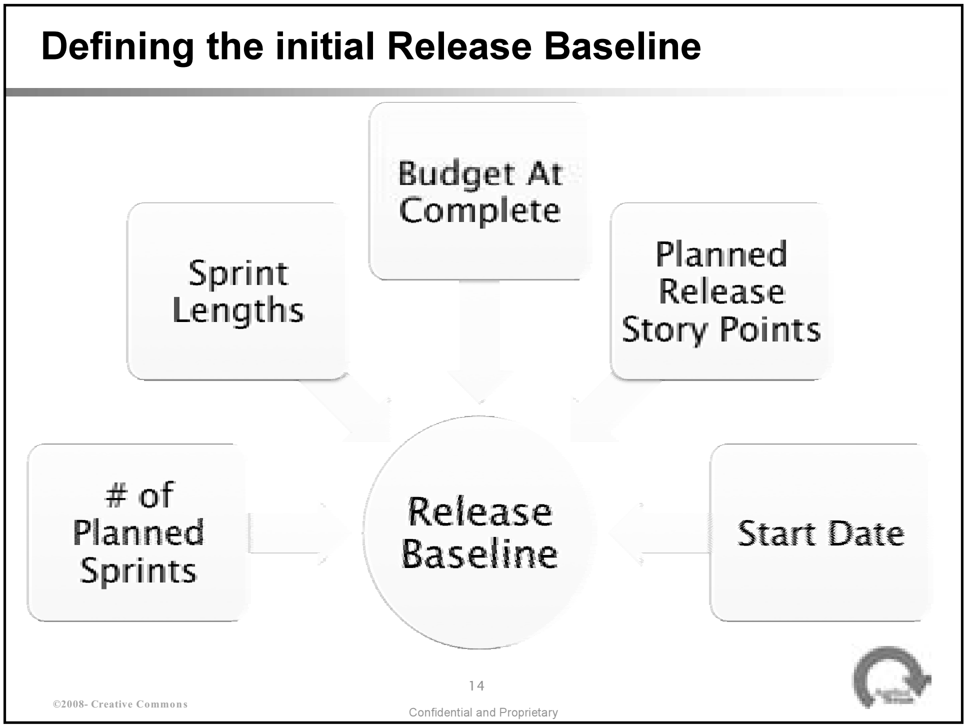
Confidential and Proprietary



The Importance of the Percent Complete

Why are the Percent Completes calculations so important?

Actual Percent Complete?	Expected Percent Complete?



Defining the initial Release Baseline

Budget At Complete

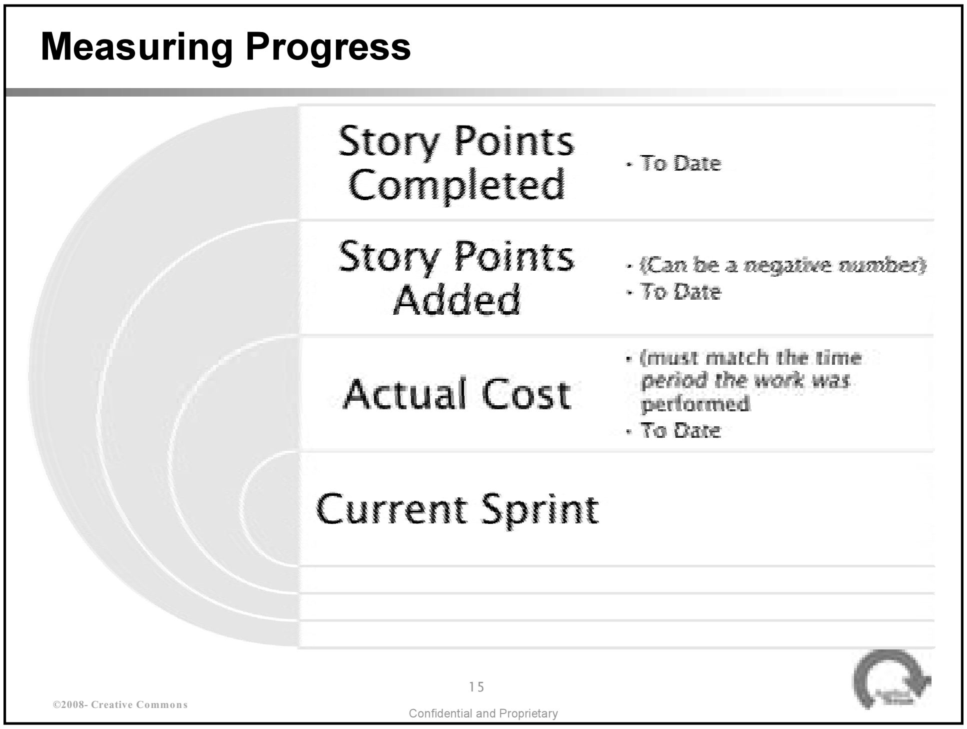
Sprint Lengths

Planned Release Story Points

# of Planned Sprints

Release Baseline

Start Date



Measuring Progress

Story Points Completed

- To Date

Story Points Added

- (Can be a negative number)
- To Date

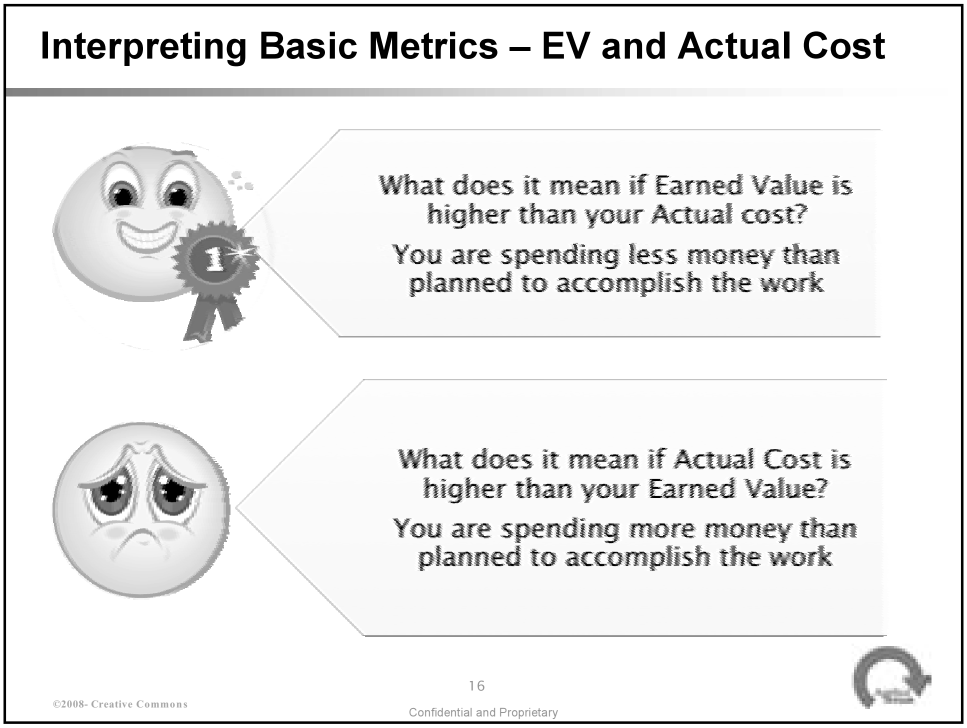
Actual Cost

- (must match the time period the work was performed
- To Date

Current Sprint

15

Confidential and Proprietary



Interpreting Basic Metrics – EV and Actual Cost

What does it mean if Earned Value is higher than your Actual cost?

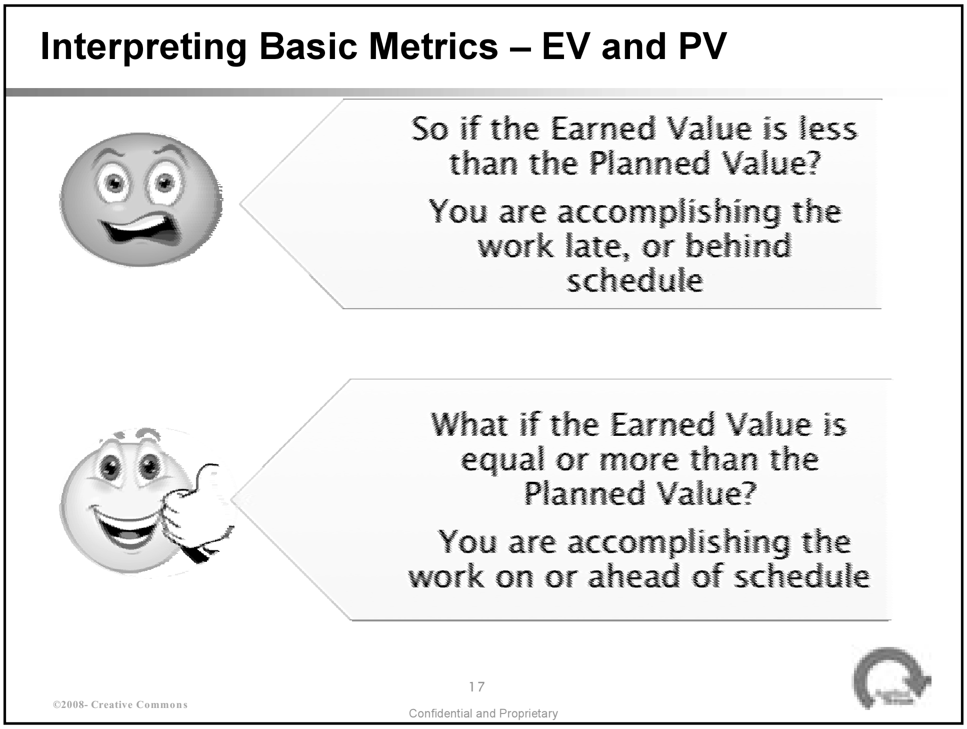
You are spending less money than planned to accomplish the work

What does it mean if Actual Cost is higher than your Earned Value?

You are spending more money than planned to accomplish the work

16

Confidential and Proprietary



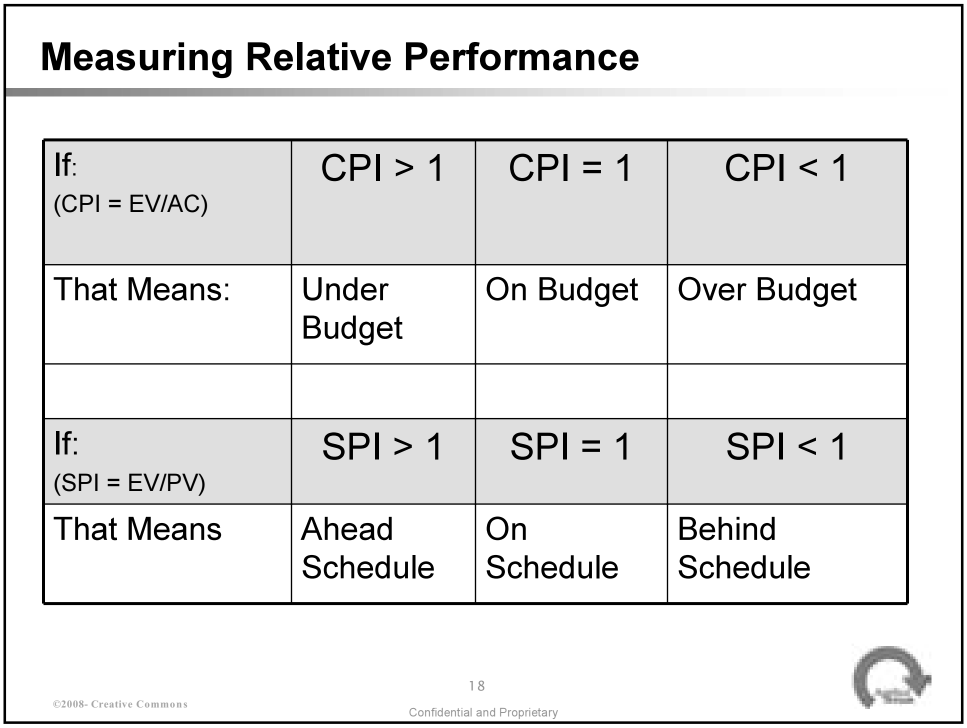
Interpreting Basic Metrics – EV and PV

So if the Earned Value is less than the Planned Value?

You are accomplishing the work late, or behind schedule

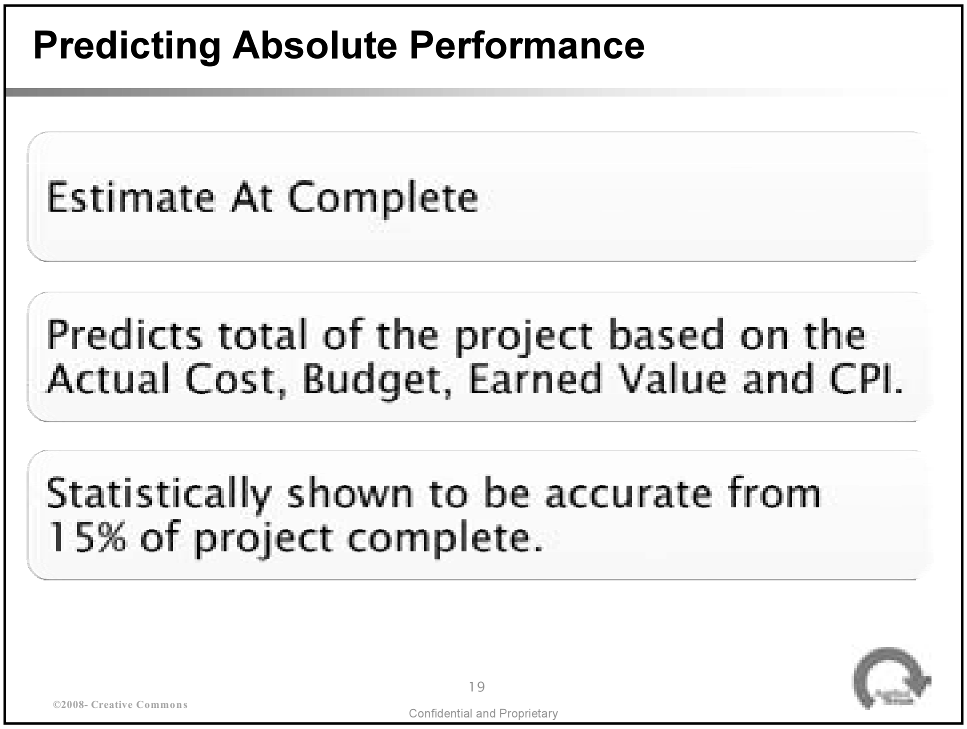
What if the Earned Value is equal or more than the Planned Value?

You are accomplishing the work on or ahead of schedule



Measuring Relative Performance

If: (CPI = EV/AC)	CPI > 1	CPI = 1	CPI < 1
That Means:	Under Budget	On Budget	Over Budget
If: (SPI = EV/PV)	SPI > 1	SPI = 1	SPI < 1
That Means	Ahead Schedule	On Schedule	Behind Schedule

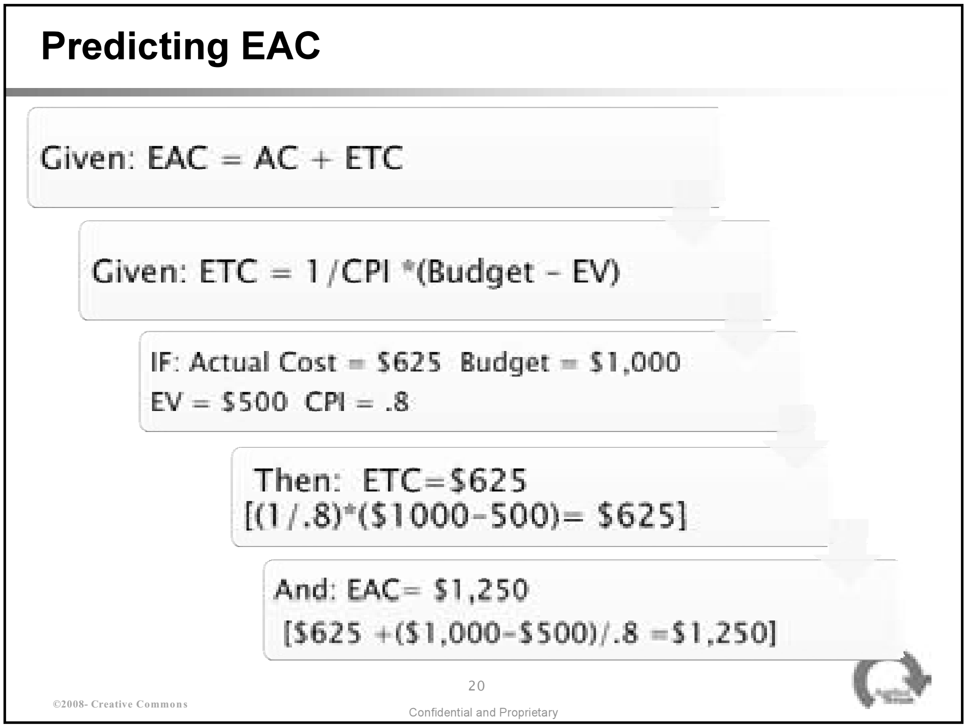


Predicting Absolute Performance

Estimate At Complete

Predicts total of the project based on the Actual Cost, Budget, Earned Value and CPI.

Statistically shown to be accurate from 15% of project complete.



Predicting EAC

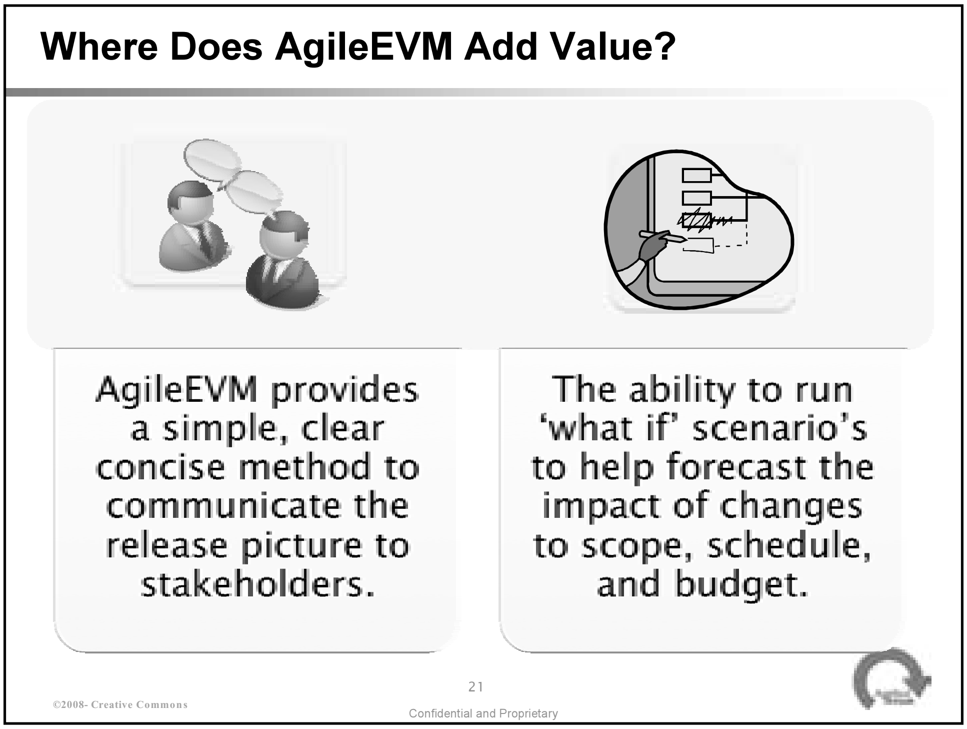
Given: EAC = AC + ETC

Given: ETC = 1/CPI *(Budget – EV)

IF: Actual Cost = $625 Budget = $1,000
EV = $500 CPI = .8

Then: ETC=$625
[(1/.8)*($1000-500)= $625]

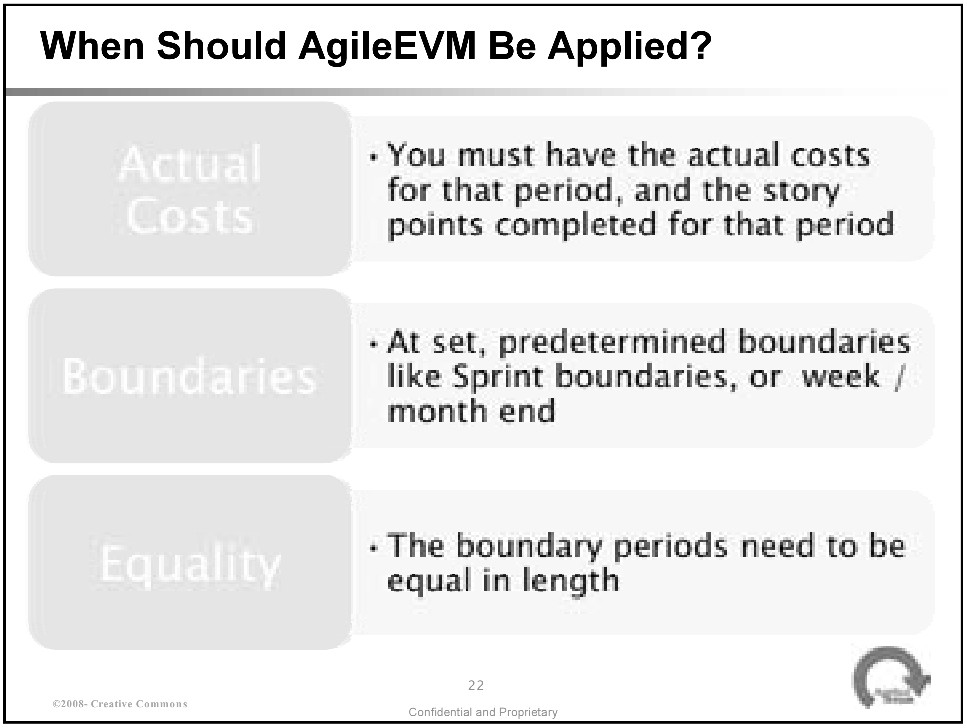
And: EAC= $1,250
[$625 +($1,000-$500)/.8 =$1,250]



Where Does AgileEVM Add Value?

AgileEVM provides a simple, clear concise method to communicate the release picture to stakeholders.

The ability to run 'what if' scenario's to help forecast the impact of changes to scope, schedule, and budget.



When Should AgileEVM Be Applied?

Actual Costs

- You must have the actual costs for that period, and the story points completed for that period

Boundaries

- At set, predetermined boundaries like Sprint boundaries, or week / month end

Equality

- The boundary periods need to be equal in length

COLLABORATION!

Guerilla Techniques

By Celeste Yeakley of Education Finance Partners
Cyeakley@educationfinancepartners.com

and

Jeff Fiebrich of Freescale Semiconductor
j.fiebrich@freescale.com

Biographies

Celeste Yeakley is an organizational leader with more than twenty years of experience in software/systems engineering and process improvement. She is a broad-based team lead and participant, with proven skills that focus teams by applying creative and innovative business planning processes. She's accomplished in business strategy, business planning, team/project management (certificate in Software Project Management), software capability enhancements, and organizational leadership. As an internal assessor, she either participated or led software process evaluations for organizations in the United States, Brazil and Russia. A University of Texas at Austin graduate with a Master's degree in Science & Technology Commercialization, Celeste has collaborated at small start up companies as well as corporate giants such as Dell and Motorola. She has contributed to her profession by serving on the UT Software Quality Institute board from 1993-2003 as well as in community quality assessments. Her passion is for helping people understand how to work together using established frameworks like CMMI and ISO to their best advantage. She encourages lateral thinking and uses every day examples to drive people to understand the big picture and how it fits into their software worlds.

Jeff Fiebrich is a Software Quality Manager for Freescale Semiconductor. He is a member of the American Society for Quality (ASQ) and has received ASQ certification in Quality Auditing and Software Quality Engineering. He is also an RABQSA International Certified Lead Auditor and has achieved CMMI-Dev Intermediate certification. He has previously worked as a Quality Manager for Tracor Aerospace in the Countermeasures Division. A graduate of Texas State University with a degree in Computer Science and Mathematics, he served on the University of Texas, Software Quality Institute subcommittee in 2003 - 2004. He has addressed national and international audiences on topics from software development to process modeling. Jeff has over twenty years of Quality experience as an engineer, project manager, software process improvement leader, and consultant. As both an internal and external consultant, he has played significant roles in helping organizations achieve ISO certification and Software Engineering Institute (SEI) Maturity Levels. He has led numerous process improvement initiatives in many areas of software engineering including project management, employee empowerment, software product engineering, quantitative management, training program management, and group problem solving. Jeff has worked extensively on efforts in the United States, Israel, Europe, India, and Asia.

Celeste and Jeff are the authors of the recently released book, 'Collaborative Process Improvement', Wiley-IEEE Press, 2007.

INTRODUCTION

Today's economic climate makes it more important to work smarter. Dedicated Quality personnel are stretched to the limit, and time constraints force greater emphasis on effective process improvements. Yet Quality and building Quality departments seem to be the lowest priority in many companies. With overwhelming amounts of work to do and no resources—at least any incremental resources—what is the Quality worker to do? The battle cry to "do more with less" has become "work hard, work smart."

This paper discusses the framework we used for developing Software Quality Engineering (SQE) Advocates on each of its projects using existing personnel. This paper also addresses the use of process Champions to perform activities at the organizational level. We found that by using Advocates and Champions, a Quality department can extend its reach. In fact, engaging non-Quality personnel in process watching can be rewarding not only to the quality of the end product but also to the overall development process.

In order for a Quality manager to be effective, he (or she) must be prepared to:

- Train non-Quality personnel
- Devise pertinent checklists
- Communicate goals effectively
- Manage by influence and
- Analyze and promote process improvements using metrics, evaluations and rewards

A Quality manager must engage the entire organization in process improvements. This can be done with no incremental increases in resources. The result is improved overall quality in both products and processes.

TRAINING NON-QUALITY PERSONNEL

Training non-Quality personnel is easier than it sounds. First you need to realize that you have to instruct members of the organization in the defined process anyway. By recruiting select personnel—those doing the development and supporting services work—you can build a team that understands the mechanisms of change and the need to engage the organization. We call these individuals Champions. Using Champions is discussed in greater detail later in this paper, but basically Champions are the main points of contact in key process areas such as Configuration Management and Requirements Management.

The other group of people we identified as being necessary to designate a specific role were what we called the Advocates. An Advocate is the day-to-day person who lives with projects and forms the very fabric of project teams. This person's duties directly relate to tracking and reporting on project progress from a process quality perspective. Both the Champions and the Advocates must be trained.

Our help was requested in improving its quality and process improvement activities. We leveraged the required training by engaging the organization early in process definition and by using pilots. This allowed staff to begin to understand the processes they needed in order to be able to perform process assessments in the future.

We recruited individuals from all levels. We looked for persons with seniority and superior technical talent and human relations skills. Those who were on probation, were mediocre performers, or were not respected within the business were not recruited or accepted. Testers, developers, and program managers all entered the arena of Quality at the same level. Although the individuals remained in their current reporting structure in the business, they also reported to the Software Quality Engineer, as shown by the dotted lines in Figure 1.

This dotted-line reporting was acknowledged and accepted by both the business and the Quality department. Our SQE position was staffed with an experienced individual with an engineering degree and 10 years of SQE experience.

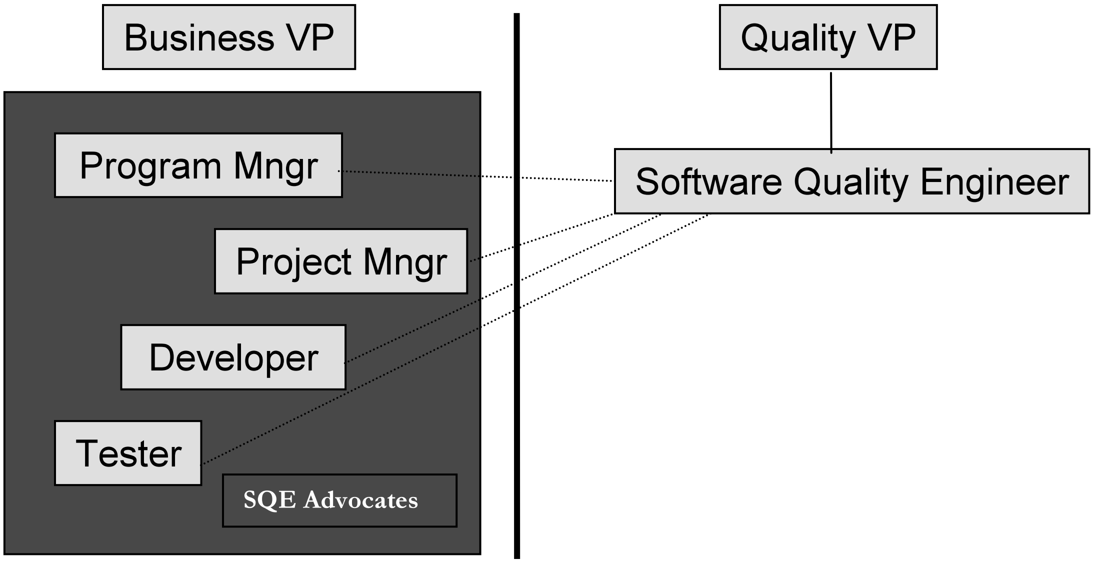


Figure 1. SQE Advocate Reporting Structure

Once the Advocates and Champions were recruited, they had to be trained to perform their new tasks. The 12 to 20 hours of training illustrated in Figure 2 was accomplished over a two month period and largely occurred during luncheons. Although few of them had ever participated in Quality activities, they soon would be performing these tasks.

Not knowing what future industry standards we might be asked to follow, we chose to use the Carnegie Mellon's Software Engineering Institute (SEI) models. Other development methodologies supported by SEI and ISO were considered when we developed our processes and were mentioned to the group, but we chose to stay focused on the SEI guidelines.

For SQE Advocate training, our first proposal was to subcontract all training to an outside source. We investigated hiring a subcontractor from two local training houses to assist with SEI training classes. The classes would be conducted in person domestically and via net-meeting at all the off-shore sites. This fee would literally double if the classes were to be specific to our group. To train all our Advocates, we required six classes at two domestic locations and at three remote locations: Romania, India, and China.

Fortunately, we discovered that we had the talent in-house to establish our own curriculum and delivered the six classes noted in Figure 2 at each site. The training curriculum for each class was coordinated with and received approval from our company's university.

Class Title	**Audience**	**Class Length**
SQE Advocate Roles & Responsibility	SQE Advocates	2 hours
Performing Assessments	SQE Advocates	2 hours
Software Configuration Management Plan	SQE Advocates	2 hours
Software Quality Assurance Plan	SQE Advocates	2 hours
SEI Overview	All	4 hours
SEI Key Process Areas	SQE Advocates & Champions	8 hours

Figure 2. Training Classes

All class materials and attendance was routed back to Human Resources for historical purposes. This resulted in the curriculum accepted by Human Resources as accredited training for individuals.

In order to ensure that this training effort would be utilized during the next several years, we also investigated hiring a local firm to record the training for future use.

Instead, we developed customized PowerPoint curricula for each site, and the local Champions provided voice-overs. These classes were recorded on CD, and featured our Champions' names and voices. We also customized the training to include supporting tools and processes. The result was a training program digitally captured for future use.

Once our group of Advocates had been established and technical training had been accomplished, we discovered that their skill set still lacked traits necessary to effectively perform the SQE tasks required:

- Quality background
- Public relations, professionalism, communication skills

- Organizational skills
- Assessment background
- Vision of overall process

QUALITY BACKGROUND

One of the first things that we noticed when working with non-Quality personnel was that they feared they would be seen as the bad guys of the project if they took on a Quality role. Most non-Quality personnel with this background are very hesitant to take on a role of this sort. In fact, in our own organization, the test group had an overly heavy burden of being the initial SQE Advocates.

We mitigated the police problems this way.

- We hosted weekly training and process forums. In order to recruit future SQE Advocates, we had to train all personnel on the basics of the SEI models.
- We educated management. Convincing management that Advocates are management material was one of the first steps we undertook in order to raise awareness of the Advocates' critical role. After a bit of training, management bought in to the idea and realized that Advocate personnel understood the overall process better than most.

PUBLIC RELATIONS, PROFESSIONALISM, COMMUNICATION SKILLS

Of course, it is easy to train on the specifics of a particular methodology and guideline—compared to trying to train behaviors! SQE Advocates' personalities and communication skills are certainly as diverse as those found in the engineering populations with whom they work. Not every person is adept at tactfulness, professionalism, or managing the project team. These are the very skills essential to communicating process issues and getting results.

You can mitigate these gaps with coaching and training. By role playing discussions with Advocates, you can teach them how to communicate both good and bad news. It is important to be able to deliver constructive criticism concerning process adherence. Several good books offer excellent information on communication and accountability. We are currently using *The Oz Principle* by C. Hickman for training.

ORGANIZATIONAL SKILLS

Lack of organizational skills can create more chaos than the process chaos you were trying to avoid in the first place! Signs of organizational issues include not being able to find process assessment information easily, not informing the team of results in a timely manner, and conducting assessments only sporadically.

Particularly valuable organizational helpers are checklists and adding the SQE Advocates activities into your project management schedules.

- By providing appropriate checklists (refer to the Devising Pertinent Checklists section), you make the work easy and help in the organization of information. You don't have to rely on individuals setting up their own reporting scheme.
- By building SQE Advocate activities directly into your project management scheduling tool, both you and the Advocate can track when to do an assessment and what processes need to be assessed.

THE AUDITING MINDSET

Your chosen SQE Advocates might demonstrate more auditing personalities than you really want! Evidence of a problem might be demonstrated by an Advocate writing up projects on non-issues or being dogmatic about your organization's chosen processes. Alternatively, an Advocate can try to be the good cop by forgiving blatant non-compliance. Other symptoms of the lack of an appropriate assessment mindset are that your Advocate is simply lost about what to do. This can be followed by a general reluctance to accept a process role.

We addressed these problems in the following ways.

- We provided extensive checklists and activity description.
- We held monthly SQE Advocate meetings. In this meeting all Advocates joined Quality personnel in a general discussion. This forum provided a free-flowing question and answer session as well as training opportunities.

VISION OF OVERALL PROCESS

By having a cross-functional group of personnel populate your Advocate Program, the vision of the overall process can easily be realized. When personnel from Sales, Marketing, Test, Development, and Program Management sit down at a table and discuss how to perform SQE activities, it quickly becomes evident what the overall objective is. The negative repercussions of omitting certain activities are also quickly and effectively communicated.

DEVISE PERTINENT CHECKLISTS

As soon as you are certain you have the most effective process, someone will find a way to improve it. This continuous improvement is a good thing—although change is usually uncomfortable. Every aspect of ensuring Software Quality is an evolutionary process. Methods to help your group cope with the evolution include the following:

- Using diverse user groups
- Piloting and refining checklists
- Executing continuous process improvements
- Migrating activities into an automated management tool

USING DIVERSE USER GROUPS

The key to the SQE Advocate Program is diversification. To have marketing, engineering, and test conduct a round robin concerning configuration management, is an eye-opening experience. When developers, testers, and managers brainstorm a new process for requirement management, you can feel the electricity in the air. Before the meeting is over, each individual is able to think about the issue in a completely new way. Cross-pollination of ideas is an invigorating experience. We noticed that individuals wanted to continue discussions long after the meeting was over and always arrived fully prepared for the next meeting.

PILOTING AND REFINING CHECKLISTS

Of course, the only way to really validate a new checklist is by applying it to a pilot project. Typically, the pilot project is an effort that is well staffed and on schedule. Picking a project such as this ensures that the checklists are well received and executed.

However, the biggest return on effort is achieved if the new checklists are applied to a problematic project. Use the buy-in of a problematic project to quickly, easily, and constantly display improvements. Project staff will be revitalized when witnessing the improvements on their own efforts.

Transforming a problematic project into a model project is what everyone wants to do. The bottom line is that no one enjoys being part of a losing team.

Checklists are the key to smooth, consistent execution. Having a standardized checklist removed most of the human personality from the assessment equation. While our checklists were initially developed by the Quality group, they were quickly updated by the Advocates. During the first six months of the Advocate Program, each checklist was updated three times on average. Continuous Improvement! Each time a checklist was updated; the update was reviewed and approved by the entire SQE Advocate team.

EXECUTING CONTINUOUS PROCESS IMPROVEMENTS

When questions are raised like "*When will this process be done? How much more work on this process is needed?*" your role, as an Advocate or Champion is to say, *"We will be constantly improving all of our processes."* At a time when our projects are heavily driven by milestones and schedules, it is difficult to keep individuals motivated when you never plan for the effort to be finalized.

When establishing and maintaining processes, we have found that the effort is never finalized. Just as we strive to improve our products every day, we must strive to improve the processes that create them every day. We aggressively recruited individuals to participate in the Six Sigma effort. The principles used in this Define Measure Analyze Improve Control (DMAIC) method not only aided SQE Advocates in active, measurable process improvement activities, they also supported SEI measurement activities.

MIGRATING ACTIVITIES INTO AN AUTOMATED TOOL

Coordinating and orchestrating SQE Advocate activities is a full-time job. While spreadsheets can be used to schedule tasks, document compliance, and initiate reports, this is probably the most difficult tool for accomplishing this task. Using a powerful automated program management tool enables you to remove all of the personality from the equation. Assessments and reviews are conducted systematically. SQE activities are standardized for all projects. Progress reports are generated on a daily, weekly, or monthly basis without the project team feeling like a renegade Advocate has figured how to take control away from the team.

COMMUNICATE QUALITY GOALS EFFECTIVELY

In order to communicate quality goals effectively, you must get the correct information disseminated as quickly as possible—and to the right people. The information often moves slowly or not at all between the improvement groups and the guys in the trenches. One effort may not know another effort even exists. Some groups see things changing, but others say, *"These improvement efforts haven't reached ME."* This problem was the driving force behind establishing a Software Quality Assurance Plan. Both the SEI and ISO models require supporting the establishment and maintenance of this plan. This document contains the schedule and provides a means of activity coordination. This plan discusses the authority of SQE and the SQE Advocates. To ensure that this is understood by all members of the team, the Program Manager, Project Manager, and Functional Managers are required to review and approve this plan.

Other methods to remedy communication shortfalls—

- Weekly forums
- Project team meetings
- Reporting

WEEKLY FORUMS

Meeting with all members of the Advocate Program on a regular basis is essential. Initially, this should probably be on a weekly basis. It takes several meetings to brief/train the group on the roles and responsibilities; documenting process areas; and performing assessments, surveys, and reviews. A set day and hour each week helps solidify the stability of the Program. As individuals become more involved with the Advocate Program and its effect on their program, the level of questions and requests for future forum topics will surprise you.

We added a little extra incentive each week by providing homemade hot lunches. This was surprisingly well received. No boxed lunches for us! Many team members confessed that they would not have come if it were not for the lunches. The investment in making these weekly forums enjoyable and fun paid off in process improvement results after only one year.

PROJECT TEAM MEETINGS

One of the Advocate's largest responsibilities is to communicate their activities at their project's team meetings. This meeting is the conduit for flowing information from the Advocate to the project. Advocates typically discussed upcoming assessments and reviews and also highlighted any accomplishments. Team meetings are also the most effective forum for briefing the project about its non-compliance with their processes goals.

REPORTING

The Advocate should communicate good and bad news to the project at weekly project meetings. If there is an issue that cannot be resolved with project personnel, the Advocate may discuss the item with the Quality Manager. It is important that the SQE Advocate be empowered by Executive Management to stop a project that is in non-compliance. For this reason, it is better to share Advocates. In other words, an Advocate would ideally be assigned to report on a project to which he or she does not directly report. In the case of two or more projects, the projects would trade an Advocate's time on an independent project. In this way the Advocate can be spared the fear of a backlash from his or her own manager for reporting process non-compliance.

MANAGING BY INFLUENCE

The Quality Manager must put on his or her best consultant hat and step out of the limelight to encourage a non-Quality person to step up and be an example for the group. As Malcolm Gladwell points out in the book *Tipping Point,* you need to recruit a thought leader—someone who is known and respected among the non-Quality community to help speak those Quality words for you and manage by influence. If you cannot locate someone within the project group who is willing to dedicate time to this effort, a new independent-eye may be recruited from outside the project group. But, without question, it is essential to have someone well known and highly respected play this important influential role and lead this activity

Managing by influence involves the following.

- Using Champions

- Engaging upper management
- Ensuring annual goal commitments
- Obtaining executive team participation

USING CHAMPIONS

Having a Champion for SQE activities is an absolute for communication. This person is responsible for recruiting more members to the Advocate Program and for their training. Curriculum should be documented, briefed, evaluated, and improved on an annual basis. Members should be recruited year round. Having a Champion as the point of contact for all activities is even more helpful. The Champion is often the liaison between the Advocates and management. The Champion is also the key source for problem resolution.

The Champion often helps to define new and improved processes that are quickly adopted. Champions for each key process area are also responsible for process documentation, improvement, and training.

During the first 12 months of the Advocate Program, we saw the enrollment increase from 3 to 30, as shown in Figure 3. That is a 1000 percent increase in participation. This growth was directly attributed to the communicated importance from upper management and the positive project impacts of the program that were the subject of many Operational Review meetings.

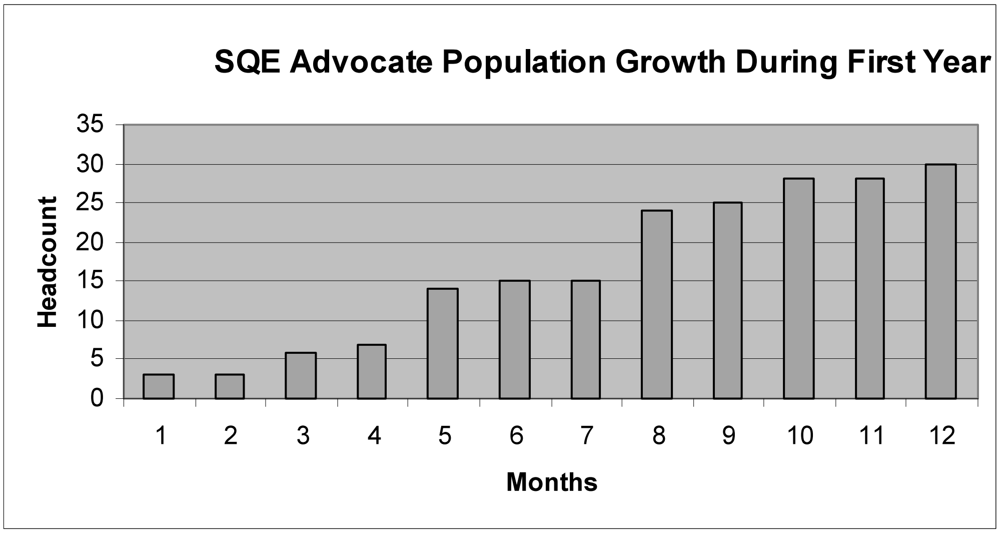


Figure 3. SQE Advocate Population Growth

ENGAGING UPPER MANAGEMENT

The Advocates and Champions must engage upper management. We strongly recommend that Advocates and Champions initiate this action. Upper management has many responsibilities; Advocate and Champion interfaces are one of many that they must address. When communicating improvement plans to members of upper management, management will likely voice their support and then assume that during the next months, the plan is successfully executed. It is up to the Advocates and Champions to go back to upper management regularly and report the status. Upper management should be briefed on successes as well as setbacks. If not approached again, management may believe that all is well and successful.

One key tip for the Quality Manager: never assume that upper management Gets It just because you've explained it to them. Advocates and Champions should plan on monthly status updates and formal reports.

ENSURING ANNUAL GOAL COMMITMENTS

By using a vehicle such as a Balanced Scorecard to communicate annual goals, you can ensure that testers, engineers, and managers are aware and supportive of the Advocate effort. It is wise to make the goals of the Advocate program as visible as the engineering goals. The rewards associated with engineering goals should map to the rewards associated with Advocate goals. It should be communicated that the achievement of Advocate goals are essential to the success of the company.

OBTAINING EXECUTIVE TEAM PARTICIPATION

Having executive team members participate in activities is essential—even if executive team members only attend team briefings. Having the executive team meet with the Advocates and repeatedly reassure the Advocate that they are empowered to stop the project if the project is found to be non-compliant is important to the success of this effort. It is also important for Advocates and Champions to attend the Upper Level Management Briefings on at least a monthly basis.

Making Advocates and Champions visible in the executive environment is as important as having executives visible in the Advocate environment. Better yet, if you have the ability to fast track an Advocate or Champion to more responsibility, their peers will take notice, and this will cause individuals to actively pursue one of those positions as it will be seen as highly regarded by management and a way to career success.

ESTABLISING MEANINGFUL METRICS, EVALUATIONS, AND REWARDS

Because our effort revolved around the SEI guidelines, we measured our progress according to that scale. We set up SEI assessment polls on our internal web site and posted green/yellow/red status charts, as shown in Figure 4. The entire population could view these results. Before we knew it, projects were vying for the top spot. We measured compliance by percentage and tallied the results. Many organizations share this information only with the team. We found that the culture thrived on competition. First, they were excited to get all green. Then they competed to have the highest percentage for compliance. It added a lot of fun to the metrics experience to announce weekly status.

	REQ	**PP**	**PMC**	**SUB**	**QA**	**CM**	
Project A	Y	G	G	G	Y	Y	**83%**
Project B	Y	G	G	R	R	Y	**70%**
Project C	Y	G	G	G	R	Y	**76%**
Project D	Y	Y	Y	Y	R	R	**68%**

Figure 4. Stoplight Chart (Red, Green, Yellow)

Evaluations were key to getting every person’s attention. Because of our need to keep management informed and because we had specific and measurable goals to obtain, it was relatively easy to get all supervisors to add process improvement activities to yearly goals. People who were active in the process improvements efforts were rewarded for their alignment with and support of the group’s goals. In addition, it was obvious that teams that engaged in Quality activities performed better. Comparisons of project team performance revealed that teams that had an active SQE Advocate performed better by delivering better products closer to the time they said they’d deliver them.

Each SQE Advocate was engaged in giving constant feedback to the team by reporting on compliance to specific activities. Teams could benchmark themselves against others as well as compare their own performance to that in the months prior to the improvements.

We used intrinsic and extrinsic rewards and recognition throughout the year. Intrinsic rewards and recognition included public recognition, trophies, plaques, certificates, special parking spaces, pictures on bulletin boards, and names on a list. Extrinsic rewards and recognition included cash, vacation time, bonuses, and major gifts. When initially planning the types of rewards for the Advocates and Champions, 80 percent of the Advocates said they would prefer extrinsic rewards: *“Money is what pays my bills!”* Much to our surprise, as illustrated in Figure 5, twelve months after program initiation, simply receiving a certificate during a staff meeting was everyone’s desire. As a matter of fact, at least once a week, we had individuals approach us and state *“I do not care about a cash bonus; “I want to know what I have to do to have the Director discuss my accomplishments for 10 minutes at the next Operations meeting.”* And of course, this reward cost the business nothing.

Team rewards, which are intrinsic or extrinsic items given to the team, should be the same for all members of the team.

The ultimate reason that rewards and recognition are given is to provide positive reinforcement for correct behavior, with the expectation that the correct behavior will be repeated in the future. Rewards and recognition are best received when they are personal to the individual receiving them.

Management recognition of an individual who has successfully completed an objective can be very positive and can encourage other individuals to strive for excellence.

Reward Preference	Program Initiation	One year later
Extrinsic Reward Cash, stock, gift certificates, movie tickets	80%	5%
Intrinsic Reward Certificates, pins, plaques, t-shirts	20%	95%

Figure 5. Extrinsic vs. Intrinsic Rewards

Probably one of the best rewards is *"Thank you"* when it is sincerely meant. Employees who are aware that their efforts are appreciated are often willing to do more than if they were to receive a large financial reward.

DIVISION OF WORKLOAD

At the initiation of the Advocate program, the first question asked by all parties involved was *"How much of my time will this require?"* Initially, we had no idea, but we stated a target of 10 to 20 percent. A year into the program, we totaled the hours being spent by the Advocates. At first glance, it appeared that 40 percent of the individual's time was being spent away from his normal tasks. This was alarming. After a deeper dive into the data, we determined that the Advocates were spending 60 percent of their time doing engineering and 20 percent of their time performing code and documentation review for which they were responsible anyway. Only 20 percent of their time was being spent specifically on Advocate activities—Configuration Management, Quality Assurance, and process assessments—as shown in Figure 6.

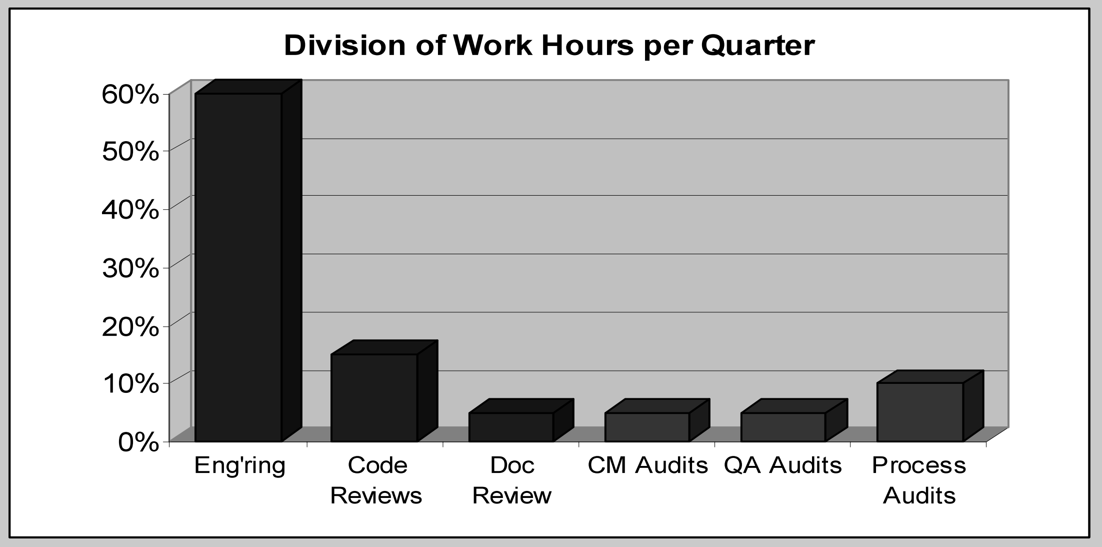


Figure 6. Division of Work Hours per Quarter

CONCLUSIONS

Quality processes seem to pay off in quality products. What organizations miss sometimes is something that Ford Motor Company (and others) discovered long ago; "Quality is Job 1". It should be woven into all activities of any company but particularly in a software company because there are many ways that software can fail. It can fail during requirements, design, implementation, and test—literally *anywhere* in its lifecycle—but most of all, it can fail with the customer. It just makes sense to insert Quality processes in along the way when building software. An important part of this Quality approach should be in training the organization—the whole organization—about what quality is for your products. Relying solely on an organizational Quality group for overall quality is difficult and can fail if you do not engage the individuals creating the product in the process. Quality should be the thread by which the fabric of software is built. In order to accomplish this, you need to engage people actively in all stages.

In this way, you weave quality in rather than coating the threads after the fabric is woven. That makes quality easier to attain and sustain. If you are fortunate enough to have a Quality group, this approach will only serve to strengthen it.

We found that engaging everyone in the organization in Quality work resulted in quality results. By training the entire organization and enlisting our techniques, we were able to not only do more with less but work harder and smarter. The payoff was in the bottom-line benefit to the company.

ACKNOWLEDGEMENTS

We would like to thank the reviewers who provided useful suggestions for improving this paper and also to gratefully acknowledge Cinda Cyrus for her editorial assistance in preparing this manuscript.

REFERENCES AND RESOURCES

The following sources were either used in the preparation of this paper or will serve as resources to the practitioner.

Fiebrich, J., C. Yeakley, (2004). "The Quality KISS—Keeping Quality Simple in a Complex World," in *Proceedings*, International Conference on Practical Software Quality Techniques East, March 2004, Washington, D.C.

Fiebrich, J., C. Yeakley, (2004). "Guerilla Quality—Innovative Ways to Engage Personnel in Process Improvement." in *Proceedings*, International Conference on Software Process Improvement, June 2004, Washington, D.C.

Fiebrich, J., C. Yeakley, (2004). "Strategic Quality—Planned Empowerment." in *Proceedings*, International Conference on Practical Software Test Techniques North, October 2004, Minneapolis, Minnesota.

Fiebrich, J., C. Yeakley, (2005). "The Q-Files—What Is Your Customer's Definition of Quality?" in *Proceedings*, Software Develop & Expo West 2005, March 2005, Santa Clara, California.

Fiebrich, J., C. Yeakley, (2005). "Guerilla Quality—It's Time to Deploy!" in *Proceedings*, Software Develop & Expo West 2005, March 2005, Santa Clara, California.

Fiebrich, J., C. Yeakley, (2005). "Development Lifecycles—Plunging Over the Waterfall!" in *Proceedings*, International Conference on Practical Software Quality Techniques West, May 2005, Las Vegas, Nevada.

Fiebrich, J., C. Yeakley, (2005). "Customer Facing—It's Time for an Extreme Makeover!" in *Proceedings*, International Conference on Practical Software Quality Techniques West, May 2005, Las Vegas, Nevada.

Fiebrich, J., C. Yeakley, (2005). "Configuration Management—A Matter of Survival!" in *Proceedings*, UML & Design World 2005 – Architecture, Design, Modeling, and Beyond, June 2005, Austin, Texas.

Gladwell, Malcolm. *Tipping Point: How Little Things Can Make a Big Difference.* Little, Brown, and Company. 2000 – 2002.

Hickman, C., Connors, R., Smith, T. *The OZ Principle*. Muze Inc, 1995 – 2005.

Trust: The Key to Project Team Collaboration

Diana Larsen, FutureWorks Consulting LLC

Trust forms the bedrock of effective software teams. Trust allows teams to communicate quickly and to respond rapidly to changes in the project. Without sufficient trust, team members waste effort and energy hoarding information, forming cliques, dodging blame, and covering their tracks. A climate of trust provides a foundation for effective team processes, adaptability, and high performance. How can we shatter the deep-seated cycle of distrust in many organizations and help this essential trust emerge? Team leaders can stimulate and accelerate trustworthiness and trusting among team members, and between the team and its stakeholders by paying attention to membership, interactions, credibility, respect, and behaviors. In this session we'll investigate ways to accelerate trust-building within teams, including a definition of professional trust, a model for team interactions that leverages trust, ways to recognize when a team has "trust issues," and skills that help teams develop greater trust.

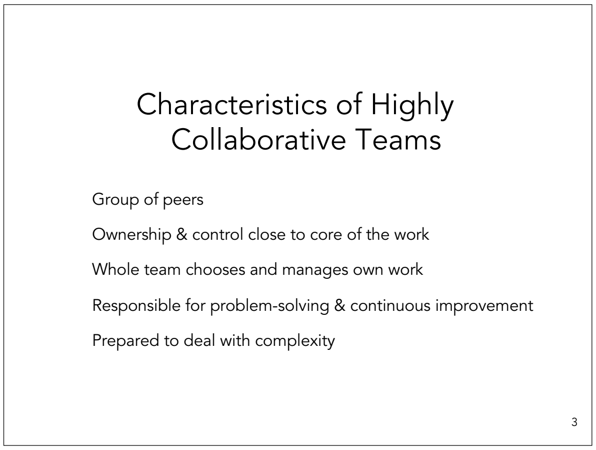
***Diana Larsen** is known in the software industry for conducting project retrospectives and transitioning groups to Agile processes. She currently chairs the board of the Agile Alliance. Her publications include Agile Retrospectives, Making Good Teams Great, coauthored with Esther Derby. She consults and speaks internationally.*

Trust

The Key to Project Team Collaboration

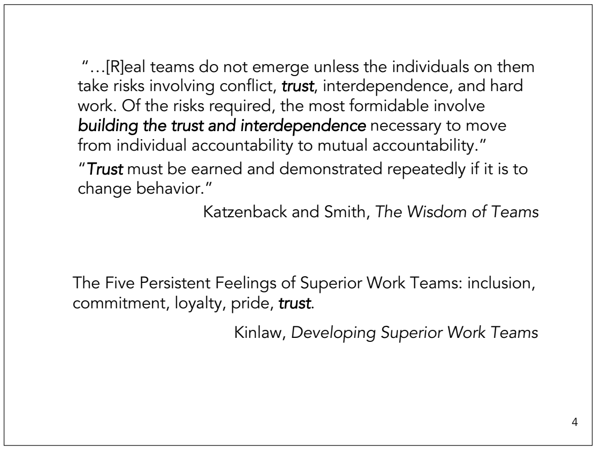
We work as a Team when we have…

- Common purpose & performance goals
- Complementary skills for interdependent work
- Shared approach to work
- Joint accountability
- Small number of people
- Mutual History



Characteristics of Highly Collaborative Teams

- Group of peers
- Ownership & control close to core of the work
- Whole team chooses and manages own work
- Responsible for problem-solving & continuous improvement
- Prepared to deal with complexity



"…[R]eal teams do not emerge unless the individuals on them take risks involving conflict, ***trust***, interdependence, and hard work. Of the risks required, the most formidable involve ***building the trust and interdependence*** necessary to move from individual accountability to mutual accountability."

"***Trust*** must be earned and demonstrated repeatedly if it is to change behavior."

Katzenback and Smith, *The Wisdom of Teams*

The Five Persistent Feelings of Superior Work Teams: inclusion, commitment, loyalty, pride, ***trust***.

Kinlaw, *Developing Superior Work Teams*

Trust is a significant factor in project success. Trust in leaders and other team members relates to higher organizational performance. The level of trust positively correlates to:

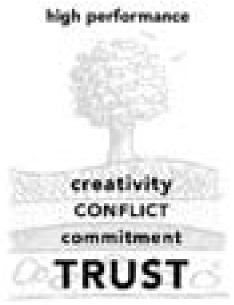
- job performance
- organizational citizenship behavior
- turnover intentions
- job satisfaction
- organizational commitment
- commitment to decisions

summarized from
Dirks & Ferrin, 2002

"The key, we believe, is trust. When members of a group trust one another's motives, their competence, and their concern for the task, the work of any becomes the work of all. Group dynamicists know that. It's one reason they try to build interpersonal trust from the very start."

Lipman-Blumen and Leavitt.
Hot Groups

5



Collaborative Team Communication Model

6

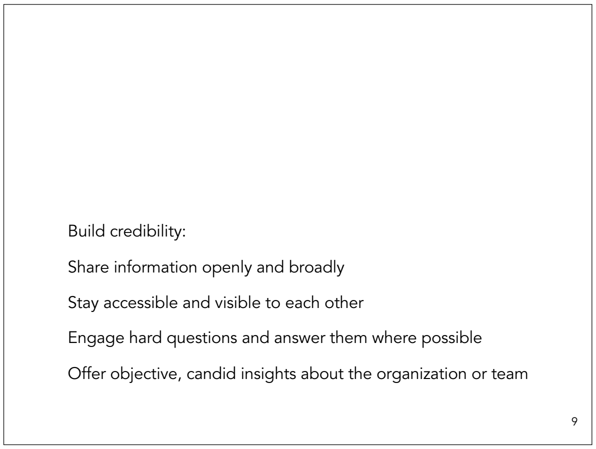
Three Aspects of Professional Trust

7

Credibility

competence, believability, integrity

8



Build credibility:

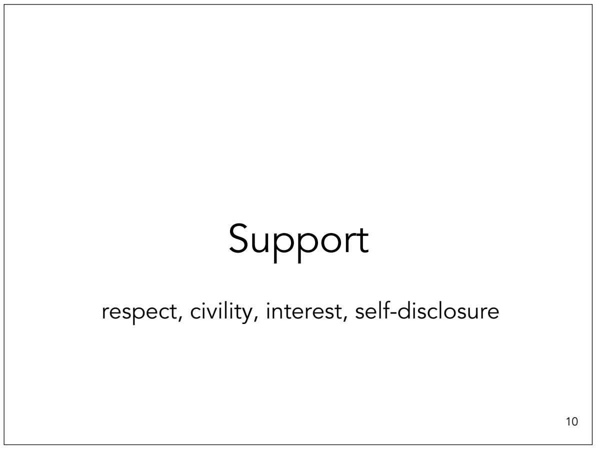
Share information openly and broadly

Stay accessible and visible to each other

Engage hard questions and answer them where possible

Offer objective, candid insights about the organization or team

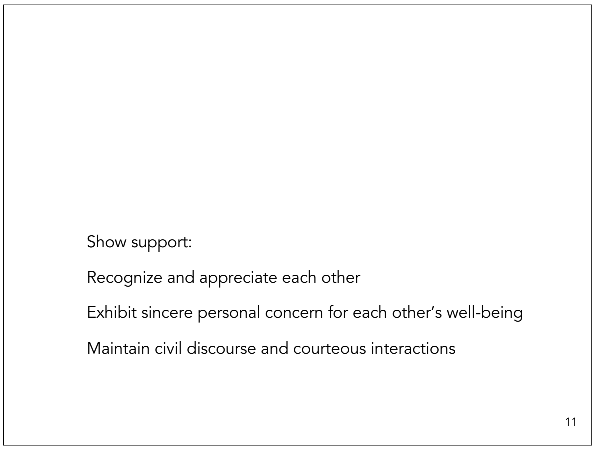
9



Support

respect, civility, interest, self-disclosure

10



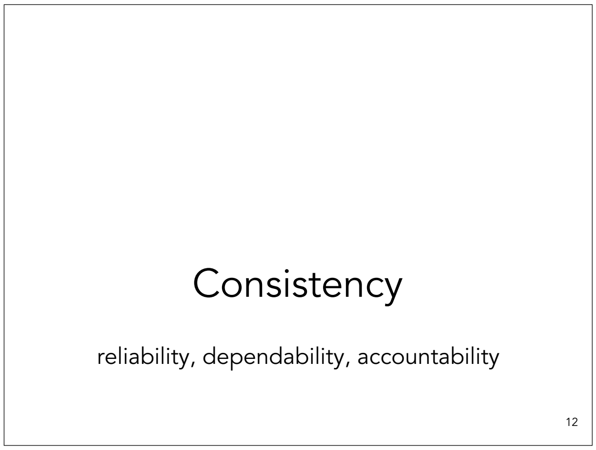
Show support:

Recognize and appreciate each other

Exhibit sincere personal concern for each other's well-being

Maintain civil discourse and courteous interactions

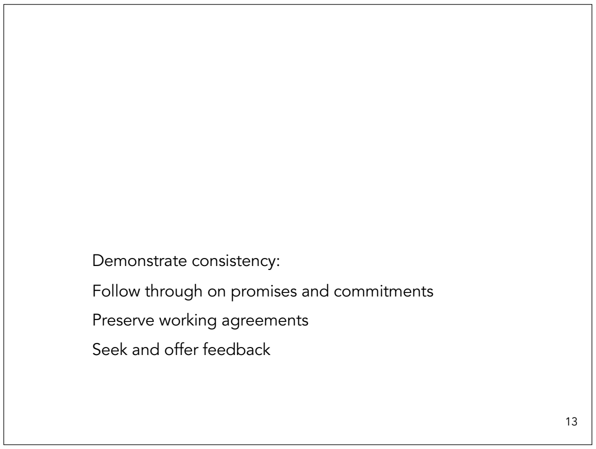
11



Consistency

reliability, dependability, accountability

12

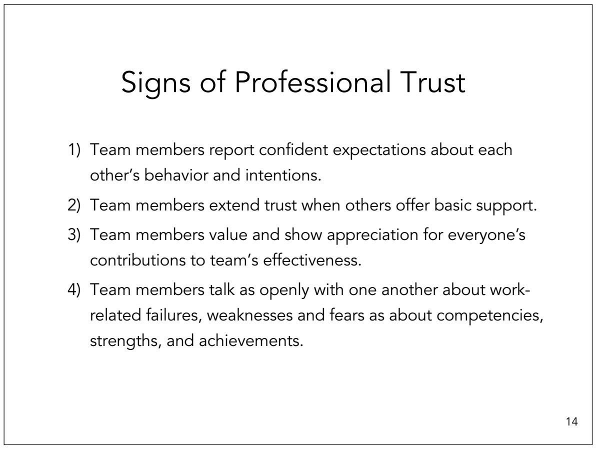


Demonstrate consistency:

Follow through on promises and commitments

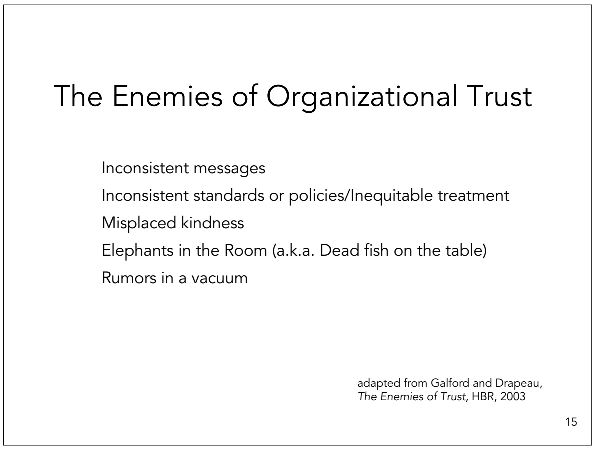
Preserve working agreements

Seek and offer feedback



Signs of Professional Trust

1) Team members report confident expectations about each other's behavior and intentions.
2) Team members extend trust when others offer basic support.
3) Team members value and show appreciation for everyone's contributions to team's effectiveness.
4) Team members talk as openly with one another about work-related failures, weaknesses and fears as about competencies, strengths, and achievements.



The Enemies of Organizational Trust

Inconsistent messages

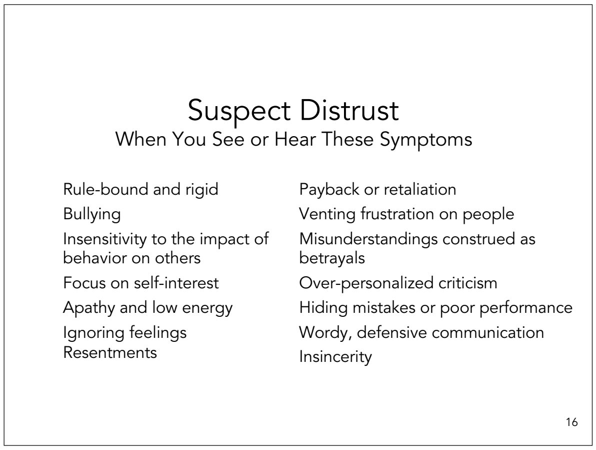
Inconsistent standards or policies/Inequitable treatment

Misplaced kindness

Elephants in the Room (a.k.a. Dead fish on the table)

Rumors in a vacuum

adapted from Galford and Drapeau,
The Enemies of Trust, HBR, 2003



Suspect Distrust
When You See or Hear These Symptoms

Rule-bound and rigid

Bullying

Insensitivity to the impact of behavior on others

Focus on self-interest

Apathy and low energy

Ignoring feelings

Resentments

Payback or retaliation

Venting frustration on people

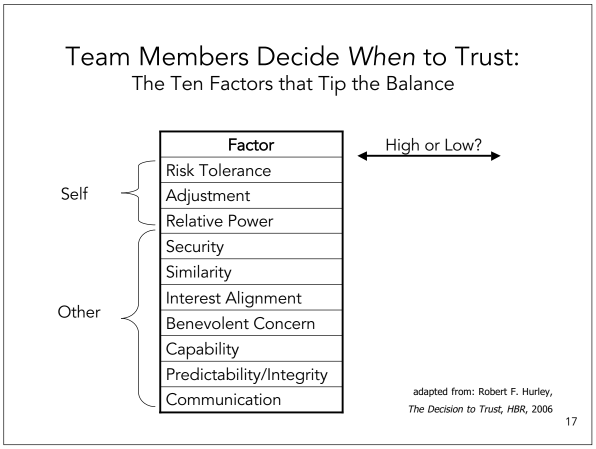
Misunderstandings construed as betrayals

Over-personalized criticism

Hiding mistakes or poor performance

Wordy, defensive communication

Insincerity



Team Members Decide *When* to Trust:

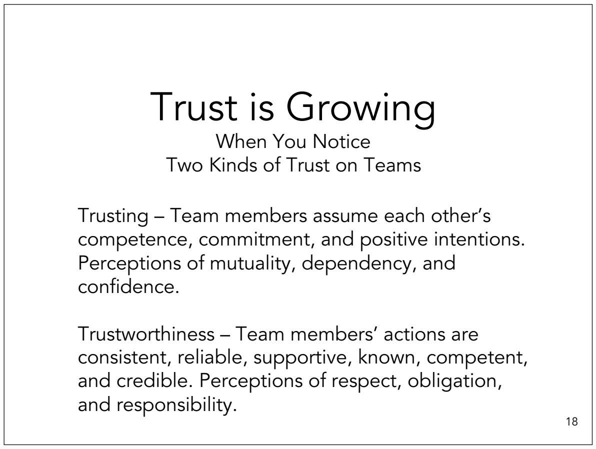
The Ten Factors that Tip the Balance

	Factor
Self	Risk Tolerance
	Adjustment
	Relative Power
Other	Security
	Similarity
	Interest Alignment
	Benevolent Concern
	Capability
	Predictability/Integrity
	Communication

High or Low?

adapted from: Robert F. Hurley, *The Decision to Trust, HBR*, 2006

17



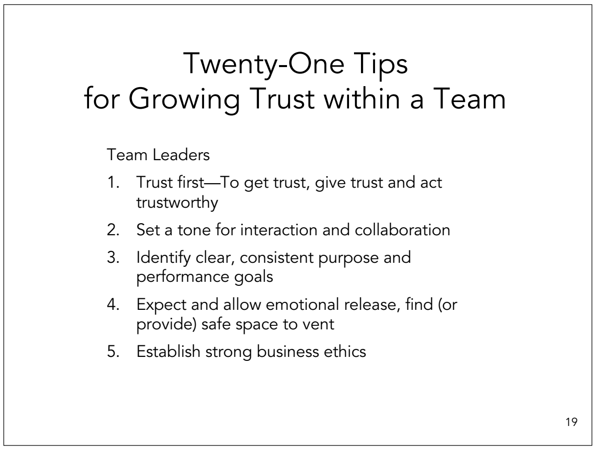
Trust is Growing

When You Notice
Two Kinds of Trust on Teams

Trusting – Team members assume each other's competence, commitment, and positive intentions. Perceptions of mutuality, dependency, and confidence.

Trustworthiness – Team members' actions are consistent, reliable, supportive, known, competent, and credible. Perceptions of respect, obligation, and responsibility.

18

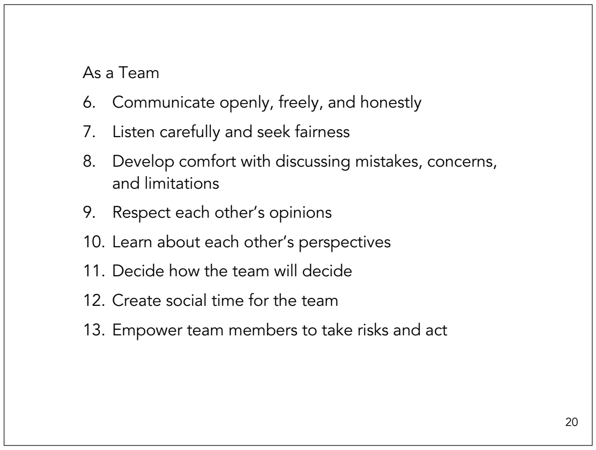


Twenty-One Tips for Growing Trust within a Team

Team Leaders

1. Trust first—To get trust, give trust and act trustworthy
2. Set a tone for interaction and collaboration
3. Identify clear, consistent purpose and performance goals
4. Expect and allow emotional release, find (or provide) safe space to vent
5. Establish strong business ethics

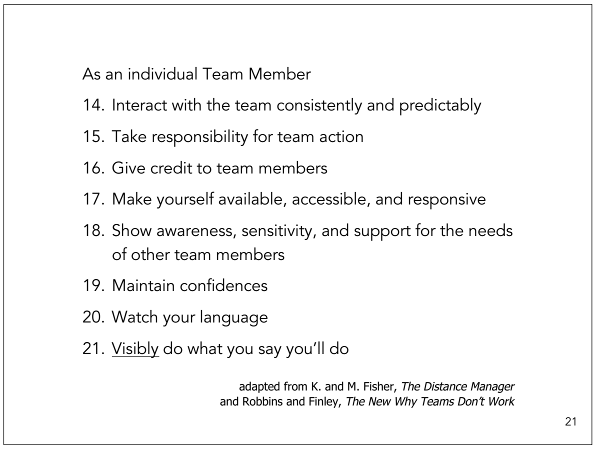
19



As a Team

6. Communicate openly, freely, and honestly
7. Listen carefully and seek fairness
8. Develop comfort with discussing mistakes, concerns, and limitations
9. Respect each other's opinions
10. Learn about each other's perspectives
11. Decide how the team will decide
12. Create social time for the team
13. Empower team members to take risks and act

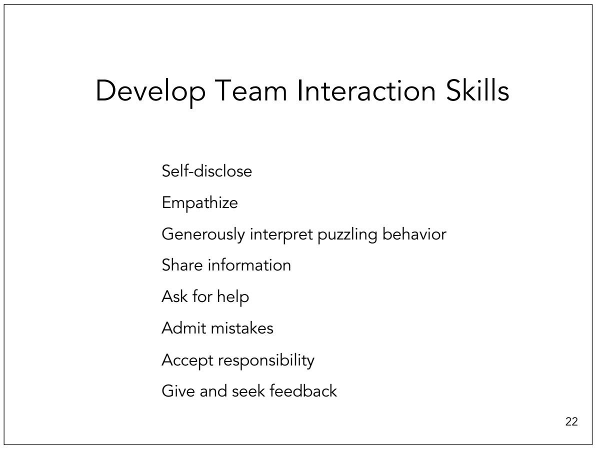
20



As an individual Team Member

14. Interact with the team consistently and predictably
15. Take responsibility for team action
16. Give credit to team members
17. Make yourself available, accessible, and responsive
18. Show awareness, sensitivity, and support for the needs of other team members
19. Maintain confidences
20. Watch your language
21. <u>Visibly</u> do what you say you'll do

adapted from K. and M. Fisher, *The Distance Manager*
and Robbins and Finley, *The New Why Teams Don't Work*



Develop Team Interaction Skills

Self-disclose

Empathize

Generously interpret puzzling behavior

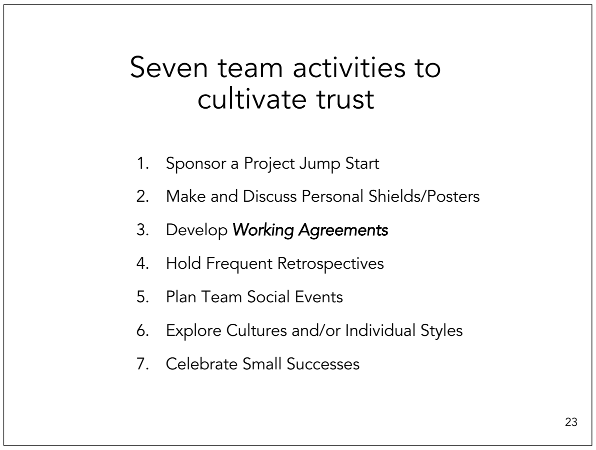
Share information

Ask for help

Admit mistakes

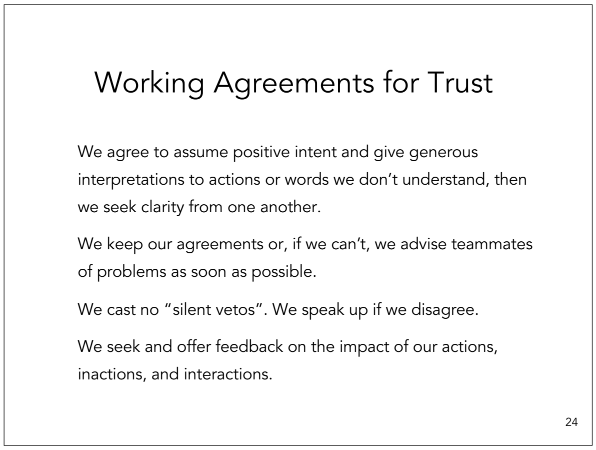
Accept responsibility

Give and seek feedback



Seven team activities to cultivate trust

1. Sponsor a Project Jump Start
2. Make and Discuss Personal Shields/Posters
3. Develop *Working Agreements*
4. Hold Frequent Retrospectives
5. Plan Team Social Events
6. Explore Cultures and/or Individual Styles
7. Celebrate Small Successes



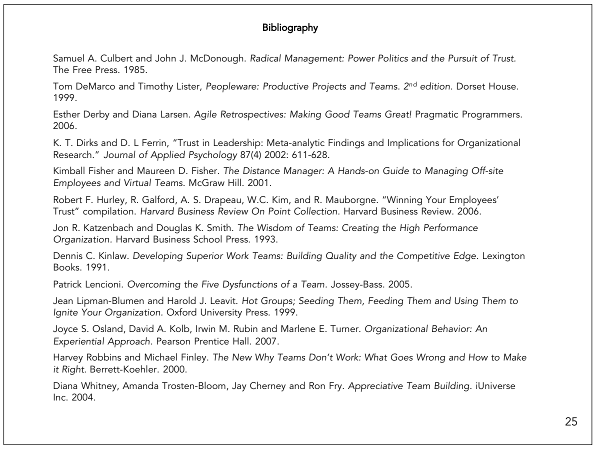
Working Agreements for Trust

We agree to assume positive intent and give generous interpretations to actions or words we don't understand, then we seek clarity from one another.

We keep our agreements or, if we can't, we advise teammates of problems as soon as possible.

We cast no "silent vetos". We speak up if we disagree.

We seek and offer feedback on the impact of our actions, inactions, and interactions.



Bibliography

Samuel A. Culbert and John J. McDonough. *Radical Management: Power Politics and the Pursuit of Trust.* The Free Press. 1985.

Tom DeMarco and Timothy Lister, *Peopleware: Productive Projects and Teams. 2nd edition*. Dorset House. 1999.

Esther Derby and Diana Larsen. *Agile Retrospectives: Making Good Teams Great!* Pragmatic Programmers. 2006.

K. T. Dirks and D. L Ferrin, "Trust in Leadership: Meta-analytic Findings and Implications for Organizational Research." *Journal of Applied Psychology* 87(4) 2002: 611-628.

Kimball Fisher and Maureen D. Fisher. *The Distance Manager: A Hands-on Guide to Managing Off-site Employees and Virtual Teams.* McGraw Hill. 2001.

Robert F. Hurley, R. Galford, A. S. Drapeau, W.C. Kim, and R. Mauborgne. "Winning Your Employees' Trust" compilation. *Harvard Business Review On Point Collection.* Harvard Business Review. 2006.

Jon R. Katzenbach and Douglas K. Smith. *The Wisdom of Teams: Creating the High Performance Organization*. Harvard Business School Press. 1993.

Dennis C. Kinlaw. *Developing Superior Work Teams: Building Quality and the Competitive Edge*. Lexington Books. 1991.

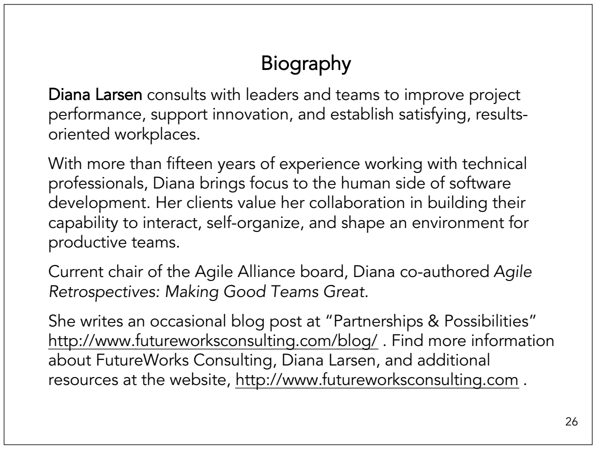
Patrick Lencioni. *Overcoming the Five Dysfunctions of a Team*. Jossey-Bass. 2005.

Jean Lipman-Blumen and Harold J. Leavit. *Hot Groups; Seeding Them, Feeding Them and Using Them to Ignite Your Organization.* Oxford University Press. 1999.

Joyce S. Osland, David A. Kolb, Irwin M. Rubin and Marlene E. Turner. *Organizational Behavior: An Experiential Approach*. Pearson Prentice Hall. 2007.

Harvey Robbins and Michael Finley. *The New Why Teams Don't Work: What Goes Wrong and How to Make it Right*. Berrett-Koehler. 2000.

Diana Whitney, Amanda Trosten-Bloom, Jay Cherney and Ron Fry. *Appreciative Team Building.* iUniverse Inc. 2004.



Biography

Diana Larsen consults with leaders and teams to improve project performance, support innovation, and establish satisfying, results-oriented workplaces.

With more than fifteen years of experience working with technical professionals, Diana brings focus to the human side of software development. Her clients value her collaboration in building their capability to interact, self-organize, and shape an environment for productive teams.

Current chair of the Agile Alliance board, Diana co-authored *Agile Retrospectives: Making Good Teams Great.*

She writes an occasional blog post at "Partnerships & Possibilities" http://www.futureworksconsulting.com/blog/ . Find more information about FutureWorks Consulting, Diana Larsen, and additional resources at the website, http://www.futureworksconsulting.com .

Playing Nice in the Sandbox

Janet Gregory

In the beginning of the agile discussions, developers and customers reigned over the world. Over the years, testers have raised their heads and said, “We want to be involved. We think we have a place in this agile world!” Well, testers have found a place in many agile teams, but not in the same way as the testers of old. Instead of being the “quality police,” they have become a conduit for providing feedback to the team. Collaboration is the key to a healthy working relationship. Teams that are moving from a traditional phased approach where developers and testers are separated and have a throw it over the wall mentality may struggle with the concept. Janet will share tips for collaboration between developers and testers, how they can learn from each other, and how together, they contribute to make a successful team.

***Janet Gregory** is a Calgary-based consultant specializing building quality systems and her passion is promoting agile quality processes. She has helped to introduce development agile practices into companies as tester or coach, and has successfully transitioned traditional test teams into the agile world. Her focus is working the business users and testers to understand their role in agile projects.*

She is currently writing a book on agile testing with Lisa Crispin due out early 2009. Janet has presented at the Agile and StarWest conference several times and is active in the Agile Testing community.

Actually, It IS All About You!

Jim Brosseau

Biography

Jim Brosseau

Jim has been in the software industry since 1980, in a variety of roles and responsibilities. He has worked in the QA role, and has acted as team lead, project manager, and director. Jim has wide experience project management, software estimation, quality assurance and testing, peer reviews, and requirements management. He has provided training and mentoring in all these areas, both publicly and onsite, with clients on three continents.

He has published numerous technical articles, and has presented at major conferences and local professional associations. His first book, Software Teamwork: Taking Ownership for Success, was published in 2007 by Addison-Wesley Professional. Jim lives with his wife and two children in Vancouver.

Actually, It IS All About You!

Jim Brosseau

August 11, 2008

Most branded approaches for software development are pitched with little active consideration for the team that will be dealing with that new approach. One could cynically suggest this is because these approaches are hailed as silver bullets that are independent of the target team, but it goes deeper than that.

If we really want to improve the way we develop software, we need to start by understanding the team itself: what each participant brings to the table. We need to bring all this out into the open, discuss our shared goals, reflect on our cultural distinctions, and leverage our combined strengths. The better we understand each other on the team, the stronger our trust will be. Our relationships will be more genuine, and our interactions will be more effective. With a consciously managed inventory of our skills, experiences and attitudes, we are better equipped to select which approach will be most effective and how to deploy this refined approach with the team.

This all takes time, to be sure. Consider, though, the cost of miscommunication, disappointments, broken trust and other forms of interpersonal dysfunction on most projects. In that light, the effort to better appreciate everyone involved on the project starts to look much more like an investment than a cost.

Relationships are hidden behind tools and techniques

There are several reasons for neglecting details about the human element, but deeper challenges in failing to do so.

First, the complexity of dealing with the human element when we are dealing with teams and processes can be daunting. Teams come in all shapes and sizes. There are at least as many different motivating factors for participating on projects as there are people. There are many complex emotions that come into play, and this entire mix is constantly changing. It is impossible to codify this complexity and provide a generalized solution as we often do with practices and methodologies. It would need to be simple enough to be easily communicated without losing the audience's attention, but there are too many variables here.

Secondly, most of these branded approaches got their start with a handful of people who had success on a handful of projects. While there may be some discussion of elements such as trust and fearlessness, these are generally dealt with through hand waving and generalities, such as "get along with your teammates" or "have fun". For the most part, concrete emphasis is on the practices themselves. In almost any retrospective I have been a part of, discussions about what worked and what needs to change focus almost exclusively on 'best practices' such as defining scope and planning, design practices, implementation and validation approaches. Even if symptoms of dysfunction such as the team not getting along are raised, the solution space usually consists of practices we see as within our control – those same practices described above.

Implicit in most successful teams is the relationships that have developed among the team members. Indeed, when you are asked to think back and describe the ***approach*** used on a project that you would have considered successful, there is a good chance that you would describe elements like a change management process, or nightly builds, or test-first development, or tactical access to a customer. If you look deeper, though, and consider ***how*** the team worked together, you will find other, more important elements. The team surely cooperated well with one another, demonstrated a level of trust and shared a common goal. Everyone felt like they belonged, and actively participated in working toward that project's success.

In working with teams, I explicitly asked them to identify the characteristics of their best project memories, carefully avoiding biases towards approaches or relationships. The one common element that everyone has identified as part of their successful project experience was that they had fun.

There are rarely any characteristics raised about tools or technology, and nothing about lifecycles or methodologies. These are necessary, but insufficient. Success is primarily dependent on the relationship between the team members. This does not simply go without saying. It is absolutely critical, and needs to be consciously managed. As a team member, it IS all about you.

There will be relationships between team members whether or not these are consciously observed or managed. Regardless of our selected approach for developing software, people will have to interact with one another. Each of these interactions will be successful or not based on the trust, respect and shared vision of the participants. If all of this is left to chance, how much of a consistent impact can one selected approach have over another?

We need to understand *OUR* roles

It is all too predictable. Whenever my son and daughter are playing together, it is only a matter of time before I hear raised voices. One of them, if not both, will come to me complaining about something the other one did. I've settled into a response that usually works quite well: to ask them to consider their contribution to the situation. What is most fascinating about this is how often a similar dance occurs on project teams.

Whenever a relationship goes sour, even a little bit, there are two sides of the story. It is human nature to pay most attention to the details as they are seen from our perspective. This often exists to the point where we are blind to the idea that there even can be another perspective at all. From our perspective, we usually see that we have been wronged, that our actions were innocuous. Unfortunately, that is rarely the case. Here are a couple of interesting examples, based on working with clients using our diagnostic product over the years.

The first case comes from a group that performs better than most of the teams I have worked with. While not always performing to that high level, they have dramatically improved over the years. They are at the point where they are a strong example of the effectiveness of conscious focus on practical implementation of reasonable practices and continuous improvement. They are also one of the few companies that has been successful at leveraging outsourced resources, by carefully managing subcontractor performance. Indeed, they are able to do so only because they are one of the few companies that has a quantified handle on their own performance.

Running the diagnostic with this group, we split the responses so that we could compare the development team to the test team, and both to the outsourced team offshore. The question we always start out with is to get people to identify, from a list, their major sources of pain. In this case, the strongest response from the development and test teams was that their subcontractors were causing the most pain. Interestingly, for the group offshore, it was the customer that caused the greatest spike in responses. For a group with one of the strongest outsourcing relationships around, there were certainly some remaining challenges to deal with: both sides perceived that it was the other side that was the challenge.

The second case is with another company located only a short distance away from the first group physically, but almost diametrically opposed to them in terms of performance. This group had been struggling for some time to deliver quality product, and had literally been spending the lion's share of their time dealing with client-reported issues. Here, when asked about their greatest issues, the interesting responses came from a selection called 'other', where respondents could answer in freeform text. While the technical people indicated there were issues that they could deal with, such as a need to do more testing or a lack of understanding of the user's needs, management responses fell into a different category. All their responses indicated that the problem was outside of their sphere of influence: the war in the Middle East, the weakening economy, the weakening US dollar (this, mind you, was several years ago). With that attitude, all they thought they could do was throw up their arms in despair.

This was unfortunate as there really was quite a bit they could have done to better manage their circumstances.

We could be refining our already strong performance and dealing with minor tweaks, or battling stronger demons in a fight for survival. In either case, it is in our best interests to be able to step back when we look at issues to include our own contributions as part of the mix. If we look at all the respondents to the diagnostic to date, 38% have indicated that their customers or contractors were a significant issue they had to deal with. In how many of these could they have stepped back and considered that there were things they could do to better manage that relationship?

In any challenged relationship, there is always something that we could have done differently. The fact that we are in the midst of such a struggle should give us pause to consider our own behaviors. Indeed, one of the most powerful tools I can bring to work with clients is an objective mirror.

We need complete, open communication

Failure to communicate is at the root of almost all challenges we face in software development. Process and procedures are a means of ensuring that we do the right things at the right time. Analysis models help us communicate different aspects of our product in more precise forms than the English language can convey.

Beyond these engineering solutions to communication issues, we also have to deal with the nuances of relationships and teamwork for a complete solution.

Understanding individual motives and honing our skills in active listening and conflict resolution make it still easier to communicate, but there is still more: we need to engage others with candor.

Communicating with candor is more than merely telling the truth. Candor involves full disclosure of all information that you are aware of (and indeed, disclosure of the areas where you don't have all the information), both positive and negative. In order to do this, everyone on the team needs to feel safe in their environment. If there is any lack of trust among the team, or any issues that have not been completely dealt with in the past, it can become too easy to hold back. Candor and hidden agendas are like water and chocolate (if not in that order).

Some people have the self-confidence to be able to walk into almost any situation and speak with complete candor, but it is a very rare team where everyone exhibits this trait. In the stages of team development (forming, storming, norming and performing), it is only when a team has achieved the performing stage that we have built the infrastructure of trust, of openness and caring that supports candor. For teams that have been there, most know that it is all too easy to backslide. It takes effort to get to the performing stage, and effort to stay there.

Why is this important? It is at this stage where the team is firing on all cylinders, working together effectively as a group, and having a good time in the process. With complete candor, the shared memory of the team is more complete, and more consistent. Fewer balls are dropped. Improved efficiency opens the door for the team to be more creative.

Agile principles suggest the most efficient and effective method of communication for a development team is face-to-face conversation, where candor is a must. If you are not consciously maturing your team dynamics to the point where candor is possible, even your agile projects will be at risk.

We are a core part of a larger machine

As our team size grows, we compartmentalize ourselves into specific roles: project manager, scrum master, developer, tester, and a wide range of others. With this often comes the assumption that each role has the responsibility to produce specific products or artifacts: a project schedule, a specification, some code, or something similar. Problems arise when we take this to mean that we are the sole proprietors of the products of our roles: that we have the responsibility of doing it ourselves.

The experience of producing one component of a larger system can range from harrowing to extremely rewarding, or anywhere in between. If we have good information about what that component should be, supported by previous experience, it can be deceptively easy to get the job done. Work breakdown structures help us remember everything we need to consider when building out a project schedule. Document templates give us an outline that allows us to simply fill in the blanks. Even the defined structure of a daily scrum meeting gives us guidance on who needs to participate (everyone) and what needs to be covered.

Unfortunately, when we are doing these things on our own, it is easy to fall into the trap of following the letter rather than the intent. For collaborative events such as scrums, we can still think of this as a DIY activity if only the scrum master considers whether this collaboration was achieved. With one person thinking about whether something is done, it is easy to satisfy our own preconceived notions of completion. When working to some deadline, the task often becomes an exercise of making sure that all the fields are filled in with something within the time constraints. There might be time for polishing if you are lucky enough and there aren't any other pressing tasks to deal with.

We then check off the completion box for that task, but are we really done? As others are in the same boat, review is often superficial. Deep issues only arise much further downstream. For most tasks on a project, if you were to hand them out to be completed independently by different people, you would find that most would complete the task and believe they had done a good job, but provide wildly different solutions. A great example of this is a vision statement: a simple expression of who the key client is and what problem is being solved, along with an identification of the key competitor and the primary differentiation. Ask each person on the team to develop one independently, and you are guaranteed to get very different results, each one being complete. Which is the correct one?

This is a problem.

If done in a collaborative fashion, the dynamics are completely reversed. The differences of perspective are exposed early, highlighting the need for further discussion and clarification. The team works together to come to a common understanding, rather than the team inheriting one viewpoint and having to live with it, as usually happens. Done this way, these discussions occur with the right people at the table, at the right time in the project. Compare this to the late-in-the-project discovery that there was no alignment in vision, that most of us have experienced.

As we split into more detailed roles on a project then, we need to think of our tasks as things we are responsible for ensuring they are done, rather than things we need to do ourselves. In this way, we make sure that different perspectives are appropriately balanced. With broader participation, everyone has a stake in the production of the artifact. With that stake comes a stronger feeling of ownership: it becomes 'our plan' rather than 'his plan', or 'our tests' rather than 'his tests'. Big difference. It helps break down the silos between departments, the handing-over-the-wall of artifacts, and the carrying down from the mount of the stone tablets that drive our work.

The whole group becomes responsible for a successful project. If one person has the lead in ensuring something gets done, that person should never be alone in that activity. Nobody likes to be told what they should be doing, or what they have to do. It is far better to be part of the decision-making process, to have the opportunity to see what is needed and step up and volunteer for the task. The stake and ownership itself will tend to provide success far more often.

The counter-argument of not having enough time to collaborate on these things just doesn’t fly. I’m not suggesting that everyone participate together for every component of every task, but anyone affected by a decision should be involved in the decision-making process. If we do things on our own, the collisions downstream because we haven’t involved others will cost us far more in the long run. It is these costs that we never account for in our original planning.

Regardless of your role, don’t get caught in the trap of thinking that you are solely responsible for the products of that role. A better approach is to be sure that the decisions you are responsible for actually get done, but get done and agreed upon taking different perspectives into account. The perspectives of others can only be clearly understood if they participate: don’t do things yourself.

We need to respect the effort behind change

We all deal with change in our lives, generally doing so by avoiding it at all costs. Change is not trivial topic to deal with, which is likely one of the reasons it is so intimidating to all of us. While I often discuss change in the context of the learning curve for training, I find it more appropriate to think of change outside the context of software development. I have been struck by the applicability of the Satir Change Model to software teams.

Virginia Satir (a family therapist, but the analogy is apt, yes?) suggested “Familiarity is more powerful than comfort”. That seems to be the case for the software industry as well. There is some overwhelming inertia holding the industry back from realizing its potential, despite our not being in the comfort zone.

One of the best introductions to the Satir model comes from Steven Smith (1), one of the keynotes at this year’s conference. The model is simple and clear, but most importantly makes sense because we can instantly see ourselves in the model. We naturally have an affinity for the status quo. For myriad reasons, change is something we detest. Quitting smoking, traveling to exotic places, developing software in a way that differs from the past (in some places, that 'past' can span decades). There’s the unknown, the anticipated chaos, the displacement from our current way of doing things. It is easier to just stay where we are, even if we can envision a better place at the end. As a physicist I know how to express the strength of inertia through formulae. From the perspective of change, we all feel the inertia keeps us in our place of familiarity, whether or not it is comfortable.

Change is far more than simply deciding to do things in a different way. The greater we appreciate and understand all the complexities and nuances, the more effective we can be at fostering effective change at home and in the workplace.

Most critical of these issues is the recognition that change is embodied in a clear sequence of stages. For meaningful change to stick, you can expect to go through all these stages in turn. Deep change will drop us out of our comfort zone, our status quo, and carry us through a stage of disruption and chaos. Quite often we will see this as bad, and leap back to our old ways, back to that soothing place of familiarity. It takes significant effort to continue on our path, to find some mechanism or transforming idea that allows us to recognize the value of seeing the change through. If we have done well, we will reach a point where we have been successful in changing. This can be a tough journey, and at each stage of change there are new reasons to fall back to where we started.

Often, though, we attack change in ways that make the journey tougher than it needs to be. At one conference, it was interesting to note that several people saw their role as one of a change 'inflictor', and indeed this is the way many people foster change as consultants. "You guys need to adopt Scrum...” or "I can help you get to CMMI Level 3...". Recognizing the sequential nature of change is important for accepting that it’s not as trivial pointing someone in a certain direction. There are other ways to simplify things, to make change easier to swallow.

A critical element of driving successful change is controlling the magnitude of change. Often, in software development or in life, we take on ludicrously large elements to change. We’ll go from a totally chaotic ad-hoc development style (often in the form of every participant using their own preferred approaches from previous jobs), to a complete single way of developing software. While great in theory, this is extremely disruptive in its implementation. Whether agile or not, this complete system is made up of a collection of individual elements, each with their own change profile. Some of these may have little chaos and huge return, while others may be extremely chaotic, and actually have a net negative impact. Throw them all together, and you will have washed out much of the value of the best elements, and imposed an overall change that can be quite daunting. The team can easily find the status quo quite attractive in comparison, and slip back into old habits.

We need to find those atomic elements that will provide us the most value with the least disruption. If we select them correctly, we will find far less pushback, and faster adoption. We also benefit from demonstrating to the team that this change stuff doesn’t have to be so painful after all.

The other side of simplifying or making change more attractive is cranking up the engagement from the participants. We have to be careful to present the potential change in a way that is clearly understood in terms that people understand. In addition to this, we need to avoid simply handing this change over as a task for them to do. While we may not see these elements as disruptive, this is because we have already made the required journey. We have already internalized the value and overcome the disruption.

Any change will be easier if the participant is carefully supported, given the time to acclimatize to the different way of doing things, reminded of the value of the change in making their life easier. Participants need to understand how the change brings them more closely into alignment with their own value system. They need to see what is in it for them, and this attraction will vary dramatically across the team.

In the minds of most people, change is frightening and diametrically opposed to remaining in the status quo. If we are careful in how we present change, selecting the low-hanging fruit and fostering the change through our teams, we can dramatically reduce the barriers we often face. Indeed, we can get to the point where change becomes an attractive ongoing way of doing business. We can become a true learning organization, and remove the apparent dichotomy between change and the status quo.

We all need to see the value from our perspectives

Some things sell themselves. Tell an eleven year old that Apple has a new iPod on the market, and she'll save her allowance money for months. The same goes for seven-year-old boys and Pokémon cards. If you have kids around that age, you will certainly understand where I am coming from.

The same principle holds true in the software industry. Tell a young group of developers that you are going to give Scrum a try, and you are not likely to see a great deal of resistance. They are all over it like...well, pick your favorite attraction metaphor here. Get a bigger group of more established (read: older) practitioners and wave the CMMI in front of them, you will often see a similar response. Give almost anyone in the industry a software tool as a means of solving a problem they are dealing with, and…you get the picture.

The problem is that it is not quite as easy to sell these things to everyone on the team. Simply waving the brand around is insufficient to capture the entire audience. You need to engage that whole audience in order for many practices to truly work. There is still hope for bringing good practices into your project, but you need to step away from the hype.

Senior management and many marketing people aren't interested in agile approaches. In fact, from what they have heard they are probably afraid they will have less control and visibility into what is being built than before. Unfortunately, these rumors are likely to have come from weak implementations of good practices in the past. On the other hand, if you talk about giving them a continuous stream of new features, keeping the product running so there is no big-bang integration, and giving them a voice in the prioritization of these new features, you will hook them quickly.

Talk to an end user about a use case workshop, and their eyes will likely glaze over. Ask them (without jargon) about the different kinds of users for the product, and what each of these users will want to achieve, and you will have an engaged crowd. Dive into discussions about the steps that make the most sense for them to interact with the system to achieve their goals, rather than the normal flow. You can even discuss alternative ways they might achieve their goals, what they might expect when for some reason they can't get to where they want to be, and what state the system should be in before you get started and when you are done, and you have pretty much covered the breadth of use case issues. More importantly, you haven't scared them off with the terminology.

Tell a development team you are seeking a CMMI rating or ISO certification, and you will have a bunch of disillusioned people expecting to waste much of their precious time building needless documents. Again, perceptions based on experience can be strong. Talk about getting together to agree on an approach that will improve predictability on your projects, simplify planning, ease maintenance grief and shorten schedules to boot, and you might get them to sit up and take notice.

If you have a conflict with a co-worker and suggest that the two of you step through a five-step approach for conflict resolution, and their opinion of you will likely drop lower than it already is. Head into the discussion ready to learn something about the other person's perspective and get them to acknowledge that you really do see where they are coming from before proposing solutions to the challenge, and you'll be amazed by the connection you make.

Whenever we work with others, we need to sell reasonable approaches to solving problems first by understanding our audience. There is a strong chance that they are not motivated in the same ways you are, and it is critical that you know WIIFT – what's in it for them.

Once you know that you are well on the way. Pitch to their needs; show them how these approaches address their needs, and the pushback that often occurs with change will dissolve. As an added bonus, this approach will help keep you honest about doing the things that are right for the team as a whole. It will prevent you from being swept up in the hype surrounding trends that might happen to be in vogue today. When you are selling, you need to play to the needs of the audience.

It really is all about you

Most change is driven by selection of new tools or techniques, and is offered up as a point event to the team, with the expectation that there will be immediate improvement within the group. While some of these efforts can have a positive impact, we are not controlling all the variables in this equation.

We need to take the time to focus on the team dynamics, and manage these dynamics such that the group interacts more cohesively. With this approach, any of the usual changes we apply have a much higher likelihood of succeeding and providing even more value than before. Indeed, with the team working together effectively, we can find dramatic improvement without the addition or change of tools or techniques, and we can select changes that are more aligned with the group's needs.

References:

(1) http://www.stevenmsmith.com/my-articles/article/the-satir-change-model.html

THE 2008 STATE OF REQUIREMENTS MANAGEMENT REPORT.

The results of a recent industry survey shed light on the latest trends, challenges and solutions in software product development for 2008.

Written by Jama Software co-founders:
Eric Winquist, ewinquist@jamasoftware.com and John Simpson, jsimpson@jamasoftware.com

Publication notice:
Previously published to survey participants and Jama Software customers.

OVERVIEW

Reality or hype? Discover what product teams are really doing.

Whether your role is Product Manager, Project Manager, Business Analyst, Development Director, QA Manager or Chief Requirements Guru, if you are involved in the planning and development of products, software applications or systems at your company, this report applies to you. The goal of this report was to survey professionals in software product development and to provide a real-world perspective.

Gain insights into:

∉ What are the biggest challenges in innovation that companies face?
∉ Where are companies getting their next great product ideas?
∉ What are the top barriers to success?
∉ Which metrics matter most when measuring success?
∉ Is the Agile process over-hyped?
∉ Which tools top the wish list in 2008?
∉ How does collaboration apply to requirements management?
∉ What frustrates people more – scope creep, unrealistic expectations or lack of testing?
∉ Which genre of music is most popular? OK, that one we threw in just for fun.

Take ten minutes to read the full report and learn more about the latest trends, challenges and solutions that other organizations are focused on this year. Some survey results you might expect, others might surprise you. Either way, it’s time to cut through the hype and uncover what teams are really doing to successfully plan and develop new products in a customer-driven economy.

ABOUT THE SURVEY

This survey was conducted by Jama Software in partnership with Ravenflow. The report includes data collected from 203 survey participants from April 15 to May 9, 2008. Professionals were invited to complete the online survey in return for a copy of this published report. For privacy, all survey participants, responses and comments remain anonymous in this report. Ninety percent of participants completed the entire survey. Survey participants represented a world-wide audience and a diverse sampling. Thanks to everyone who participated. Here are the breakdowns by role, company size, industry and average project team size.

Role in the Organization:

40%	Business/Requirements Analyst
19%	Project Management
10%	Product / Program Management
9%	Product Development / Engineering
8%	Outside Consultant
5%	Executive Management
3%	Research / Usability / Design
7%	Other

Company Size (Annual Revenue):

26%	Greater than $1 billion
14%	$500 million - $1 billion
17%	$100 - $500 million
14%	$25 - $100 million
29%	Less than $25 million

Industry:

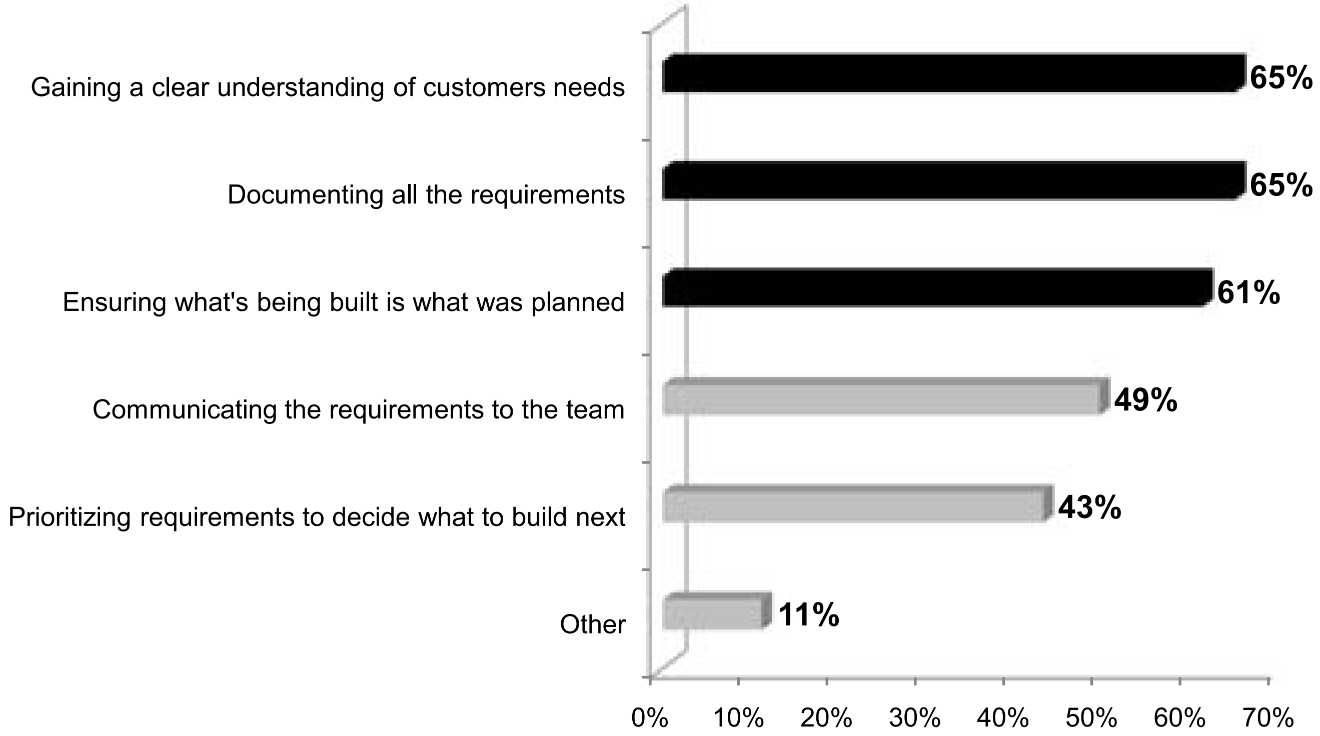
28%	Technology / Software
17%	Financial Services / Insurance
17%	Aerospace / Defense / Government
11%	Healthcare / Medical Devices
8%	Telecommunications / Media
3%	Automotive / Consumer Products
3%	Energy / Chemical / Utilities
12%	Other

Team Size (Project Team & Stakeholders):

61%	Less than 25 people
28%	25 – 50
6%	50 – 100
3%	100 – 250
2%	More than 250

QUESTION

In your opinion, what are your company's biggest challenges when it comes to innovation? (Mark all that apply)



SURVEY RESULTS

Three fundamentals of requirements management top the list.

The buzz around "innovation" is everywhere – in the news, at events, on the Web. A Google search will deliver up over 150 million pages related to innovation. For context, that's more than Britney Spears. Despite all the enthusiastic chatter, innovation is easier said than done.

What are the real challenges that teams face when developing products their customers really want?

One thing is clear – in order to innovate successfully, you must manage requirements successfully.

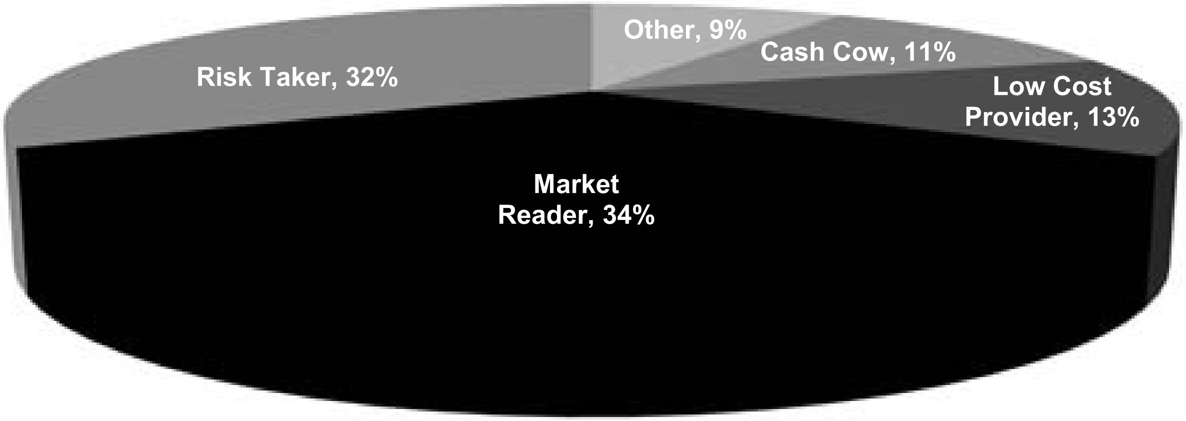
As the data shows, the top challenges map to three fundamentals of requirements management – gaining a clear understanding of what customers want, documenting all the requirements and then ensuring what's being built is what was planned. There's no substitute for the fundamentals.

"Managing the rapid change of requirements and traceability is our toughest challenge."

– Survey participant

QUESTION

How would you characterize your company's approach to innovation?



Risk Taker – We seek to be first to market with breakthrough ideas.

Low Cost Provider – We focus on operational efficiencies by delivering similar products at lower costs.

Market Reader – We try to be a fast follower, and focus on incremental improvements.

Cash Cow – We try to milk as much revenue as possible from existing products.

SURVEY RESULTS

When it comes to product innovation – speed does matter.

A recent McKinsey Research study [1] shows that over 70% of senior executives say that innovation will be at least one of the top three drivers of growth over the next three to five years. So, what approaches to innovation are companies taking?

Be first to market with a breakthrough product or be fast to follow with a better one – that's how the majority of those we surveyed characterize their company's approach to innovation.

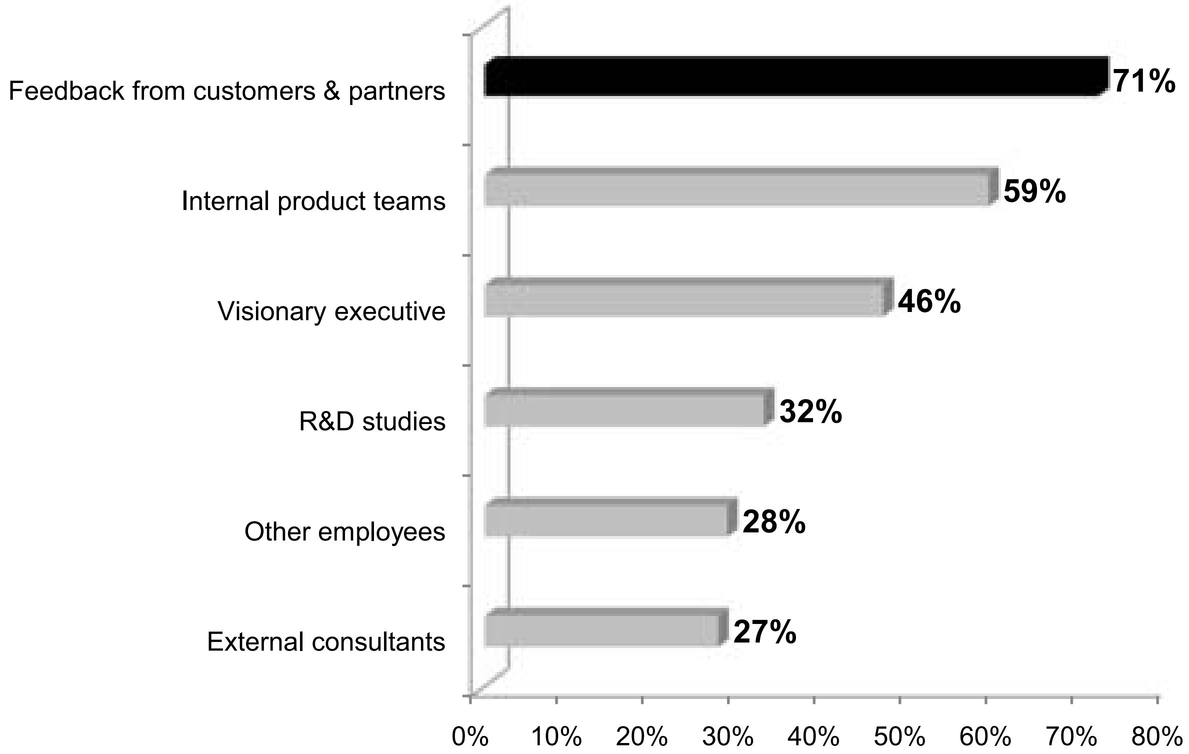
Did different industries answer differently? What about the size of a company? Surprisingly, neither size nor industry had a significant variance when we filtered the survey results. The common theme represented by the responses and comments provided is that whether a risk taker or a market reader, the majority of those surveyed viewed rapid product development as a key driver to their ability to innovate.

> **"We're a fast follower. We watch the competition closely and then make it better."**
>
> – Survey participant

QUESTION

What are your sources of new product ideas and requirements?

(Mark all that apply)



SURVEY RESULTS

Think R&D studies hold your next great idea? Ask your customers.

A similar "a-ha" is occurring at companies everywhere – they are embracing the fact that customers are willing to openly share their ideas and participate in the product planning and development process. It's less expensive, real-time, and as unfiltered and pure as a good Hefeweizen.

In the world of customer-driven product development, it's a trend that's been underway for several years, but it's recently hit another gear with the explosion of online customer communities and Web-based collaboration tools.

As this data illustrates, the #1 source for new product ideas and requirements is feedback from customers and partners.

"Lavish R&D budgets don't deliver better performance. Customer focus does."

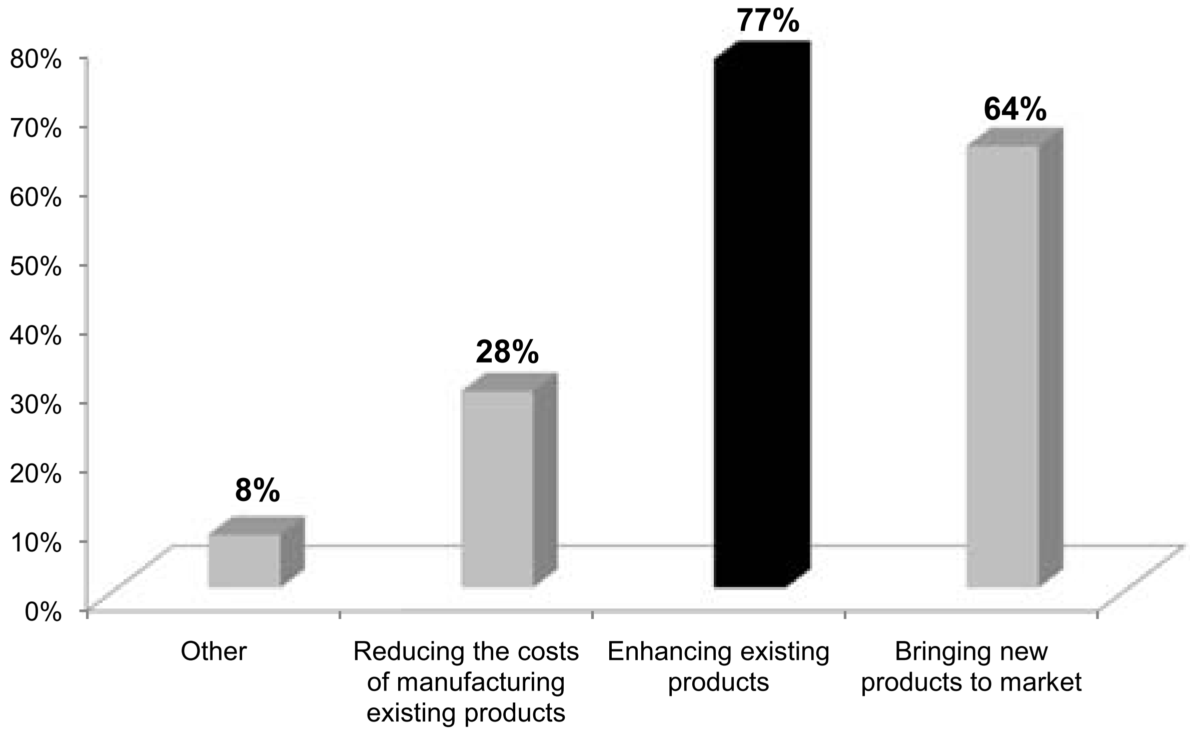
– Booz Allen Hamilton,
Global Innovation 1000 Report [2]

Does this mean traditional R&D goes away? Does it mean your visionary executive takes a back seat? Not necessarily. It simply means that companies that achieve greater alignment with their customers achieve greater results. Your customers are leading the conversation. Are you listening?

QUESTION

What are the goals of the projects your team works on?

(Mark all that apply)



SURVEY RESULTS

Enhancing existing products outweighs developing brand new ones.

Is this surprising? Maybe not. New products tend to grab the spotlight, but they also tend to be slower to develop, more expensive and higher risk. Companies are finding success through smaller, more focused releases with incremental enhancements over time.

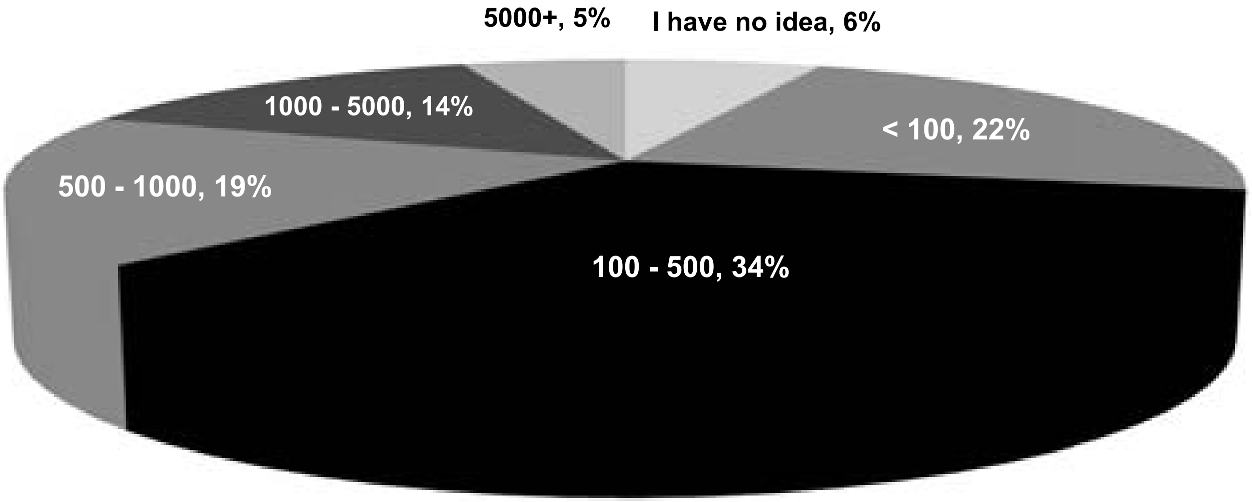
These survey results support the trend toward more and more product development teams adopting the philosophy of "release early and release often".

What we found surprising was that only 28% answered that "reducing costs" was a goal. You read and hear a lot about efficiency being a top initiative especially during tougher economic times, but these survey responses didn't reflect that.

> **"Our goal is to deliver quality software in a reduced timeframe using an iterative approach to development and systematic testing."**
>
> – Survey participant

QUESTION

Let's talk about complexity – on average, how many requirements does a typical project or product contain?



SURVEY RESULTS

No one ever said this job was easy. I hope you like requirements.

Maybe we should run a contest for the largest requirements specification document – 800 pages, a thousand pages? For the 5% whose projects on average have over 5,000 requirements each, there's a Guinness World Record just begging to be set. Yes, we checked and there isn't one yet.

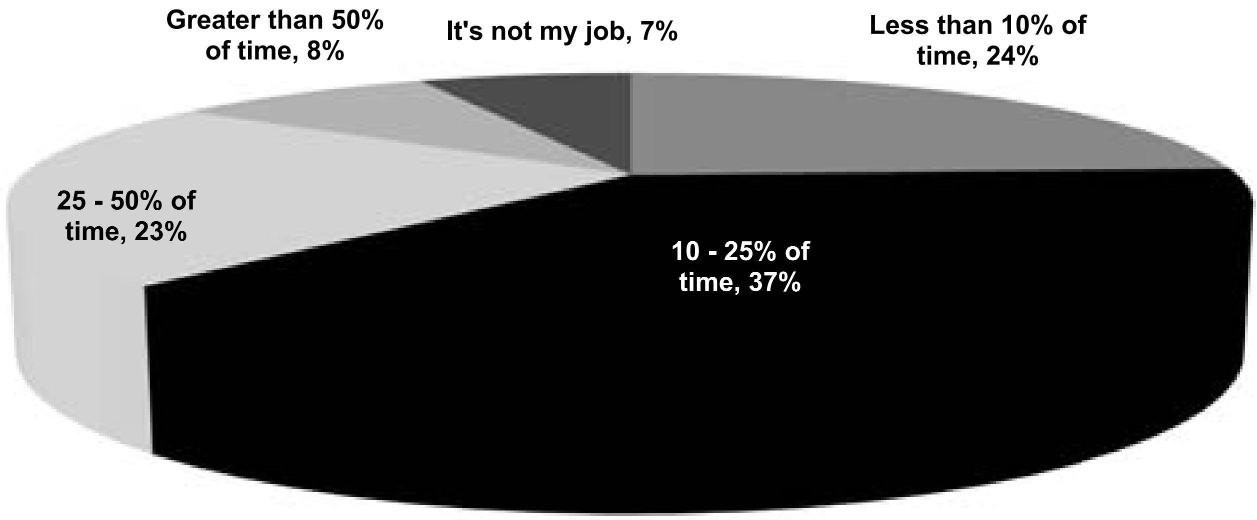
This question provided some interesting segmentation.

As you might anticipate, the more complex the projects (meaning we filtered on those with 500 or more requirements per project), the lower the average success rates, the greater the time spent managing changes to requirements, and the greater the interest in using requirements collaboration and management software.

Does the size of the team have any correlation to the size of projects? Yes, the bigger the team, the bigger the projects. Whereas overall 72% of those surveyed averaged at least 100 requirements per project; for those teams with 25 or more people, 90% averaged 100 or more requirements per project.

QUESTION

What percentage of your time is spent each week dealing with changes to requirements?



SURVEY RESULTS

The vast majority spend at least 10% of their week managing changes.

Oh man, you've got to really feel for the 8% that spend over half of their time just managing changes to requirements. It's a reality of product development though, customer needs change.

So, how can you best manage the change and keep everyone in sync without killing yourself?

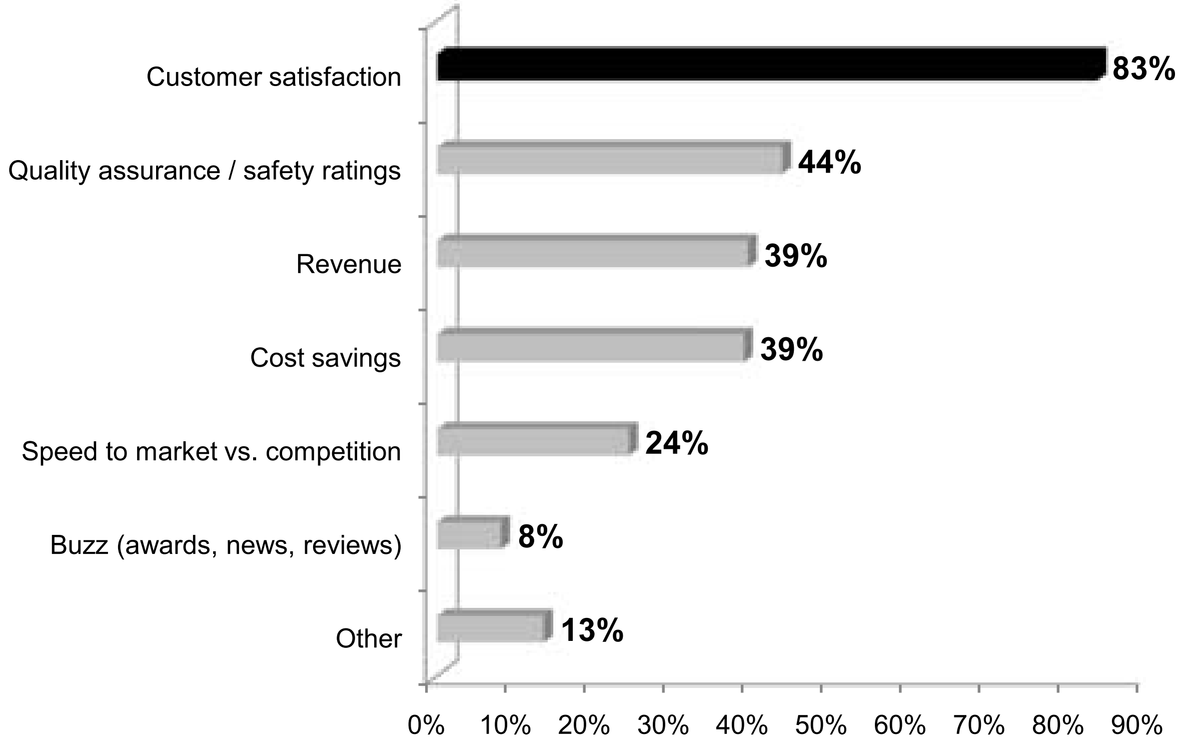
> **"Managing the rapid change of requirements and traceability is our toughest challenge."**
>
> – Survey participant

As Forrester Research [3] defines it, that's where requirements management solutions help by:
1) Storing requirements in a central location
2) Tracking relationships among requirements and artifacts
3) Controlling changes to individual requirements and groups of requirements

This was another interesting question to segment the results on. When we look at those that spend at least 25% of their time or more managing changes, success rates were lower, the #1 challenge shifted to "ensuring what's being built is what was planned" and the interest in requirements collaboration and management software increased to 80%.

QUESTION

How do you measure the success of your delivered projects/products? (Mark all that apply)



SURVEY RESULTS

Customer satisfaction outshines revenue and other success metrics.

Surprised by this answer? Why isn't revenue higher?

This is a question where role plays a factor. For business analysts and project managers, which represent 59% of those surveyed, customer satisfaction reigned supreme. For product managers and executives, revenue was the top metric they cared about, with ROI being a popular write-in theme for "other".

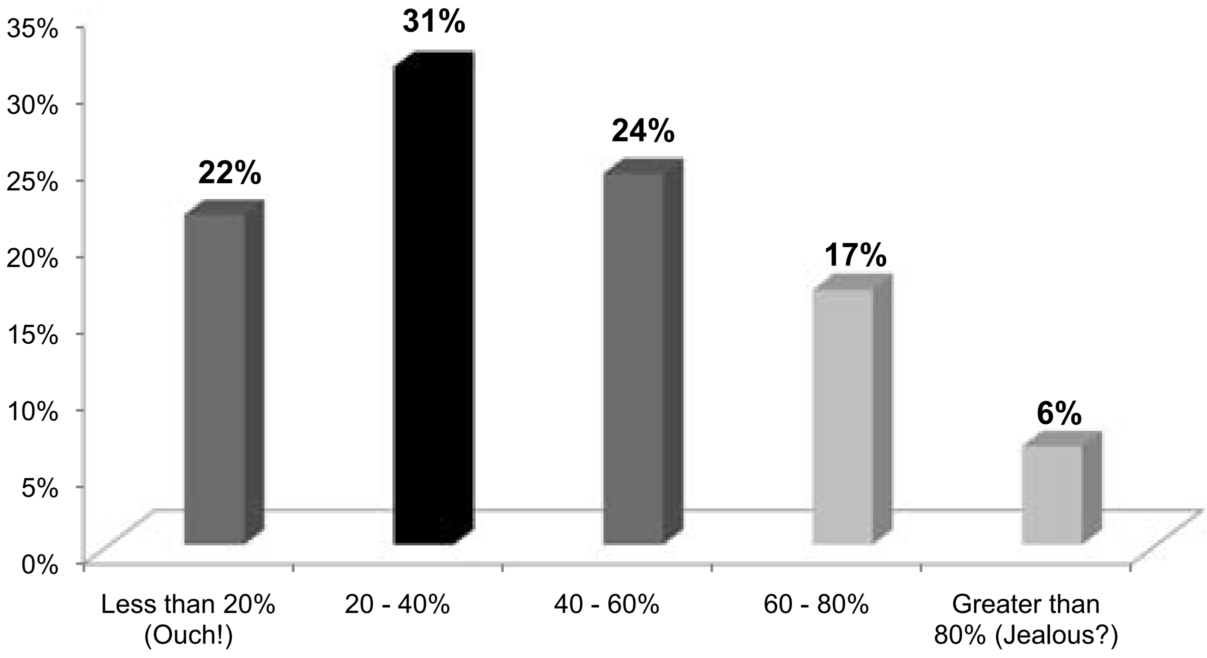
These results speak to the interesting dynamic that exists between project management and product management.

As Jeff Lash [4], the author of the blog, *How To Be A Good Product Manager*, writes, "To avoid conflicts between project management and product management, product managers, project managers, and project teams should all agree on shared goals and metrics as much as possible."

> **"Success for us is measured by the return. What is the ROI back to the business?"**
>
> – Survey participant

QUESTION

How often are the projects or product launches that you're involved with delivered on time and on budget? Let's be honest now…



SURVEY RESULTS

For the large majority, success rates are 60% or lower.

It's like Shaquille O'Neil shooting free throws – you expect better, but the reality is you're lucky if just 60% of the attempts are successful.

Why are these success rates what they are? Are we all just numb by the same old story about project failure? Billions are lost each year on bad software. Got it. Delays in product development can bankrupt companies. Yep.

In the IEEE Spectrum article [5], *Why Software Fails*, it suggests, "The biggest tragedy is that software failure is for the most part predictable and avoidable. Unfortunately, most organizations don't see preventing failure as an urgent matter…"

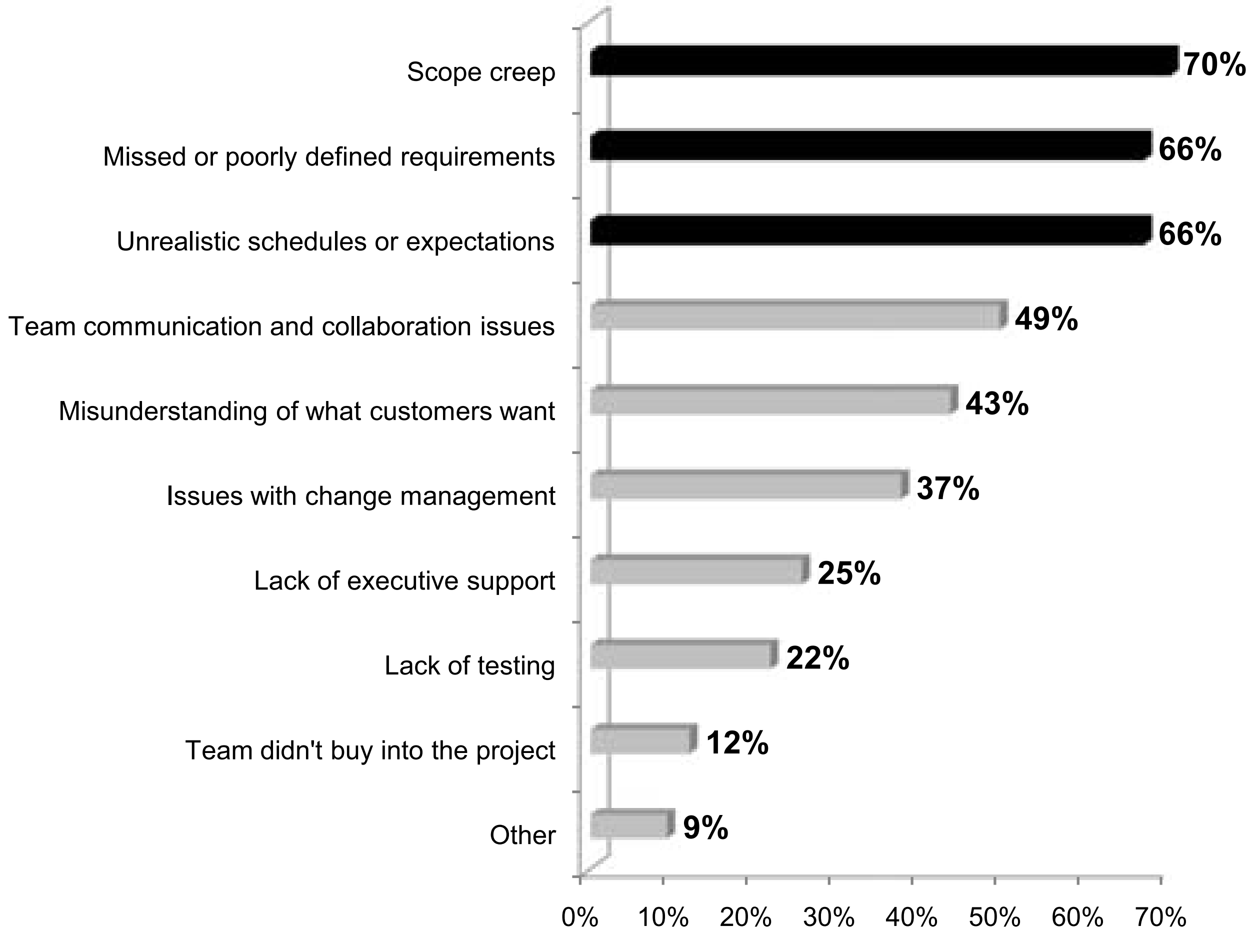
That may be true, however something tells us it isn't just organizational complacency, but that there's much more to it than that. In the next question, we explore the leading causes for failure.

"It's all about an on-time delivery. Did we meet the target date?"

– Survey participant

QUESTION

When a project/product is NOT viewed as successful, what typically are the causes? (Mark all that apply)



SURVEY RESULTS

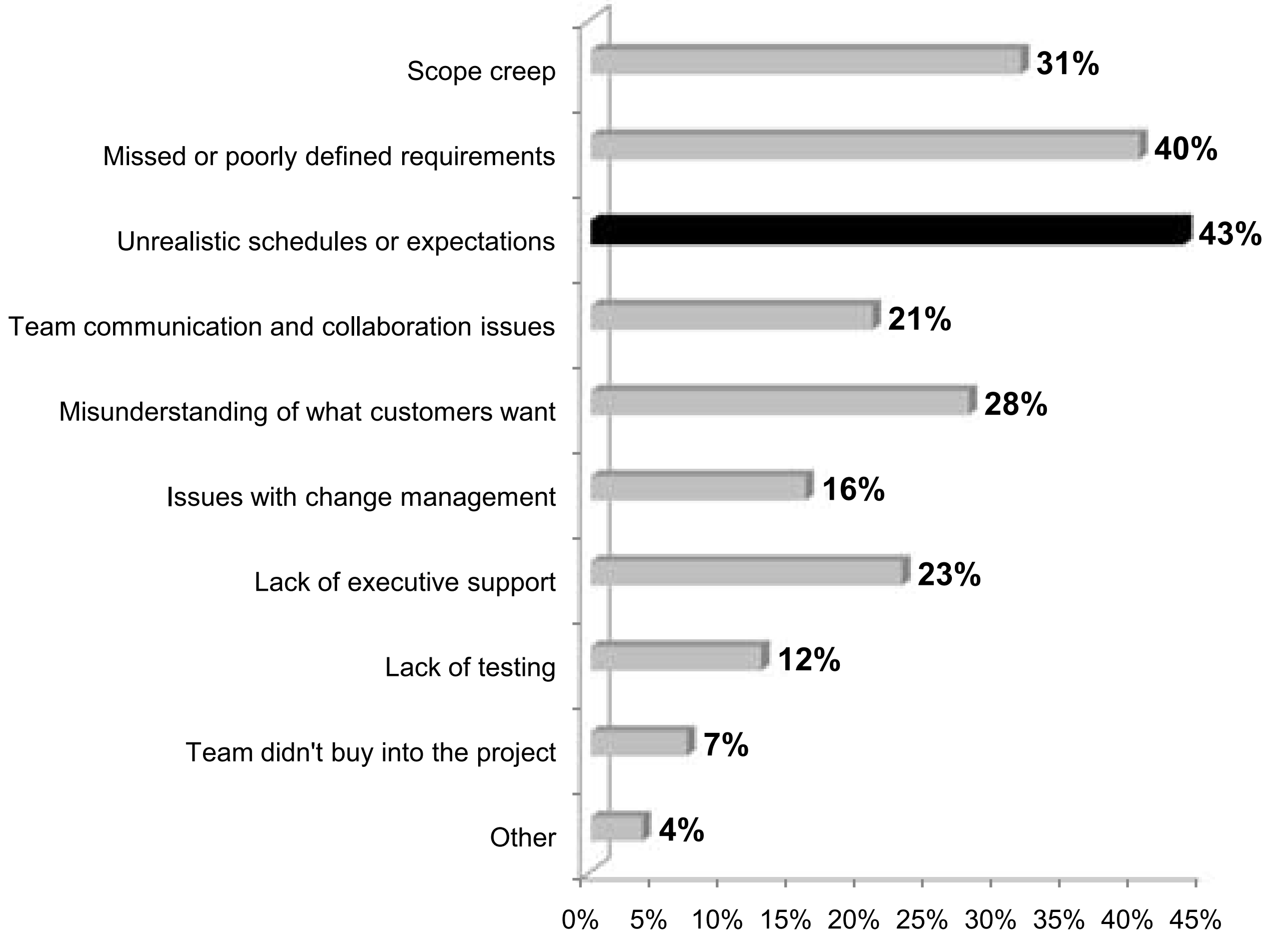
Beware of the dreaded “scope creep”.

It lurks by the water cooler, on customer status calls and in team meetings – it’s the dreaded scope creep and it wreaks havoc on projects. It’s not alone though, tied for a close second are its nasty cousins “missed or poorly defined requirements” and “unrealistic schedules or expectations”.

How do you avoid these? Tools can help, process is critical, but more than anything else it takes really skilled people to keep these issues in check. Otherwise, these issues will continue to creep up (no pun intended) and create unnecessary frustration, delays and costly rework for organizations – all of which lead to failure.

QUESTION

Of these barriers to success, which ones do you PERSONALLY find the most frustrating? (Mark all that apply)



SURVEY RESULTS

“Unrealistic schedules or expectations” drive people crazy.

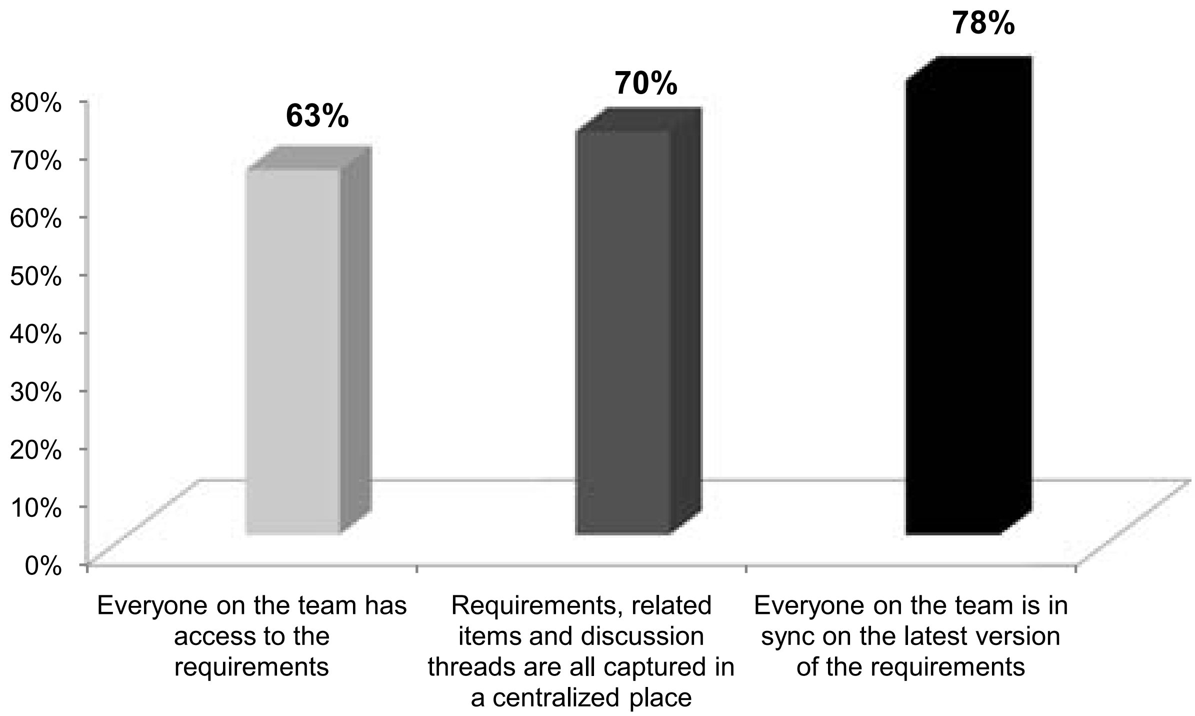
Even though scope creep was the top cause for failure, “unrealistic schedules or expectations” takes the top prize in what professionals personally find most frustrating.

Some barriers you can overcome mid-project, but when unrealistic expectations or schedules get set and approved, it’s difficult later to hit the reset button with stakeholders and customers. It’s a lesson even the most experienced product development teams have experienced.

> **“Some of the biggest overall problems come from pursuing what the customer says they want, without determining what they really need.”**
>
> – Survey participant

QUESTION

“Collaboration” is a word being talked about a lot. In your opinion, how does collaboration apply to requirements management?



SURVEY RESULTS

Well, actually collaboration applies to all of the above.

Next to innovation, “collaboration” might be the second biggest buzzword in business right now. So, what does collaboration really mean as it applies to requirements management? We were curious too, so we asked the question.

As the survey results show, no one clear answer stands out. Essentially, collaboration embodies all three of these things – team-wide access, centralized place of all assets, and continuous alignment to the latest version of the requirements.

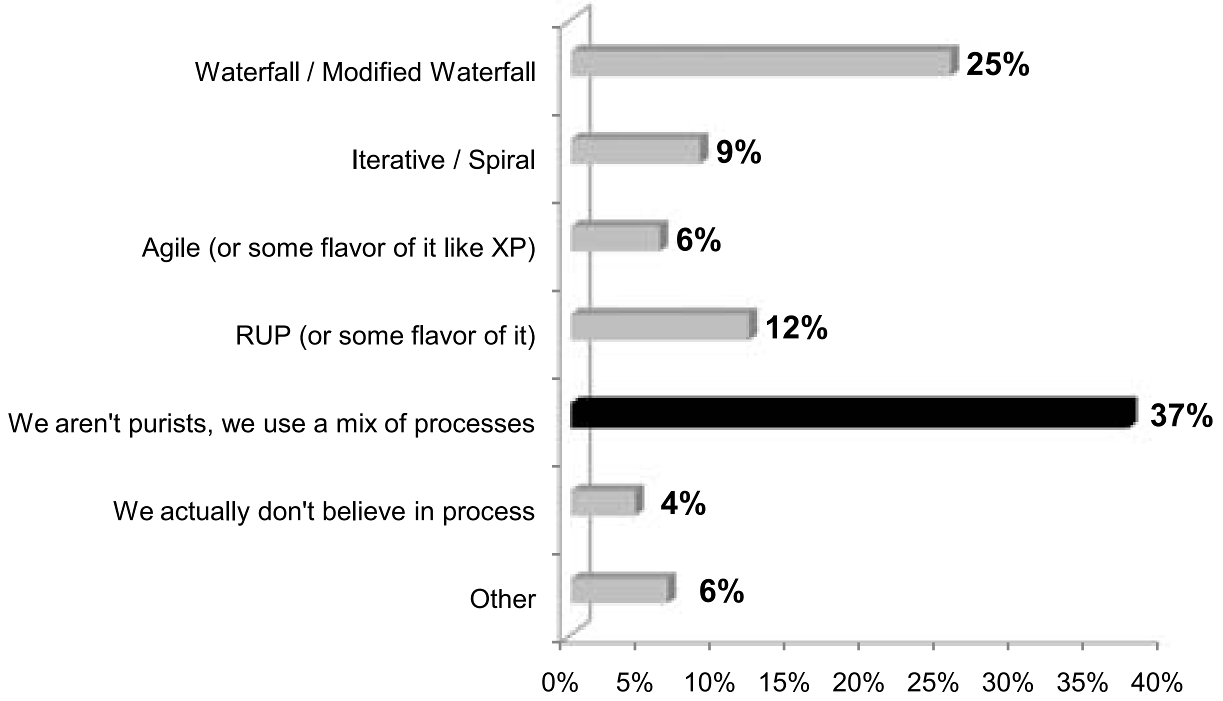
Some experts view collaboration as one of the key ingredients to being more successful with requirements management. And, based on our own personal experience, we agree. A collaborative approach is a faster, more successful way to stay in sync throughout the planning and development cycles – both internally (with your team) and externally (with your customers and partners).

“Requirements management is a communication process. Collaboration happens when everyone has the same understanding of the requirements.”

– Survey participant

QUESTION

Which process does your team use?



SURVEY RESULTS

Surprised? Only 6% are pure Agile shops. Many use a mix.

There's no denying the momentum that Agile has in the product development world, but is it overhyped? At Jama, we use a modified Agile process ourselves and have used various processes before, so we were curious to learn what other teams are really doing.

As the data illustrates, the largest segment is actually using a mix of processes. A quarter of survey respondents are using a traditional or modified Waterfall method exclusively, but few beyond that are purists of any one process.

So, as one survey participant pointed out, "It's important for the tools to be flexible to adapt to whatever processes your team or company uses, because inevitably they will change."

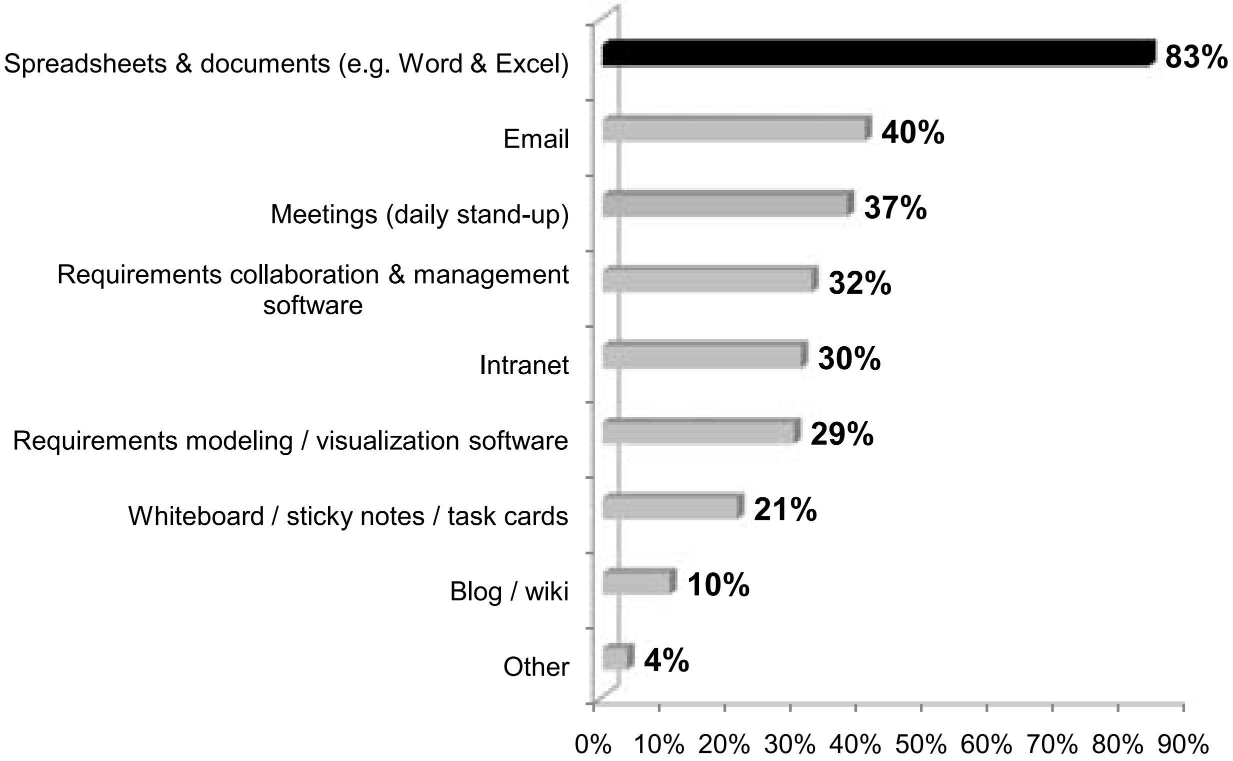
These survey results and our own experiences confirm that no single process is a silver bullet. Different projects, different products, different teams – they require different processes. Adapt and survive.

"It depends highly on the project characteristics, so it varies from Waterfall to Agile."

– Survey participant

QUESTION

How does your team currently document and communicate requirements? (Mark all that apply)



SURVEY RESULTS

Help! We're stuck in the land of documents and spreadsheets.

It's pretty amazing when you think about it – the tools (e.g. Excel spreadsheets and Word documents) that your kids might use to do their next homework assignment are the same ones professionals use to manage massive software development projects.

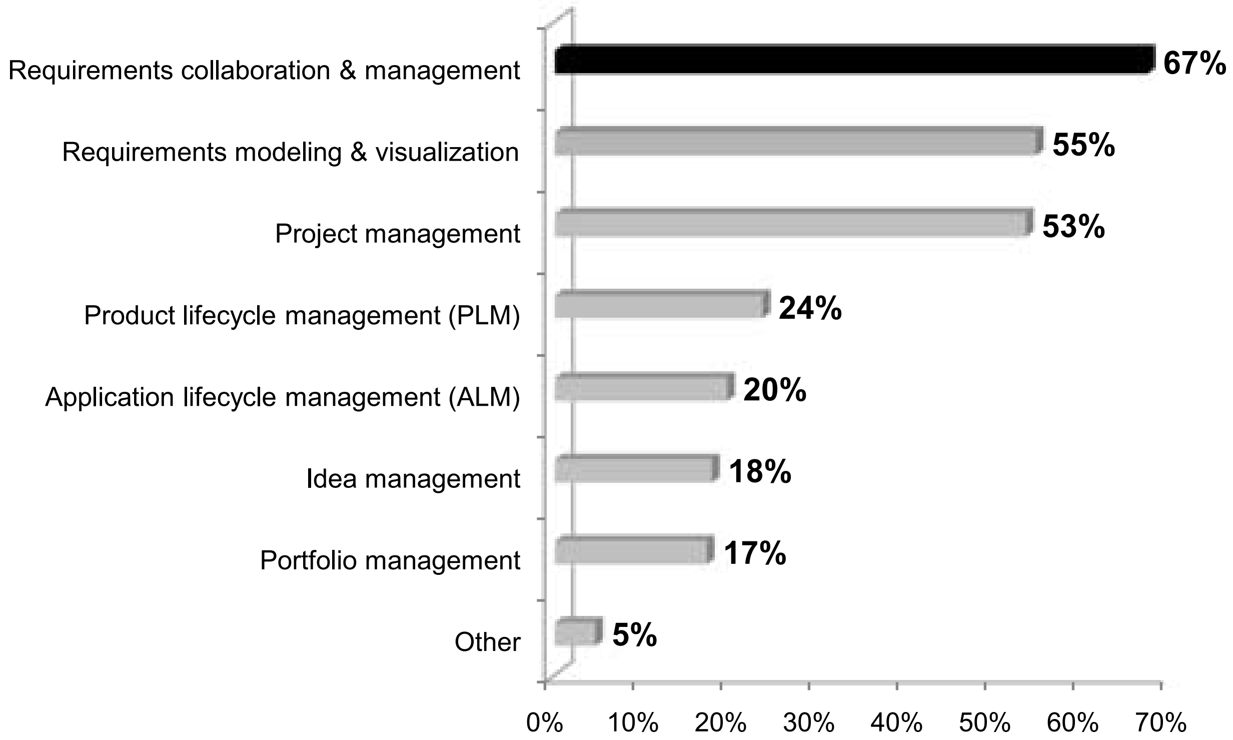
Sure, these tools are ubiquitous and we all know how to use them, but are they really the best way to capture and communicate thousands of requirements for complex projects with distributed teams?

As this survey data supports, more often than not, business analysts and project managers rely on manual effort and Microsoft Office to accomplish the documentation and communication of requirements.

But, as Forrester points out in their recent Wave Report [3] for application development professionals, "Purpose-built requirements management tools dramatically increase the efficiency of proper requirements management practices."

QUESTION

What's on the list of software tools your team will use or would like to use in 2008? (Mark all that apply)



SURVEY RESULTS

"Requirements collaboration & management" tops the wish list of tools.

Why do you need specialized tools – can't you just manage everything in documents? It's a common question, and one often asked by senior management when presented with a budget request to buy a specialized tool.

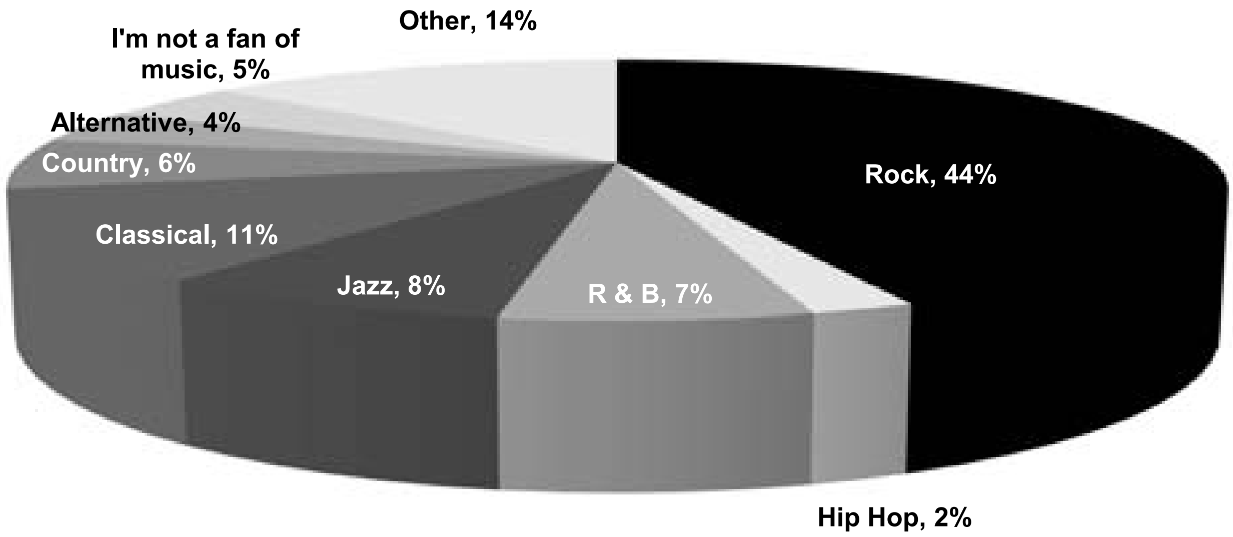
As Forrester Research [3] defines it, "The purpose of requirements management tools is to maximize the likelihood that a development initiative will deliver applications that function as desired."

This survey shows that two thirds of organizations are interested in using requirements collaboration and management software in 2008. What's on your list?

> **"Tools improve the efficiency of mature requirements management practices."**
>
> – The Forrester Wave ™: Requirements Management, 2008 [3]

JUST FOR FUN

What is your favorite genre of music?



SURVEY RESULTS

Work hard. Play music. A shared mantra for product development.

You know what they say, “All work and no play, makes product development a dull job”, or something like that.

We admit, there’s no real business value to this question being a part of this report other than to remind us that despite the challenges and never-ending demands of product development, this profession still rocks. Would you rather be a lawyer? Forget about it.

There are definitely times when managing requirements can feel like a thankless and unglamorous gig. But, as this report illustrates requirements management plays an important role in the bigger picture of being able to successfully develop products on time, on budget and within scope. And, a little music to keep us sane during the process never hurts, right?

> **“I like a bit of everything. Who can choose only one style?”**
>
> – Survey participant

CONCLUSION

2008 is the year of requirements collaboration and management.

The goal of this survey was to identify what product development teams are really doing this year to be more successful and to hopefully cut through some of the hype and buzz that's out there.

So, what have we learned from this survey? Here's a summary of the findings:

1) **There's no substitute for fundamentals.** In order to innovate successfully, you must manage requirements successfully. The top 3 challenges to innovation were: gaining a clear understanding of customer needs, documenting all the requirements and ensuring what's being built is what was planned.
2) **Customer-driven product development.** R&D studies and visionary executives are helpful, but your customers hold the keys to your next product ideas and requirements. The Web has ushered in faster and more efficient ways to elicit feedback from customers to help you build the products they really want.
3) **Customer satisfaction rules.** Revenue? Buzz? Time to market? Which success metric is most important to product development teams? Customer satisfaction is #1.
4) **Beware of scope creep.** Scope creep tops the list for the #1 cause to projects that fail. Followed closely by missed or poorly defined requirements and unrealistic schedules and expectations.
5) **Demystifying "collaboration".** A popular buzzword, collaboration means different things to different people. As it applies to requirements management, it embodies three things: everyone on the team has access to the requirements, everyone is in sync on the latest version and all requirements, related artifacts and discussion threads are captured in a secure and centralized place.
6) **When it comes to process, we're not purists.** There's a lot of media attention around Agile processes, but few organizations have shifted to being a pure Agile shop – in fact only 6% of those surveyed. Most organizations are using a mix of processes, so tools must be flexibile and adapt to your processes.
7) **Documents still dominate, but RM tools top the wish list.** Not surprisingly, over 80% of professionals manually use MS Office to capture and communicate requirements using documents and spreadsheets. However, when asked which tools they plan to use or would like to use this year, requirements collaboration and management tools top the list.

REFERENCES

1) "How companies approach innovation: A McKinsey Global Survey". The McKinsey Quarterly, October 2007. **www.mckinseyquarterly.com**
2) Barry Jaruzelski and Kevin Dehoff. "The Customer Connection: The Global Innovation 1000". Booz Allen Hamilton, December 2007. **www.boozallen.com**
3) Carey Schwaber and Mary Gerush. "The Forrester Wave™: Requirements Management, Q2 2008". Forrester Research, May 2008. **www.forrester.com**
4) Jeff Lash. "Product management vs. Project management". How To Be A Good Product Manager Blog, September 2007. **www.goodproductmanager.com**
5) Robert N. Charette. "Why Software Projects Fail". IEEE Spectrum, September 2005. **www.spectrum.ieee.org**

ABOUT JAMA SOFTWARE

Jama Software is a team of experienced project management, product development professionals who believe in taking a collaborative approach to requirements management. Jama's mission is to build professional-grade, Web-based applications that help companies ensure their product development projects succeed – delivered on time, on budget and meet customer needs. Its product, Jama Contour, is the leading Web-based requirements management application and is used by enterprise teams at companies worldwide including Intel, Volvo, Wiley, Smart Technologies, Fluid, Optimedica and others.

503.922.1058 | info@jamasoftware.com | www.jamasoftware.com

ABOUT THE AUTHORS

Eric Winquist
In March 2006, Eric created Jama Software with the vision of providing customers a more collaborative way to develop software products and eliminate the common frustrations with traditional approaches to requirements management. Eric is an accomplished entrepreneur and project manager with over 14 years experience working with a wide range of enterprise organizations, teams and technologies. In 2001, Eric founded Redside Solutions, a software development consulting firm. At Redside, Eric saw first-hand the need companies had for a more collaborative approach to requirements management which led to the creation of Jama Software and the development of Contour. Eric is a graduate of Oregon State University and lives with his wife and two kids in Portland.

John Simpson
John is a co-founder of Jama Software and is responsible for capturing the needs of its customers. He leads the product marketing efforts and customer communications for Jama. Prior to Jama, John was Director of Marketing at Omniture and brings 12 years of experience developing brands and managing customer-focused marketing programs for web technology companies such as Microsoft, WebTrends and ZAAZ. He has contributed to several books, whitepapers and speaking engagements. As an ambassador for Susan G. Komen for the Cure and a delegate for the Lance Armstrong Foundation, John is an active fundraiser for cancer research. He is a graduate of Oregon State University and lives with his wife and three kids in Portland.

Let us know your thoughts.

Did this report confirm what you already knew? Did some of the findings surprise you? What other things would you like to see in future surveys? Let us know, we're interested in your feedback. Email us:

Eric Winquist, **ewinquist@jamasoftware.com**
John Simpson, **jsimpson@jamasoftware.com**

Collaborative Quality:
One company's recipe for software success

Alan Ark
QA Manager
Compli
arkie@compli.com
http://www.linkedin.com/in/arkie

Author Bio:

Alan Ark is currently the QA Manager at Compli, in Portland, Oregon. Alan has gained tremendous experience working for Unircu (now Kronos – Talent Management Division), Switchboard.com, and Thomson Financial – First Call. While at Thomson, Mr. Ark presented "Euro: An Automated Solution to Currency Conversion" at Quality Week '99 (http://www.soft.com/QualWeek/QW99/) detailing how an automated test suite written in perl was used to save the company great embarrassment. Currently, Alan is teaching his staff how using Ruby can help speed up the testing process. Alan holds both a BS and MS in Electrical Engineering from Northeastern University in Boston, Massachusetts. Interests outside of work include playing golf and listening to Jimmy Buffett songs.

Abstract:

Building software is not an easy process. There are many obstacles that can get in the way of delivering a great product in a timely manner. The intent of this paper is to share one company's experiences with getting better software out the door. This paper does not give you the silver bullet, but hopefully will present some ideas that you can take away for your shop. We don't formally subscribe to any one methodology, but we endeavor to use Agile processes, and are tweaking our process as we learn from each subsequent release that we work on.

While we've had some great results, the focus of this paper is on the little things that gets us those results. The ideas in the paper are presented in more of a cookbook style where the final results are more about the preparation and ingredients, rather than sticking to any pre-determined step by step process. The execution of the process is an important step, but without good ingredients, sometimes you get less than stellar results. We are always adapting to lessons learned with each release, but our base recipe is fully cooked. Think of this presentation as family favorite recipe that you can add your own flair to.

Background:

Compli was founded in 1999 to create the market's first comprehensive web-based compliance management system. In late 2004, a consultant was brought in to evaluate the technology used by the company and assess the product as it was. As a direct result of the analysis, a whole new engineering team was brought in to take Compli to the next level. What that meant was that the current engineering team inherited outdated .ASP code that was not scalable, not maintainable, but according to the previous CTO, contained **ONLY** one bug. Previously, releases to production usually entailed an all-hands-on-deck approach for handling customer calls for the next several days. There were no formal software processes being followed. It was a QA nightmare. There were no acceptance tests defined. There were no test plans in existence. There was no real documentation in existence besides an outdated user manual. This was life in 2006. Let the fun begin.

Base Ingredients:

The base ingredients are just the most basic things used at Compli to build software. It is very likely that you'll need to use these very same ingredients as well.

People

When it comes down to it, a business is only as strong as its people. Each company will have different needs and wants from the people that work within its walls, but without the people, the business is nothing. Teams come in different sizes and shapes, but there are commonalities across different companies. Common job functions, common responsibilities, yet each team is distinct in what it wants to achieve, each company in what product to deliver. People can be found just about anywhere, but it is upon each team to know what kind of person that they are looking for.

Trust

Trust can be defined as being a person on whom one relies. If you have trust between team members, it will be easier to work together. Without trust, it will be nearly impossible to make progress. Unfortunately, trust is not something that can be bought from a store. You must grow this on your own.

How do you earn trust?

Respect

Respect is esteem for or a sense of the worth or excellence of a person, a personal quality or ability. Like trust, one cannot buy respect at the store. It is another item that you must grow on your own. Luckily, if you find trust, then respect is usually lying right next to it.

How do you earn respect?

Communications

Communications is a generic term used to describe how one thing interacts with other things. In the context of communications as an ingredient, it benefits everyone to find the crispest communication possible. Each communication that you make or create is a direct reflection on you.

Focus on what you are trying to convey.
What's the goal that you are trying to accomplish?
A cluttered message usually leads to confusion.

Infrastructure

Infrastructure is one base ingredient that you can buy in a store. These items are more examples of things that are usually common across companies in different fields, yet how infrastructure is implemented will vary widely based on a number of individual factors – each which is unique to your specific company. Some of the major pieces in use at Compli include the following.

Development Environments
QA Test Clients
Project Documentation Repository
Code repository
Bug Repository
Continuous Integration (CI) system
Automated Testing

Execution:

With the base ingredients in place, these are the steps that we take to start baking our software projects.

Access

Even if you had the best, most expensive, magical tools that solved all of your problems, they would be useless unless you could access them. Some of the items that we have access to at Compli include.

- Engineering tools
- People
- Project Status
    - Development Status
    - QA Status
    - Bug Status
    - External Teams Status
    - Progress against the schedule.

Nurture Communication

Communication must be nurtured to ensure that the right message is being sent. It must be used correctly or the risk for misunderstanding – a miscommunication – will increase. We talked briefly about what makes for strong communication. Some examples of how we facilitate communication between team members are listed below.

- We sit in the same area.
- Single repository for core information about the project – salesforce.com
- Minimize having important communication go out thru only email
- Wiki and Blogs used to supplement
- Let people know ahead of time what is going on.

Story collaboration

At Compli, we take the following general steps to develop stories.

- Discuss general themes to implement.
- Use Portfolio Management techniques to decide what to execute on.
- Create and review stories
- Estimates on the body of work
- Establish acceptance criteria on stories

One topic that might be new for readers of this paper is Portfolio Management. This was introduced to Compli by Mark Lawler. Portfolio management can help articulate what the real priorities are. It takes into account the business needs, perceived risks and product expectations and presents the information in a visual format that brings out the most important things to tackle first. Visual examples of the portfolio management output graphs can be found as Figures 1 and 2. For a copy of an Excel spreadsheet where you can try portfolio management for yourself, please email me at arkie@compli.com. Thanks to Mark Lawler who approved distribution of his spreadsheet.

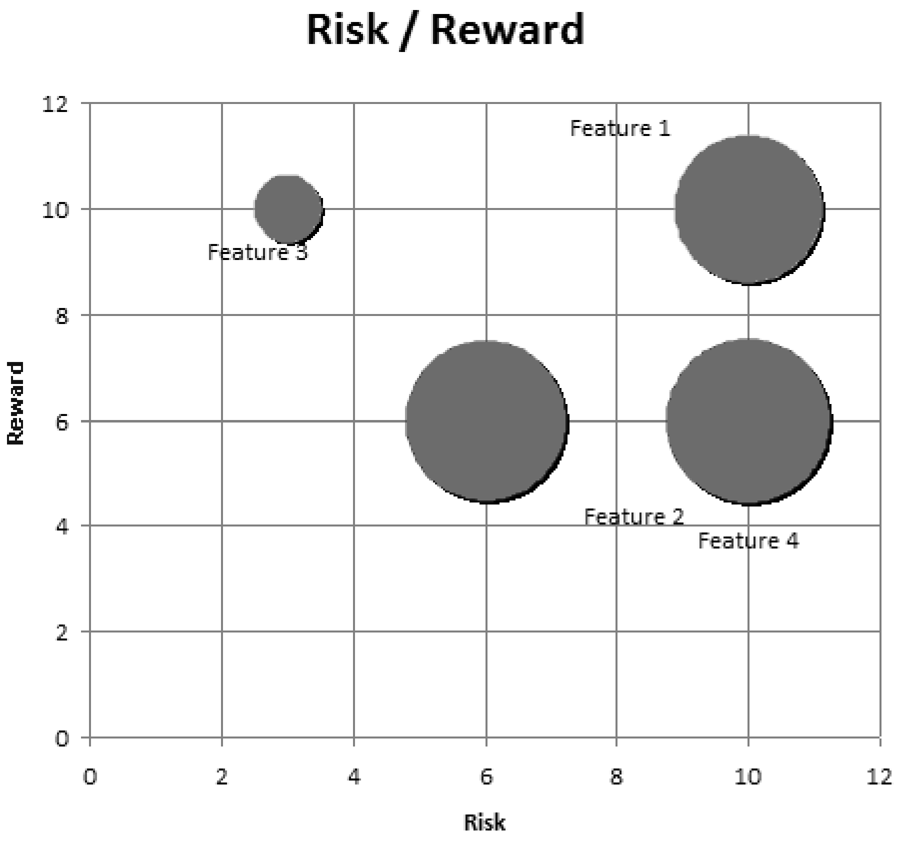


Figure 1 – Risk/Reward graph from the portfolio management strategy

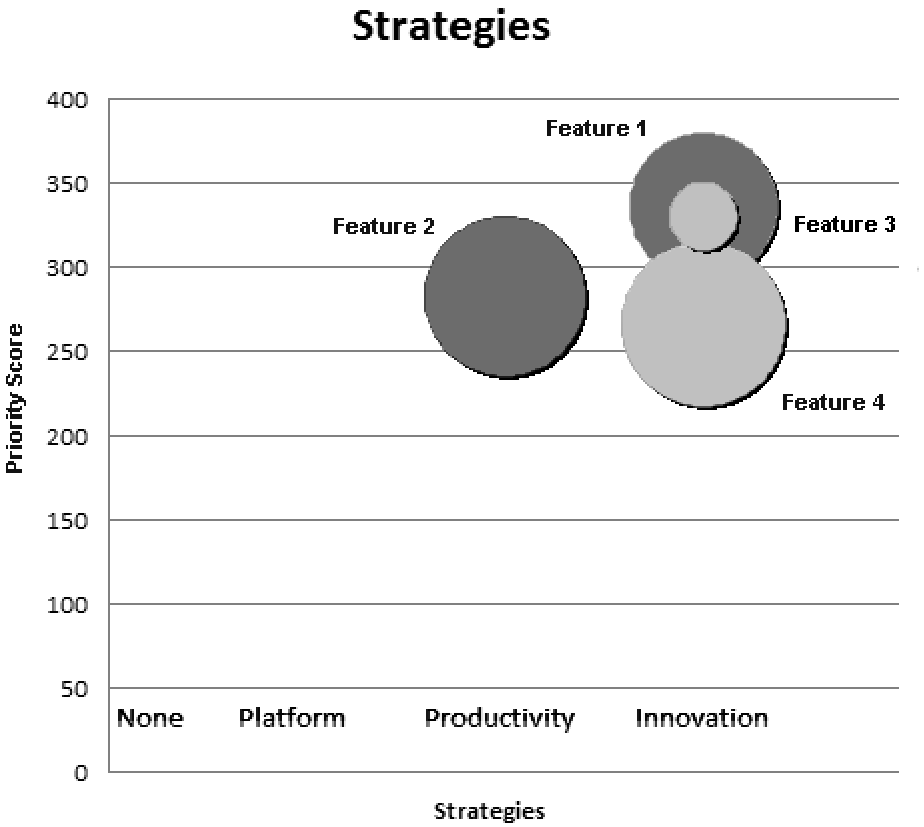


Figure 2 – Strategy/Priority graph from the portfolio management strategy

Metrics

Metrics allow us to measure our progress on the project. What matters is that we tried a bunch of things that we thought might prove useful. The ones that we kept are discussed here. Figure 3 shows a bug trend report that displays all accumulated bugs against a project over time. We have also tried other metrics that we've since discarded because they didn't prove as useful as much as we had hoped.

Metrics we kept include:

Work estimate times
Work actual times.
Bug Statuses
Bug Trends

Some of the discarded metrics included the following.

Bugs by Origin
Bugs per story
Bugs per feature

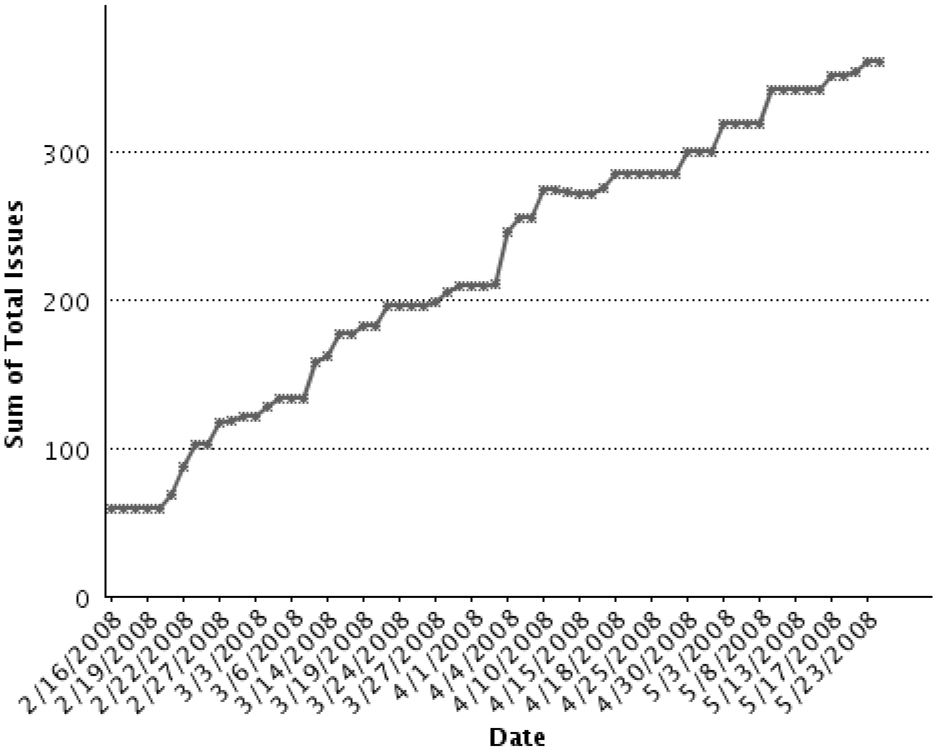


Figure 3 – Bugs accumulated on a project over time

Iterations

Our work slices are called iterations. Each iteration provides a timebox where we try to accomplish specific development/testing goals. Our iterations generally work as described below.

- Continuous Integration of code.
- nUnit tests executed.
- 2-3 week iterations depending on the list of stories to be delivered.
- Daily morning stand-up
- Story demos to the entire product development team
- Coordinated drops into the QA environment at the end of each iteration
- Watir based acceptance tests
- Quick turn around on any bugs blocking the testing of stories.
- Bug Triages as needed.
- Scheduled demos with people outside of engineering to gather feedback on ideas
- Comparison on page load response times.
- Iterations at the tail end of the project also include the following features
  - Usually run 1 week in length
  - Normally bug fixes as opposed to implementation of new features.
  - Practice of what would happen on a release night.
  - Full deployment to QA
    - Full DB migration
    - Web side rollout (web and middleware code)
  - Watir based regression tests
  - Manual regression tests
  - General Availability to fellow employees

Release to production

While the class of hardware in the QA environment is different than what is used on production, each production machine has a QA counterpart. Each rollout into QA can be considered a practice run of a rollout to production. As the project becomes more mature, we execute on full rollouts of database migrations and all build code on the QA hardware. We have a high confidence level that the behavior on the QA systems is very similar as the production system. This makes the actual release to production go smoothly.

- DB migration
- Web side rollout
- Watir based acceptance tests
- Manual verification of "hot-spots" identified in testing of iterations.

Results:

Overall, the results have been outstanding, especially when compared to the experiences of the company pre 2006.

Project duration

- 1 month – 9 months depending on the severity/complexity of the work involved

Major release rollout average 2 hours once ready for QA verification.

Tried this on 10 major releases and 30 hotfix releases in the past 2+ years.

- Only 4 hotfixes in the past 10 months

We have not had a rollback of any release.

The day after a release is a non-event for the company.

- Happy customers
- Happy bosses
- Proud employees

Conclusions:

By using a mix of project management and software development techniques, learning from others at conferences such as PNSQC, and experimentation at home, we have a living process that has helped us deliver high quality software to our customers. We view processes as guidelines to follow, but not rigid rules that must be followed to the letter. Take away bits from others to see what would work within your company. Keep those things that are working, and see what can be improved on those items that are not working so well. Figure out what your specific needs are, and use these ideas to address them.

Don't get boxed into a process

Be flexible. If a change is working, incorporate it in your main process.

If things didn't work out as well as hoped

- How would you mitigate those issues in the future?
- Try things out
- Don't be afraid
- Recognize when things are going well or not so well

If things didn't work, what else could you try to make it work better in the future?

Further reading:

Cooper, Robert G, Scott J. Edgett and Elko J. Kleinschmidt. Portfolio Management for New Products. Cambridge, MA: Perseus Publishing, 2001.

NUnit. http://www.nunit.org/

Watir. http://wtr.rubyforge.org/

Ruby. http://www.ruby-lang.org/en/

CruiseControl. http://cruisecontrol.sourceforge.net/

Salesforce. http://www.salesforce.com/

Thanks to Launi Mead and Jonathan Morris for their review of this paper.

TAILS IN THE BOARDROOM

CANINE LESSONS FOR BUSINESS TEAMS

Shannon MacFarlane
Quality Specialist
500 E. Alexander Ave.
Tacoma, WA
(253) 238-8542
shannon.macfarlane@totemocean.com

Biography:

Shannon is the Quality Specialist at Totem Ocean Trailer Express, Inc., in Tacoma, Washington, and is responsible for the health and wellness of the quality management system's document control, internal audit, corrective and preventive action, best practice program, and lots of other important and misunderstood stuff. She is currently serves as a Mentor on the Panel of Examiner and Process Development with the Washington State Quality Award after three years of service as a Senior Lead Examiner and is a first-year Baldrige Examiner. When she's not at work managing teams she manages her pack of great Danes.

Abstract:

Dogs know the secrets to productivity and harmony within their packs; humans are often blind to these simple truths. Pack theory and canine behaviorism can be successfully applied to manage and improve human relationships. This study is an examination of how common canine behaviors within packs (e.g., conflict management, leadership, and communication) are equally effective when employed by members of business teams. Canine and human examples from current literature and observation illustrate the potential for improving efficiency, productivity, and collaboration among teams without raising hackles.

INTRODUCTION

Dogs know the secrets to productivity and harmony within their packs; humans are often blind to these simple truths. Pack theory and canine behaviorism can be successfully applied to manage and improve human relationships. This study is an examination of how common canine behaviors within packs of conflict management, leadership, and communication are equally effective when employed by members of business teams. Canine and human examples from current literature and observation illustrate the potential for improving efficiency, productivity, and collaboration among teams without raising hackles.

Most wild dogs live in social groups comprised of several families called clans. The social hierarchy is based on the alpha pair, who provide leadership for the group and hold the highest social status.

Within each clan may be one or more packs, which are usually groups of five to seven dogs who hunt together. While the primary relationship between pack members is functional (i.e., to hunt and kill prey for the clan), they enhance their relationships with play. Through play wild dogs and wolves learn about their packmates as they mimic the behavioral sequence that occurs during a hunt. This experience increases the bond between pack members and enables them to better predict each other's behavior, an essential skill when coordinating efforts to take down big prey with other dogs who may be out of sight. As Rogers and Kaplan (2003) proposed, this ability to confidently predict behavior may be a requirement for pack members before setting out together on a hunt.

Canine groups have strict rules of communication that reflect their social hierarchy and each member is responsible for obeying and reinforcing the rules at all times. This constant application creates a highly cohesive group. There are no occasions when pack members can walk past another without some form of interaction, whether it is a full investigation or a casual greeting. Wild dogs work diligently on their relationships and on maintaining the cohesion of the group. It is not surprising that long-lasting groups such as those of wolves invest considerable time in group etiquette.

Dogs and wolves have strong instincts for conflict solving, communication, and cooperation and are therefore convenient models of the same behaviors for work teams.

Table 1 - Canine Principles Applied to Work Teams

	CANINE PRINCIPLE	HUMAN EXAMPLE
Communication	Master body language	Pay attention to signals beyond words when others speak
	Give undivided attention	Respect the other party enough to *really* listen when asked
	Offer and receive honest feedback	Provide feedback to help others progress and graciously and professionally accept the same
	Define boundaries	Establish and explain the policies and procedures that govern the work
Leadership	Focus on progress, not form	Be willing to accept progress even if it isn't picture perfect
	Capitalize on natural talent	Recognize that individuals have unique talents and skills and put those to work
	Enforce rules	Impose consequences fairly and as necessary to protect the agreed upon boundaries and build trust
	Demonstrate commitment	Live the policy statement
	Share knowledge	Tear down business silos and share information to increase the strength of the team
Conflict Management	Apologize well	Sincerely and willingly admit fault
	Practice forgiveness	Support those who make the effort to apologize by accepting the apology and move forward
	Bite as a last resort	Respond to threats calmly, methodically, and rationally – the consequence must be appropriate to the action
	Use calming signals	Understand and recognize distress, frustration, and anger in others and work towards appeasement before negotiating the situation
	Part with emotion	Realize that emotion and logic aren't the best combination and rely more on reason when making business decisions

COMMUNICATION

Master the Art of Body Language

Dogs rely on body language, vocalization, and odor to communicate messages to other dogs. This vocabulary is universally understood by both domestic and wild dogs across the globe. Humans are not equipped to send detailed messages via scent, but we can, and do, readily communicate with one another without saying a word. Young (2007) referred to this communication as "micromessaging." Micromessages are nuanced behaviors people blindly use and react to in dealings with others. Dogs are masters at micromessaging and most canine behaviorists attribute the dog's success in cohabiting with humans to his careful and continuous study of human body language.

The dhole is an African wild dog known as the "whistling hunter." They typically hunt prey in the bush and cannot see other pack members, so they have a network of whistles they use to communicate with each other while surrounding their prey and coordinating the strike (Alderton, 1998). Wolves, who hunt large prey in packs, have well developed, long tails in comparison to other wild dogs. These tails enable wolves to turn rapidly at high speeds but also help to signal to other pack members during the hunt (Rogers and Kaplan, 2003).

Albert Mehrabian began experimenting with what he termed "silent messages" in the 1970s and discovered that although consciously unaware of the signals, people were able to make accurate inferences about a person's feelings based on physical cues. Mehrabian's (1971) studies evidenced that human communication about personal feelings and attitudes consists of messages from three sources, known as the 3 Vs (verbal, vocal, and visual). Of the three sources, body language is the most powerful communicator (55%), followed by tone of voice (38%) and the message itself (7%). The words people speak, therefore, have the least effect on an audience – people rely on how the message is delivered more than the content of the message to understand what is meant.

To hunt together in a pack, each dog must be comfortable with the others and able to accurately predict their behavior. They accomplish this by bonding outside of their functional relationships through play. When coworkers establish a close relationship, they come to understand each other's micromessages better than other peers due in part to the amount of exposure gained through time, but also because they are sincerely interested in the emotional state of the other – they become "tuned in." (Young, 2007)

Give Undivided Attention

Domestic dogs are terrific listeners in spite of the fact they are not native speakers of the human tongue. A dog who is listening will offer every ounce of focus he can muster, at least until he spots a squirrel. If he is especially interested, he will tilt his head to one side. Whether he understands the message is another matter, but we can be certain that our best friends will happily set aside their priorities to be present and listen without judging.

Nodding, eye contact (without staring), leaning forward, and slightly tilting the head are all signals to the speaker that what he is saying is being heard and processed (Pease & Pease, 2004). None of these signals is effective, however, if the speaker is interrupted – allow him time to finish his thoughts before contributing. Effective groups allow each member dedicated time to speak.

Give and Receive Honest Feedback

Dogs, both domestic and wild, accept criticism as a tool for improvement. If a wolf does not engage in the customary greeting rituals when a pack member passes, she will be immediately corrected for the transgression. Likewise, if she witnesses behavior that violates the pack's rules, she is expected to provide feedback to the offending party (Steinhart, 1995). This constant exchange of feedback allows for members to freely communicate to each other regarding how well expectations are met without inciting violence. The pack relies on this kind of communication to maintain cooperation, which ultimately defines its health and longevity.

The ground rules for a forming group should include the commitments to give and receive feedback. Members who do not live up to the group's expectations will be confronted and it is the responsibility of each individual to see that undesirable behavior is immediately addressed. This approach depends on trust in and respect for fellow group members. To let small problems fester is to drive a wedge in the productivity and cooperation of a group.

Define Boundaries

Odor defines canine boundaries, which dogs indicate through "marking territory" with feces, urine, or paw scratches. The scent glands on each paw pad and below the tail ensure each dog has a distinct calling card. These odiferous boundaries politely communicate to other clans of dogs that "this space is taken" and consequences are readily dealt to those who choose to encroach (Rogers & Kaplan, 2003).

Office doors and cubical walls serve a similar purpose in the business environment. Employees are most comfortable in an environment where they have some claim to physical space (Evans, Johansson, & Carrere, 1994). People also wish to protect their emotional space, that is, they want others to respect their personal boundaries regarding what they define as offensive. These boundaries provide a safe haven, whether the threat is physical or emotional. It is necessary to define expectations, or boundaries, when working with others so all parties understand where they can tread without raising hackles.

LEADERSHIP

Concentrate on Progress, Not Form

A dog who recognizes the smell of bacon left unattended on the kitchen counter has one goal: eat the bacon. If reaching the bacon requires breaking several dishes, bloodying a paw on the rubble, and stepping on the cat for extra height, so be it. The important thing for her is to achieve the goal, regardless of how the bacon gets in her tummy. It's hard to fight instinct.

A leader who demands style in addition to success may reduce the group's ability to achieve by limiting their options for action. Had the bacon-stealing dog been concerned with minimizing the damage to the other items on the counter, she may have missed her opportunity for bacon because it would have taken her longer to focus on her form. When solving problems or planning to achieve goals, employees need a bit of freedom to comfortably consider all options. Imposing a requirement of method, while sometimes necessary, may constrict the group's ability to deliver. Additionally, if a group exceeds their defined goals but are chastised for how they got there, they will be less motivated to achieve the target and more motivated to look good while falling short of the goal.

Capitalize on Natural Talent

The biggest and strongest dog is not necessarily the one who leads the hunt. Dogs in alpha positions are not only physically capable but (at the risk of anthropomorphizing) have qualities people associate with ideal, natural leaders. As Meech suggested (Steinhart, 2005), alpha wolves are those who are best able to bring harmony to the pack, not those who are the meanest and most aggressive. The natural leader assumes responsibility for the pack and the remaining pack members accept responsibilities in their areas of strength. The fastest runners serve as the perimeter for encircled prey; the strongest take turns inflicting wounds.

Responsibilities within a group should go to those who can most readily perform them. Taking a talent inventory during the forming stage or at the onset of a new group project allows the group to capitalize on the talents its individual members possess that might otherwise go unnoticed. Not only will the group learn more about its members, but the inventory will help place assignments in the right hands. To ask the linear thinking accountant to design eye-catching flyers for the upcoming holiday party is to invite frustration – she will be frustrated with her lack of ideas and the group will be disappointed with her output. The lively office manager who eagerly decorates for holidays and birthdays will likely plow through the flyer design with ease. The accountant may be better employed creating a database of the employees and spouses who plan to attend, along with their dinner choices. Recognize the talents within the group and make use of them.

Enforce the Rules

The leader of any group must provide the behavioral example to be followed, and dog groups are no exception to this rule. The alpha pair are responsible not only for demonstrating appropriate behavior, but also for being the primary enforcers of the pack rules. As mentioned in *Give and Receive Honest Feedback*, however, each member carries the responsibility of enforcing the rules, regardless of station.

Group leaders must accept the responsibility of being the model of a contributing, effective member, and with that comes the unpopular position of policing behavior. In a well-formed group that has set clear expectations and ground rules, this effort should be minimal, especially if teammates subscribed to the commitment to provide and receive honest feedback. Other standard ground rules include bans on hogging (talking too much), bogging (discussing an issue that's already been addressed), fogging (avoiding, ignoring, ambiguity, or defensiveness), frogging (hopping between subjects without finishing), flogging (personal attacks), and mad dogging (interrupting or otherwise preventing people from speaking) (Lynch & Werner, 1992).

Demonstrate Commitment to the Cause

Wild dogs depend on their social groups for food, protection, shelter, socialization, and help in caring for young and old family members. Wolves reaffirm their commitment to the clan each time they engage in social behavior – it is required that all members routinely evidence their commitment to the group and its members through adherence to the rules. Each member is committed to the cause because the cause is survival; without this commitment the relationships within the group would eventually deteriorate, leaving each dog to himself.

Forming a committee is hardly a matter of life and death but requires the same care and commitment from its members in order to be successful. The members need to understand why they were brought together and what they need to accomplish. A team charter can help to define the group's mission, products and/or services, and essential processes. The group's leader must see that every member of the team understands the purpose and direction of the group.

Share Knowledge

In order to ensure long-term stability and survival of the pack, wild dogs act on succession plans. Puppies are reared with the utmost patience and are allowed ample opportunity to make mistakes. They have the benefit of mentoring from the elders and ready access to the pack's leaders. When they are ready to apply their knowledge they participate in their first hunt. Those who have knowledge share it freely with those who are learning.

When silo keepers participate in group work, they are reluctant to impart information specific to their work areas. This silent barrier naturally impedes the progress of the group because not only are the members operating at different levels of knowledge, but the persistent silos are at odds with the team effort. Leaders who freely share knowledge set the example and expectation that others within the group must do the same, whether through incorporating best practices or communicating relevant data.

CONFLICT MANAGEMENT

Learn to Apologize

The canine language of appeasement and supplication is extensive. A dog who makes a mistake may avoid eye contact, flatten her ears, flick her tongue, lower her head, or lie down. These behaviors seek to make her visually smaller and puppy-like so she can communicate that she understands her error and has no intention of causing additional trouble (Aloff, 2005). If she did not offer these behaviors she would be immediately corrected in a manner appropriate to the severity of the situation. When the apology is complete, the dogs carry on normal activity.

Humans, especially those in leadership positions, often resist apologies because they feel they need to maintain a strong presence by denying fault (Dimitrius & Mazzarella, 1998). Unfortunately this refusal to admit wrongdoing serves to weaken their image in the eyes of others. *The One Minute Apology* by Kenneth Blanchard offers several compelling reasons to apologize and outlines a simple program for making amends while building integrity. The ability to acknowledge mistakes, especially those that affect others, is an essential component of teamwork.

Practice Forgiveness

If a group is going to thrive in the long-term, for every apology there must be forgiveness. When a dog offers an apology to another dog, they exchange signals that indicate the message was received, understood, and accepted in the form of posture and lack of eye contact (Aloff, 2005). The incident is not necessarily forgotten but is certainly forgiven, allowing both to move forward without a second thought. As with apologies, the failure to forgive in a pack conflicts with the rule of harmony and will be corrected appropriately.

Forgiveness is tied to *Live in the Present* – in order to forgive and put the past behind it is necessary to live in the present. Team members who get hung up on who botched the lunch order first need to reevaluate the priorities of the group (it's likely that lunch will not be high on the list) and then commit to achieving the group's defined goals. Group in-fighting depletes the amount of time and energy directed toward productivity. When a apology is offered, it must be accepted in good faith and forgiven for the health of the group.

Bite Only as a Last Resort

When a dog is irritated with another dog's behavior and asks for it to cease, he will first snarl, lifting his lip to expose a few teeth. The other dog continues to invade his space, so the frustrated dog escalates his warning with a wider snarl and begins growling. The inconsiderate dog still does not acknowledge the message, leaving the warning bite as the last defense. The defensive dog bites in the direction of the aggressor and may make contact, which communicates the threat that will be realized without retreat. Should the other dog continue to encroach on the first dog's boundaries, a physical fight will ensue. Fights are typically bloodless but may become more serious if the issue is especially sensitive (e.g., a mother protecting her puppies from a canine predator is more likely to inflict and suffer injury than an older dog guarding his bed from a pushy youngster). Considering that the canine mouth is built to break bones and tear flesh, bloodless encounters are a remarkable demonstration of restraint.

Groups where members are at odds with each other are prone to ardent disagreements over small details. A heated argument about using Arial versus Times New Roman in the group charter is unnecessary biting and should be downgraded to a small snarl – the disagreement does not advance the goals of the group and should be treated with attention relative to its importance. Appointing a peacekeeper for group interactions can help teammates identify when the snarl-growl-bite sequence happens too quickly and redirect the focus to productivity.

It is important to consider the perceived consequences of escalated warnings. If a group member repeatedly snarls at a recurrent problem and fails to escalate to a growl, his snarls will become meaningless and his coworkers will take advantage of his unwillingness to enforce his boundaries. In order for people to respect the sequence of escalation, there must be an eventual consequence if the problem persists. This practice may be difficult for exceptionally shy group members and teammates should abide by their commitment to enforce the rules to assist the shy person in defending his boundaries, if necessary.

Use Calming Signals

Calming signals are the canine toolbox for avoiding and resolving conflict. Rugaas (2006) illustrated and explained about 30 known calming signals, such as sitting down, freezing, yawning, head turning, and tail wagging. Offering one or a cluster of these behaviors is the direct and polite way for a dog to express her uncertainty, discomfort, or nervousness or to reassure a dog who may be feeling uneasy in her presence. These signals are subtle but well understood and are usually mirrored to show that the message was received.

Appeasement behaviors in humans are similar to canine calming signals and include widening eyes, exposing palms, and moving slowly (Pease and Pease, 2004). Widening the eyes to expose the whites makes the eyes appear larger and more infantile, which elicits caring behavior. Showing the palms of the hands and moving slowly demonstrate to the opposition that the hands are free of weapons and there is no intention to sneak a physical attack, both measures that were important signals when people dressed to anticipate combat at a moment's notice. When a coworker is on his way through the snarl-growl-bite sequence, trying experimenting with calming signals to mitigate the threat he feels. Sit down, speak in a level tone, and use small gestures (like open palms) when speaking. As the aggressor relaxes and feels less physically intimidated, he will calm himself.

Part with Emotion

By and large, dogs get along with one another. When an socially ignorant dog takes liberties with a "get to know you" sniff or approaches directly rather than curving his path slightly, he is briefly corrected, apologizes, and is granted the opportunity to start with a clean slate. This ritual occurs matter-of-factly and mature dogs are able to completely avoid or quickly resolve conflict without escalation. Each follows the unwritten rules of canine etiquette and is not emotionally tied to the situation by fear or anxiety because of his confidence in the system of communication. Parting with emotion is closely tied to *Learn to Apologize* and *Practice Forgiveness* – apologizing and forgiving are much easier if emotion can become secondary to reason.

Histrionics in the office are distracting, unproductive, and the hallmark of attention-seekers. When dogs seek to eliminate unwanted behavior that does not directly violate a dog rule (and hence warrant a correction), they ignore the behavior (Pryor, 1999). Puppies that repeatedly jump and climb on an older dog to incite play will eventually give up and move on when they realize they are not getting the desired reaction. This technique is quite effective in extinguishing behavior, but in the case of a group meeting, a respectful correction may be required when the antics violate a ground rule or otherwise interfere with the group's progress. To maximize the value of the correction, it is important that it be appropriate to the offense and not emotionally charged.

WORK LIKE A DOG

Celebrate

The rapidly wiggling, wagging, licking, and barking frenzy that greets weary workers at the end of a day is an elaborate greeting that celebrates the return of a pack member to the pack. The length of the departure does not matter to the dog, who celebrates the return after a three-minute absence with the same fervor as a three-week absence. As far as the dog is concerned, the celebration is the same.

In recent years an email forward has made the rounds through cyberspace that describes a day in the life of an average dog: "Breakfast! Oh boy! My favorite thing! A walk! Goody! My favorite thing! Time for a nap! My favorite thing!" Not only do dogs celebrate the events in their lives, however small, they do not skimp on joy. A puppy who masters a command in her first obedience class wiggles in celebration when her person coos "Good girl!"; she does not postpone her joy until she passes the end-of-class evaluation.

A project manager whose team has just completed the first of four phases of a software implementation may postpone his joy and celebration to avoid later disappointment or jinxing the project. He and his team therefore miss a valuable opportunity to celebrate their success and build the motivation and momentum to achieve the targets for the remaining three phases of implementation.

Live in the Present

If the bacon-snatching dog from *Concentrate on Progress, Not Form* is scolded for her crime, she will apologize and attempt to appease. It is impossible to say with certainty, but she will probably not dwell on the incident. While dogs do have memory of the past and the ability to anticipate the future, they appear to leave the past behind them and plan to deal with tomorrow when tomorrow comes.

In business it is necessary to take into account past events as well as make plans for the future, and within a group this happens multiply because of the varied experience and perspectives each member brings to the process. With so many considerations it is easy to become distracted by last week or next year, but now is the time when the work gets done. Getting stuck in "we've always done it that way" or procrastinating a vital action item until the next progress meeting stand in the way of progress. Today is for making history, not reliving it, and if it is worth doing, it is worth doing today.

Recognize that Even Buttholes Have Good Information

As mentioned in *Define Boundaries*, dogs use anal scent glands to communicate messages to other dogs. Upon meeting new dogs or reuniting with old friends, dogs slowly circle each other and sniff under the tails. This canine handshake and small talk exchanges information about age, sex, and health, and for familiar dogs serves as verification of identity. Sniffing is the socially acceptable, polite way to greet another dog. An approaching dog has a social responsibility to acknowledge and greet each dog he comes across and learns things not obvious from appearances. Because dogs rely so heavily on their olfactory sense for gathering information, this nosey practice typically extends to other species, including people (crotch-sniffing) and cats (butt-sniffing). They seek valuable information and do not appear to be picky about the source.

The butt-sniffing behavioral sequence serves as a lesson (not a literal one, of course) to avoid judgment before reviewing the facts but also that unexpected sources can provide useful information. Dogs are constantly receptive to information acquired through sniffing and treat new situations as opportunities to learn. Members of a team may make unfounded judgments about one another based on office gossip or general appearance and effectively close themselves off from learning from each other. Everyone has unique experiences and recognizing the worth those differences can increase the collective wisdom of the group. Teammates need to be open to the potential of each member to contribute to the team's success rather than automatically discounting those who do not appear to fit in.

Ask for Help

Requests for help from dogs are most often observed in the canine-human relationship, presumably because people have opposable thumbs. The "help me" gaze happens frequently when a favorite toy rolls beneath the sofa, just out of paw's reach. After staring at the toy, attempting to cram her nose under the sofa, frantically checking other sides of the sofa to see if the toy has appeared elsewhere, and returning to crouch in front of the sofa to make sure the toy hasn't moved, the dog will put on her best wide eyes and perhaps whimper a bit while she lifts a paw. This cluster of behaviors is a clear cry for help and she does her best to appear puppyish to incite caring behavior from another (human) pack member. In this case the caring behavior is retrieval of the slobbery, squeaky fleece toy, which is received with celebration and great appreciation. The dog was quick to recognize her limitation in this situation and asked for assistance, which she rewarded with sincere gratitude. It does not appear that dogs keep track of who owes whom a favor or the number of times a fellow canine asks for a helping paw.

In stark contrast, people are reluctant to admit they need help, even when it is painfully obvious, to avoid earning a reputation as incompetent or weak. Employees tend to keep score of how often others ask for a hand in completing tasks. They expect eventual reciprocity for services rendered and keep a mental tally of "help debts" owed to them. These grudging behaviors virtually eliminate altruism and divide groups by compromising trust and respect. Team members are individuals and will therefore have different skills and challenges; as a team the other members should eagerly provide support without judgment for those who need it for the benefit of the team. Moreover, those who ask for help must also express their appreciation.

Learn About Teammates

The role of play in developing and maintaining relationships between packmates is integral in the pack's functional success (i.e., their ability to hunt large prey). All dogs mimic hunting through play, from the great Dane who springs from side to side to disorient imaginary wild boars to the Papillion who pounces on substitute mice. In the canine world play provides the practice necessary to improve productivity and efficiency in work, and much of this is done through teamwork. As mentioned in the introduction, dogs play together in part to learn about each other's reactions to different situations. Play increases the bond between pack members and tighter bonds decrease dysfunction within the group (Rogers & Kaplan, 2003).

Starting meetings with a short play period may help the group develop an identity and build trust and respect for each other. Members can share stories about their hobbies or the group can participate in brief team building games. The camaraderie that comes with recognizing teammates as distinct individuals will provide greater cohesion within the group as well as the opportunity to have a bit of fun.

Shake Off Stress

Canines have the innate ability to recognize when the stress becomes too much them and they then take the time to shake it off. They stop what they are doing and physically shake off their stress from head to tail (Scholz & von Reinhardt, 2007). This motion provides a physical break as well as a mental one – it signals to other dogs that "I've recognized my limitations and understand I can't do much else here. It's time for me to move on."

People are reluctant to signal their peers that they need to take a break – it's commonly seen as a weakness and self-indulgent, especially in the midst of a looming deadline. Taking time to recover from stress is essential for continued productivity and health. Getting away from the desk for a brief walk, a crossword puzzle, or even staying in the office to stretch can significantly reduce stress and provide a boost of energy. There is no shame in recognizing limits, and groups who push themselves through manic four-hour planning sessions also need to schedule time to regularly shake off their stress.

CAVEATS

While dogs do have a knack for teamwork, they also exhibit behavior humans should avoid at all costs. When dogs enforce rules, for example, they provide verbal (growling and barking) and physical (bristling and snarling) warnings before striking. The notion of defining boundaries and cautioning those who threaten to cross them is worthy of emulation, as is dealing consequences when the boundaries are disregarded. Strict adherence to the canine method in this instance, however, may cause injury and lead to incarceration – a physical response to a team problem is simply not appropriate or acceptable.

A dog's livelihood, whether domestic or wild, depends on his ability to get along with others, which includes respecting authority, managing conflict, and communicating effectively. For animals that live in groups and rely on others for safety and productivity, these skills are essential and are selected for by breeding or nature. Although dogs naturally take to teamwork, their abilities to solve complex problems through logic and innovation are inferior to those of humans. As the more cognizant species, it is vital that the ideas presented in this paper be evaluated for fitness of use before application – while the teamwork principles discussed are neither new nor ingenious, they may not be appropriate in all situations.

CONCLUSION

The practice of cooperative and productive teamwork is essential in the business environment that makes use of cross functional teams, self-directed teams, and special project committees. While ample references and workshops exist for team building and group facilitation, their analytical presentation style may be difficult for team members to recall, especially when they need to apply the information.

Many of the reasons people include dogs in their lives are due to the behaviors and principles describe here. They listen raptly when we need to talk about the problems we encountered at work in spite of the fact they do not speak the language. They seek harmony in their relationships and are willing to leave the past in the past. They liberally provide feedback and clearly express their esteem (or lack thereof) of others.

Business people who have experience with canines can easily relate to the commonly used pack behaviors for communication, conflict management, and leadership. With slight modifications these behavior patters are transferable to work teams; the highly structured social lives of dogs serve as models of successful teamwork. While the value of timeless group theories, such as Tuckman's (1965) stages of group development, are irreplaceable, the average dog-friendly person may elicit more personal insight to group dynamics by asking "What would my dog do?"

<u>REFERENCES</u>

Alderton, D. (1998). *Foxes, Wolves, and Wild Dogs of the World*. New York: Facts on File.

Aloff, B. (2005). *Canine Body Language: Interpreting the Native Language of the Domestic Dog*. Wenatchee, Washington: Dogwise Publishing.

Dimitrius, J., & Mazzarella, M. (1998). *Reading People: How to Understand People and Predict Their Behavior – Anytime, Anyplace*. New York: Random House.

Evans, G. W., Johansson, G., & Carrere, S. (1994). "Psychosocial Factors and the Physical Environment: Inter-relations in the Workplace." *International Review of Industrial and Organizational Psychology, 9*. 1-29.

Mehrabian, A. (1971). *Silent Messages.* Belmont, California: Wadsworth Publishing Company.

Lynch, R. & Werner, T. (1992). *Continuous Improvement: Teams and Tools*. Littleton, Colorado: Qualteam, Inc.

Pease, A., & Pease, B. (2004). *The Definitive Book of Body Language*. New York: Bantam Books.

Pryor, K. (1999). *Don't Shoot the Dog! The New Art of Teaching and Training*. New York: Bantam Books.

Rogers, L., & Kaplan, G. (2003). *Spirit of the Wild Dog: The World of Wolves, Coyotes, Foxes, Jackals and Dingos*. Crows Nest NSW, Australia: Allen & Unwin.

Rugaas, T. (2006). *On Talking Terms with Dogs: Calming Signals*. Wentachee, Washington: Dogwise Publishing.

Scholz, M., & von Reinhardt, C. (2007). *Stress in Dogs: Learn How Dogs Show Stress and What You Can Do to Help*. Wentachee, Washington: Dogwise Publishing.

Smith, P. (1998). *Rules and Tools for Leaders*. Garden City Park, New York: Avery Publishing Group.

Steinhart, P. (1995). *The Company of Wolves*. New York: Alfred A. Knopf.

Tuckman, B. W. (1965). "Developmental Sequences in Small Groups." *Psychological Bulletin*, *63*, 384-399.

Young, S. (2007). *Micro Messaging: Why Great Leadership is Beyond Words*. New York: McGraw-Hill.

Testing for the User Experience

User Workflow Testing

By Lanette Creamer

lanette.creamer@gmail.com

Quality Lead-Creative Suites Business Unit

Adobe Systems, Inc

Adobe
2008

Abstract

Customers who are delighted with a product or service are a magnificent asset which every company wishes to retain and attract. Why? Because these customers will promote that product to others. While we have many ways to test for code stability, features and functionality, there aren't as many test methodologies that explore reporting on the overall user experience. The way a user perceives quality is based on their personal experience and their ability to complete the tasks they wish to accomplish, not based on how much of the code is working as expected.

In the majority of Quality Engineering groups the mission is to do an excellent job at reporting bugs in assigned areas based on technical expertise and familiarity with that particular code. Code coverage is becoming a more common measurement of testing excellence as is efficiently completing a large number of test cases, or running tests based on models and use cases. While these all can be useful ways to measure quality, if we fail to test the user experience we risk shipping a product or service that causes a negative reaction from the very customers we intend to delight. Testing collaboratively is one way to get a feel for what customers will experience when performing a series of tasks which span more than one area or product.

This presentation discusses the practical approach that we've applied to testing collaboratively for the user experience, which we call Workflow testing. Workflow testing methodology scales from a use case all the way to testing across the Adobe Master Collection for CS3. Adobe has used Workflow testing in a point product since early versions of Adobe InDesign and has been testing features across the Adobe Creative Suites using Workflow exercises since the first version. We'll discuss how to get started with workflow testing and adapt the methodology to suit your needs as well as how to generate excitement and participation from other people to broaden your team's collaboration. Real examples of the results reported, features and interactions covered, as well as some examples of output generated by CS3 testing will be shared.

Having an efficient way to holistically report the user experience is an effective way to improve overall product quality. This is especially true in multi-user scenarios which span many features, different applications, and a myriad of operating systems. Workflow testing is fun, relevant, educational, and can make the rest of your testing better as you take a new big picture perspective back to testing a specific area or component.

Biography

Lanette Creamer has been with Adobe Systems since 2000 testing products such as Adobe InDesign versions 1.5 through CS, Adobe InCopy 1.0, Shared Technologies across applications like XMP, XML, WebDav, and has been working as the Quality Lead for the Adobe Creative Suites Workflow Team since 2006. Lanette studied Graphic Design at Western Washington University, but her true love was for Photoshop, Illustrator, and Pagemaker. After attending an inspiring seminar at the CAST conference in 2007, she started a testing blog at http://blog.testyredhead.com/ hoping to find other people who are passionate about collaborative testing.

Introduction

Workflow testing is a collaborative exploratory test methodology intended to uncover defects which may be missed by testing components and functionality in isolation. It results in an overview of issues a user is likely to encounter when producing output from software as well as information about the user experience when performing a series of tasks to accomplish an end goal. Past data, careful sampling of candidates representative of a larger user base, and informed prediction of how future features will be used can help improve the accuracy and usefulness of workflow testing. Workflow testing is intended to supplement excellent test methodology already in place.

Some believe that coding is a requirement for being in Quality Engineering. We believe that the ability to effectively represent users is a required skill for all testers. Regardless of the tools we use to test, we still are ultimately responsible for reporting problems with the software, and this includes reporting a bad experience before our customers suffer the impact. Workflow testing exercises are a chance to be creative, do exploratory testing, and learn more about users and the products your company creates. When teams are siloed and only see the next deadline, the big picture seems irrelevant. It is worth taking the time to see the overall picture of how your product or area fits in to the work a customer needs to accomplish as one piece of their day.

1.0 What is Workflow Testing?

1.0.1 Planned

A workflow is a set of connected steps to achieve an end result. As long as there is more than one step, technically, most test cases could be referred to as a workflow. In the context of workflow testing, we most often refer to the entire series of steps to generate output of some form from beginning to end. End-to-end workflow testing can give you a broad sense of the functionality, areas of strength and weakness, and user experience of a feature, product, or set of products. While many of us run elaborate scenario tests on our own, to collaboratively test requires some planning. With test resources generally limited, end-to-end user workflow testing which is based on real customer data is used at Adobe to find bugs which may be missed by other forms of testing. The intent of workflow testing is not to be exhaustive or provide maximum depth of code coverage. It is intended to provide broad coverage in the areas which are most vital to users.

1.0.2 Exploratory

By its nature, although planned and based on as much user data as possible, end-to-end user workflow testing is exploratory testing because test cases are being written as they are executed. [2] The workflow is a collaborative time constrained effort. The workflow exercise ends, and wrap up and reporting tasks begin when the allocated time is up even if some members have not completed their tasks. Participants in workflow exercises follow a loosely defined task list and complete the task with their own data in most cases, and often are choosing between multiple paths to accomplish a task. The task completion is tracked so that we can include those features in a future workflow to get the coverage we need.

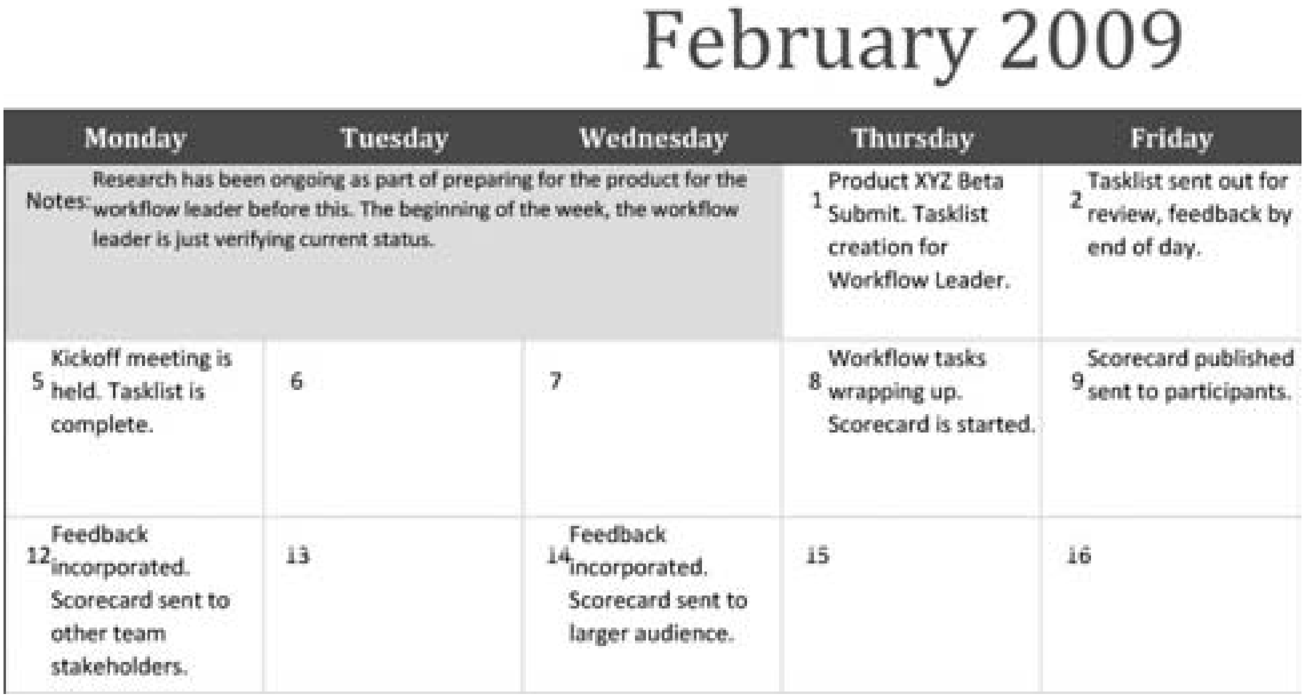
There are some competing priorities while participating in a workflow exercise. The tasklist encourages testing variety and helps the software under test remain the top priority as assigned tasks must be signed off with results when the workflow exercise time is up. There is real output generated, but it is made clear to all participants that the highest priority is important bugs and higher quality for software rather than the output being generated. If we are unable to complete our project due to data loss or performance problems the workflow exercise is still a success in the goal to find serious user workflow bugs. The scorecard will then highlight defects and design decisions which may seem less severe when not considered in a user context. Likewise, the successfully created output can show a tangible example of completed functionality. During CS3, we were able to publish our entire newsletter across products in a pre-beta build. This was a great stability increase from the previous version of the Creative Suite at the same stage in the project.

Some tasks are assigned to everyone, while others are assigned to just a small percentage of the overall representative user group participating in the workflow. Depending on the workflow being represented, the output created may be a scaled down model of what happens in the real world, or it could be true to size.

1.0.3 Documented Collaboration

An end-to-end Suite workflow exercise at Adobe starts with a schedule which takes into account the testability status of the features and products under test so that we can provide timely feedback for teams. Timing the workflow exercise is helpful in providing new and useful information, as well as preventing needless fire drills.

Figure 1. An example workflow schedule



February 2009

Monday	Tuesday	Wednesday	Thursday	Friday
Notes: Research has been ongoing as part of preparing for the product for the workflow leader before this. The beginning of the week, the workflow leader is just verifying current status.			1 Product XYZ Beta Submit. Tasklist creation for Workflow Leader.	2 Tasklist sent out for review, feedback by end of day.
5 Kickoff meeting is held. Tasklist is complete.	6	7	8 Workflow tasks wrapping up. Scorecard is started.	9 Scorecard published sent to participants.
12 Feedback incorporated. Scorecard sent to other team stakeholders.	13	14 Feedback incorporated. Scorecard sent to larger audience.	15	16

Before the workflow kickoff meeting, the workflow leader generates a tasklist which explains the roles, test environments, and products/features to be covered as well as the output to be generated for the project. It does not include any test cases, just tasks for a user to do. The tasklist will be completed at the kickoff meeting with assignments, build information, and bug tracking data for that workflow included. The builds and platforms are locked down as well as the assigned tasks each participant will attempt to cover in the given workflow exercise. Some flexibility is built in and all participants can add tasks at any time to the list, but may not remove tasks as often we rely on deliverables from one tester for the next tester to use while we are working together. We need to be on the same build with stable deliverables to replicate most real world multi-user scenarios. We do include legacy products in some workflows if customer research shows it is commonly used.

Having tried multiple ways to track bugs through a workflow, it has been helpful for us to take a collaborative approach with each product team so that they are not surprised to get extra bugs from the workflow exercise. We invite each team to contribute participants and let them know our plans in advance. Being considerate of the team culture, and bug writing style can enable teams to take over bugs that happen in just one product and verify and close them without help from the workflow exercise participants.

The final piece of documentation generated is the "scorecard". A scorecard is a snapshot of what the represented user segment would experience trying to accomplish the assigned task if they picked up that build (In our case, of the Adobe Creative Suite, in other cases it could be of a product, a component, or even an operating system). The rating system in the diagram is scientific and based only on data.

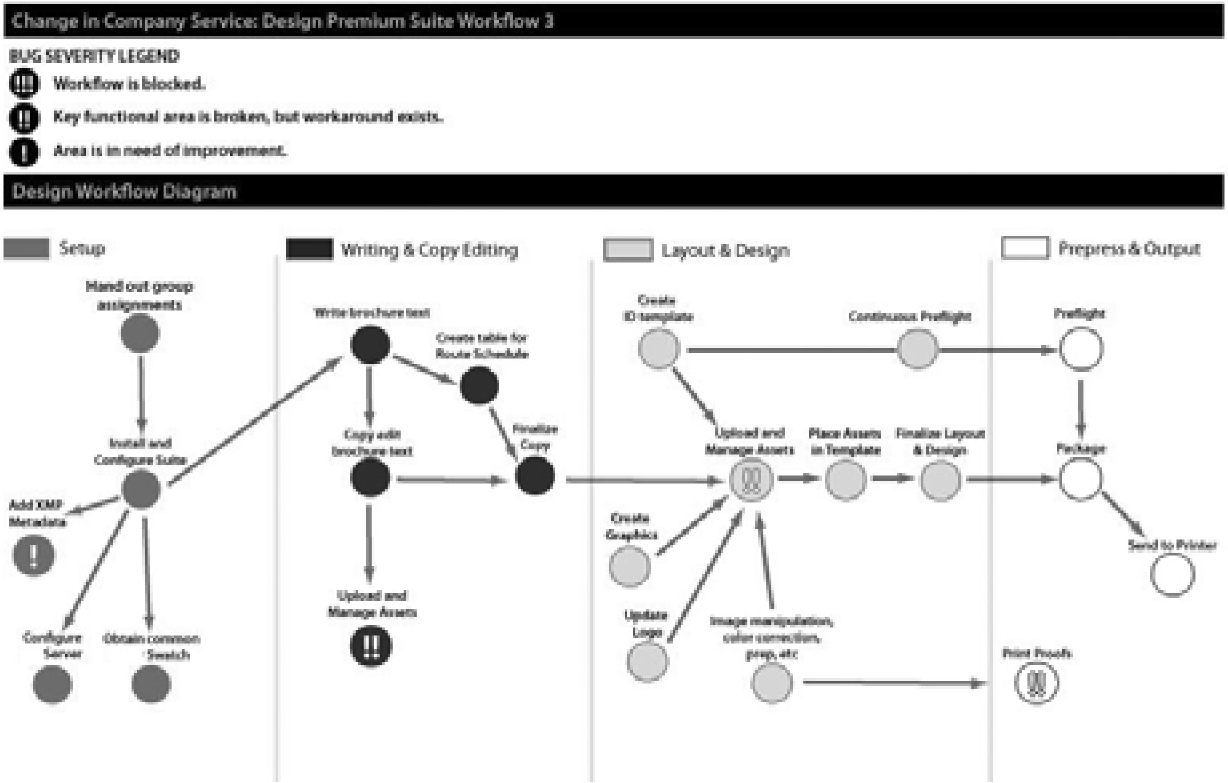
The rating choices are as follows:

!. The area is in need of improvement. This means we found significant problems.

!!. The area had a major problem, but a workaround exists. This means we had to work around missing functionality, a crash, or data loss. In cases it means we worked around an entire product not functioning.

!!!. The workflow was blocked. This means we could not continue with any workarounds. If this status is showing up late in the project, because it is based on real customer data and bugs found, it is taken very seriously by everyone interested in quality, including project managers and upper management. To avoid false alarms, we communicate with product and component teams who are called out in the scorecard before we ever publish them and we also have a column for notes indicating if the issue is fixed in an upcoming build, and any status updates, as well as who to contact regarding each serious issue.

Figure 2. An example workflow scorecard diagram



The subjective data included is generally a short paragraph about the experience and is generally backed up by usability bugs. We keep the subjective data as scientific as possible by including concrete evidence, such as percentage faster/slower or time which could be gained by this customer segment per day for performance bugs. Customer segment impact for usability problems, and percentage of users likely to be impacted based on sales data if a problem is specific to one test environment, platform, or application can also be helpful information to help convey the severity of a bug. Because the scorecard is a collaborative effort, agreement on the subjective data between multiple testers is an important aspect of giving credibility to usability bugs.

1.0.4 Real World Results

A workflow exercise should result in content that the tester cares about that makes sense for a user to generate. Even if the end-to-end workflow you are testing is for a developer tool, it should result in a real application which is debugged, compiled, and runs with a purpose far beyond “Hello World!” Whenever possible it should be an application that you can personally use to do something.

Working on something real helps you experience the user pain and delight. When you are very close to your deadline and hit the error that makes you miss it, the error becomes more significant; however, when you had a redraw problem that did not seriously impact your work, it may seem less important than it did when you were running tests in a different context.

Using content with a purpose helps you find throughput errors which could be missed otherwise. For example, when your test photo came out a bit too blue in printing, it was easy to overlook. Now your boss looks like a blueberry in the company newsletter, and that printing problem has more impact. You can easily imagine the reaction of a Sunday Newspaper with elected officials showing up with blue skin. It saves the company money and the end user pain if we find these bugs as early as possible. [1]

Figure 3. Bugs

Here are few examples of the types of Suite level bugs found and fixed during CS3 workflow test exercises.

a. Bugs in one product cause issues in multiple products.

   Example: Product _________ crashes when opening from Application A, Application B, and Application D.

b. Bugs occur when files from one application are imported into another. With the variety possible with workflow testing, it is more likely to encounter a backwards compatibility issue as some users move forward, while some stay on the old Suite to represent the user who has not yet upgraded.

   Example: Product E crashes when importing legacy files from Application A.

c. Bugs that result from using the products in a heterogeneous multi-user environment are found when different operating systems are used and multiple users are accessing the same files.

   Example: Versioned publication does not update to new version when a remote user steals the lock.

2.0 How is the Testing Done at Adobe?

The leader of the workflow exercise should be experienced in the products used and familiar with the existing data in the areas being tested. Workflow participants should be clear how much time they have to dedicate to the project and the commitment from their manager that they can see the Workflow Exercise to completion of their role, including providing scorecard data. The most successful workflows that have happened at Adobe involved tight collaboration, a strong leader, a well researched task list, and passionate participation by all members. The members share the common goal of creating an accurate scorecard with realistic output that is a model of work being published by our customers, or that we anticipate they may make in the future given improved or new features that we are developing.

2.1 Role of Workflow Leader

2.1.1 Research

Before a tasklist is started, the leader of the workflow exercise uses past data as guidance. The leader will look for any information on customers currently working in our product. This could be in the form of studies, user surveys, and customer visits. Who is this group of users? What is important to them? What are they creating? Who are the stakeholders they are responsible to?

2.1.2 Predictive Workflows

It is more difficult with new features, but as testers we often have use cases as well as past data and can piece together logical combinations of how new features may best fit into an existing workflow. Predicting entirely new workflows which could be generated is worth the risk of testing inaccurately if it results in preventing bad press, disappointed users, or brings to light other factors which may impact the overall success of the product in the market.

When a product ships, it can be helpful to verify which workflows were accurately predicted and compare that with workflows that customers did not use. It can take several releases for new features to "catch on" enough that it is possible to compare your predicted workflows with the feature in use.

2.1.3 Considering Risk

Some may wonder, with the inaccuracy of predicting workflows, why not run known workflows only? We have customer feedback on existing workflows, but very limited feedback on features not yet released and how they interact with the current user experience.

Finding bugs that interrupt existing workflows due to feature creation or modification can save your company costly mistake and dot releases. [1]

Example of risks found and addressed during CS3 workflow exercises ranged from loss of an entire server directory by introducing corrupt file information from one user that would download to multiple user accounts causing eventual server failure and loss of ability to restore, to performance issues which happened only when using multiple products with new shared technologies.

2.1.4 Publish Tasklist

Once the initial tasklist is created, it should be shared with all participants in the workflow exercise. The workflow leader creates the tasklist, but all participants have input.

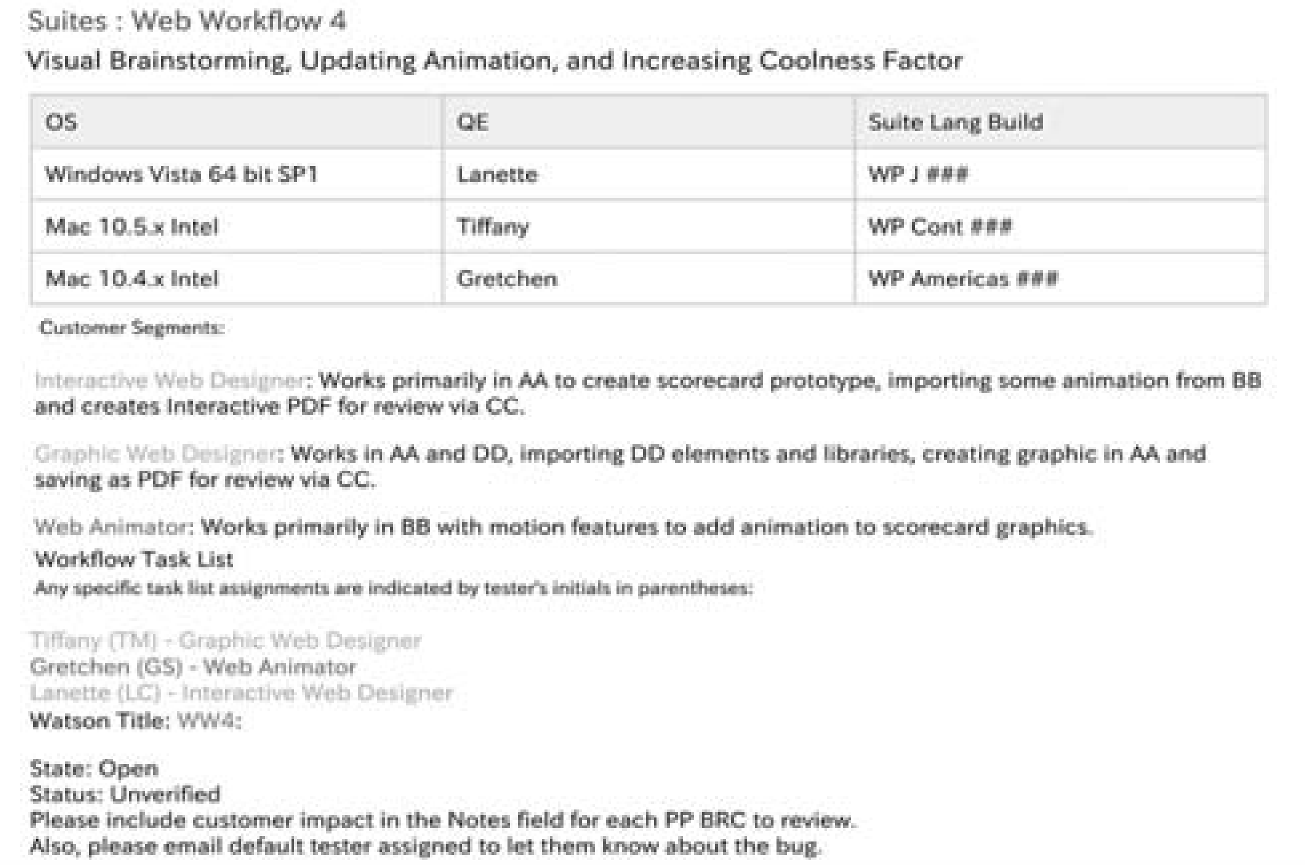
2.1.5 Collaborate

When you receive feedback from participants and stakeholders, incorporate it with the goal of the overall workflow in mind. While the process is collaborative, the workflow leader has the final say in what remains in the tasklist. After all, they have the most knowledge of the customer need and past usage in this area.

2.1.6 Hold Kickoff Meeting

The kickoff meeting is where the common vision of the task, full schedule, and roles are assigned. The workflow leader will schedule and hold this meeting. In most cases this takes about 30 minutes, but it could take longer at first. By the end of the kickoff meeting, everyone knows their role in the workflow, what they should deliver, and what build to install and which platform. The tasklist is then published in its final form for everyone to refer to as needed.

Figure 4. A tasklist example from a workflow exercise (Continued on the next page)



Suites : Web Workflow 4

Visual Brainstorming, Updating Animation, and Increasing Coolness Factor

OS	QE	Suite Lang Build
Windows Vista 64 bit SP1	Lanette	WP J ###
Mac 10.5.x Intel	Tiffany	WP Cont ###
Mac 10.4.x Intel	Gretchen	WP Americas ###

Customer Segments:

Interactive Web Designer: Works primarily in AA to create scorecard prototype, importing some animation from BB and creates Interactive PDF for review via CC.

Graphic Web Designer: Works in AA and DD, importing DD elements and libraries, creating graphic in AA and saving as PDF for review via CC.

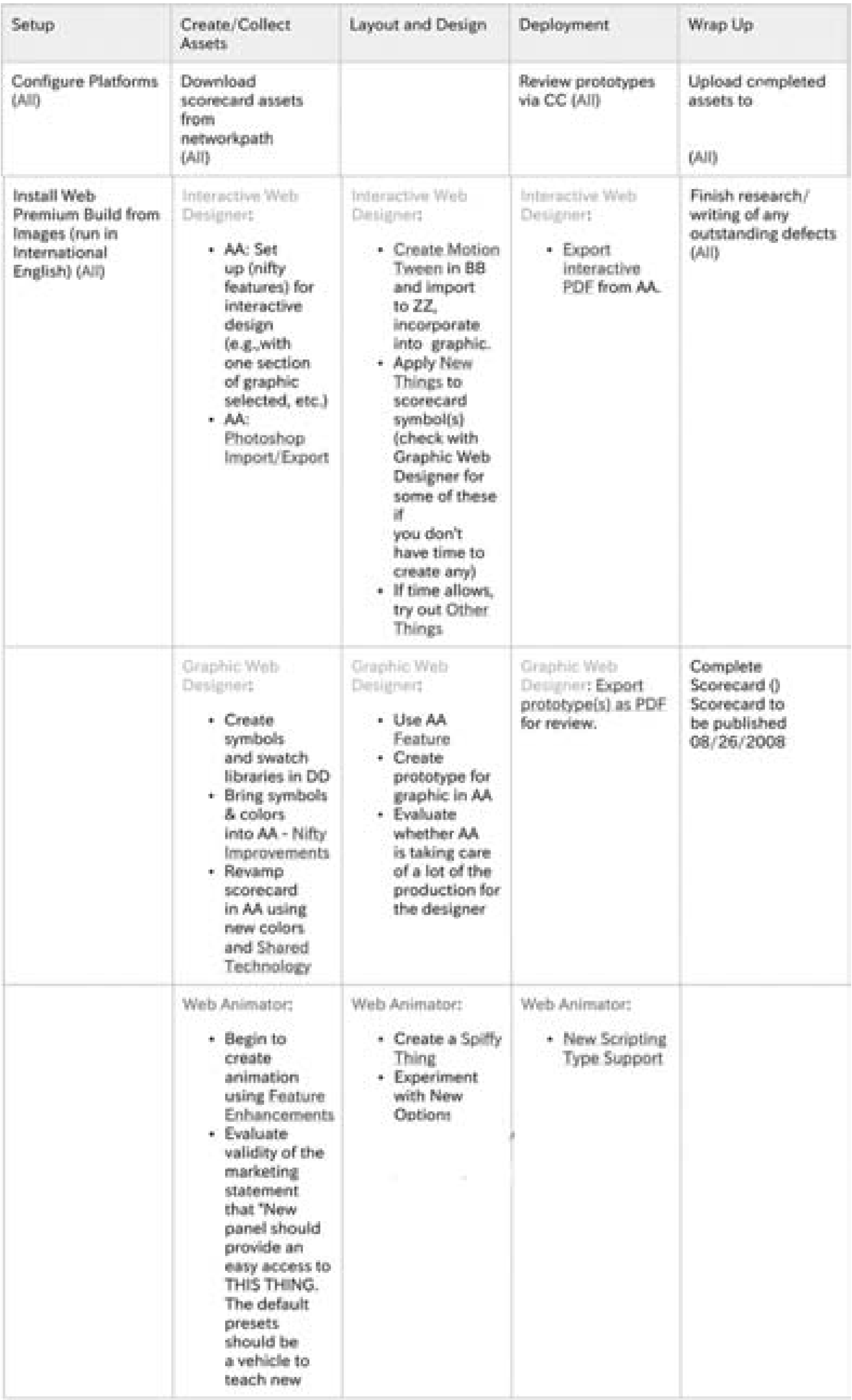
Web Animator: Works primarily in BB with motion features to add animation to scorecard graphics.

Workflow Task List

Any specific task list assignments are indicated by tester's initials in parentheses:

Tiffany (TM) - Graphic Web Designer
Gretchen (GS) - Web Animator
Lanette (LC) - Interactive Web Designer
Watson Title: WW4:

State: Open
Status: Unverified
Please include customer impact in the Notes field for each PP BRC to review.
Also, please email default tester assigned to let them know about the bug.



Setup	Create/Collect Assets	Layout and Design	Deployment	Wrap Up
Configure Platforms (All)	Download scorecard assets from networkpath (All)		Review prototypes via CC (All)	Upload crmpleted assets to (All)
Install Web Premium Build from Images (run in International English) (All)	Interactive Web Designer: • AA: Set up (nifty features) for interactive design (e.g.,with one section of graphic selected, etc.) • AA: Photoshop Import/Export	Interactive Web Designer: • Create Motion Tween in BB and import to ZZ, incorporate into graphic. • Apply New Things to scorecard symbol(s) (check with Graphic Web Designer for some of these if you don't have time to create any) • If time allows, try out Other Things	Interactive Web Designer: • Export interactive PDF from AA.	Finish research/ writing of any outstanding defects (All)
	Graphic Web Designer: • Create symbols and swatch libraries in DD • Bring symbols & colors into AA - Nifty Improvements • Revamp scorecard in AA using new colors and Shared Technology	Graphic Web Designer: • Use AA Feature • Create prototype for graphic in AA • Evaluate whether AA is taking care of a lot of the production for the designer	Graphic Web Designer: Export prototype(s) as PDF for review.	Complete Scorecard () Scorecard to be published 08/26/2008
	Web Animator: • Begin to create animation using Feature Enhancements • Evaluate validity of the marketing statement that "New panel should provide an easy access to THIS THING. The default presets should be a vehicle to teach new	Web Animator: • Create a Spiffy Thing • Experiment with New Option:	Web Animator: • New Scripting Type Support	

2.1.7 Leader Participates

The leader is responsible for creating the scorecard, the tasklist, and doing the upfront research, as well as staying in touch with other teams which may be impacted, but they also act as participants during the workflow.

2.1.8 Help Isolate Bugs

Most of the information needed for participants is included in the tasklist, or is communicated at the kickoff meeting, but if participants need help during the workflow, the workflow leader is the first source of information and help.

2.1.9 Wrap-up

Driving the schedule and checking with participants is one duty of the workflow leader. On the last day of the workflow, part of the day should be dedicated to finalizing all bugs to be entered so that the scorecard can be created. The tasks assigned to each participant will be signed off in the tasklist. Ask for subjective data from the participants to include in your scorecard as far as “how was the experience”. Part of the purpose of having a wrap-up time set aside is so that participants can go over their notes and investigate any areas they felt compelled to explore and perform some exploratory testing to flesh out any remaining bugs. We use this wrap-up time as a way to keep the user first in our mind without undermining the value of exploratory testing. Adobe consistently values exploratory testing, and I believe it shows in the quality and success of the released products. [3]

2.1.10 Creating a Scorecard

The scorecard is an overview that includes a diagram with simple ratings of 0-4 depending on what happened during that part of the workflow as explained in section 1.0.3. The common themes in the subjective data are then compiled to give an overview. Generally bugs are included only if they impact the overall workflow. If it a bug is found in a product while testing that didn’t impact the workflow progress overall, a bug is written to that product team and left out of the scorecard to keep the overview as simple and direct as possible. It is aimed at a large audience, and they tend to not read in depth information considering the limited time.

The workflow scorecard should be equivalent to an “elevator speech” and it should be readable at a glance. The first review for a scorecard is participants. Send the scorecard first draft out for review and ask for feedback on typos, or subjective data taken out of context. Once those changes are incorporated, look for any products or bugs called out in the scorecard, and ask the QE Manager and Project manager for a review. At Adobe, we have a “notes” section and ask them to directly edit the scorecard with any notes they have. Generally the notes specify if the bug is already fixed, or is under investigation to avoid false alarms.

2.1.11 Gaining support

Gaining support for workflow testing can be difficult with blackbox teams already stretched to the maximum and a lack of knowledge on how to use the product from whitebox testers. To gain support for workflow testing, we share the scorecard, and invite people to join us in workflow testing exercises taking their expertise into consideration, but also giving them some tasks which will take them into unfamiliar areas or products for learning opportunities. For example, if it is primarily a whitebox tester, we may provide them designs to modify and then code into an application. If it is primarily a blackbox tester, we may give them a code sample to modify but primarily design tasks.

It is often easier to build a coalition of the willing and avoid including unenthusiastic participants in workflow testing. When we show what bugs were found, and explain how little time it takes to participate and the advantages in learning and networking, some people will respond and some will not. It isn't necessary for everyone to participate to have a very productive workflow exercise. Ultimately, support from quality management is vital. If stakeholders do not allow the time to plan, run, and then consider the results of workflow testing, it will fail at the company.

I suggest running it on a "trial" basis, and then doing a survey. It is important to find out if your workflow testing is spreading product knowledge, testing ideas, and awareness of how changes in other products may impact users of testers assigned to another product.

Was it fun? Did it wake up your brain? Did you leave with a big picture view you hadn't considered before? Did you get a chance to be creative?

2.1.12 Sharing the results

Who is interested in the state of the project? We share our workflow results with all of those who make decisions on the Creative Suites, upper management, engineering, and all of QE. We also publically post scorecards, tasklists, output, and our schedule of upcoming collaborative workflows. All of the scorecards are available and linked to the corresponding task lists online.

It helps to welcome comments and questions on the scorecards. It's also fun to share the output by posting printed items in public areas and sharing the URL of web content created. Whenever possible, we create something that has dual purpose, such as our quality newsletter or a web based testing tool of some sort. This often gives us the chance to explain how it was created and share the idea of workflow testing with others.

2.2 Role of Other Participants

2.2.1 Complete assigned tasks

Tasks assigned to individuals in the scorecard are signed off (initialed) as a sign that they completed them at the end of the workflow as part of the wrap up phase. If a user is unable to complete a task, a bug number is noted in the notes section.

2.2.2 Test with a user mindset

The single most important responsibility of a workflow participant is to think, act, and feel like the user. This is real content with a real deadline. It will not be extended. If we cannot complete it, we failed to output our content. That is real money lost. The user role is in the front of your mind as you test, not just finding bugs. In this way, the testing varies from exploratory testing where your intuition may take you down a different path. You are thinking like a user, not like a tester. When you get that intuition, you take a note, and move on like a user trying to meet a deadline would. Does this result in missed bugs? Possibly, but the purpose of workflow testing is not to find the maximum number of bugs. It is the find the bugs most likely to negatively impact user workflows. Those are different goals. While the process is exploratory in nature, the user and content trump finding the maximum bugs. We are going for high impact, high visibility, difficult to find bugs, not edge cases and forced error conditions. Test creatively and with care for noting bugs, but in workflow testing, the customer and the customer content is the top priority.

2.2.3 Switch to tester mode

When a bug is found, use as much factual data as possible to properly isolate the bug and communicate the user impact. It is important that workflow bugs are properly reported and isolated so that they can be fixed and convey credibility. Words like “feels slow” should be replaced with “Takes 10 times longer than the last version” with the facts to back it up. The goal is a high quality bug report that is clear, actionable, and delivered to the correct person for consideration. In addition to getting bugs fixed, workflow serves the function of alerting technical support when a tech doc may be needed. There may be workflow exercises that happen after code freeze or even GM, but costs can still be mitigated by being prepared with a response and workaround.

2.2.4 Report subjective data to the workflow leader

When the workflow leader asks for your comments, they want to know how it felt as a user to experience your product.

Which parts worked well? What was difficult? Why? What would have made the experience better? This information should be conveyed in a professional and non-accusatory tone. It is helpful to include both good and bad experiences.

3.0 Conclusion

Workflow testing is a fun and educational way to collaborate with testers working on various teams as well as a chance to learn what has been important to users in the past, and what changes happening elsewhere may have impact elsewhere. While not primarily a bug finding activity, it is intended to find bugs which will hamper existing user workflows as well as predicted future workflows. As all of the products under test have already been tested, it is possible that there would be a workflow where no bugs are found during workflow testing. However, despite over 6 years of workflow testing, we have not had even one workflow result in no bugs. In addition to finding bugs, it also provides an opportunity to network and share knowledge across teams. The workflow exercise results in a user focused overview showing working and blocked functionality at a glance, which can be used to improve both feature and product integration. Subjective usability data is also shared across teams before the product ships. Not only can workflow testing be enjoyable, but the big picture thinking can inspire testers to create new test cases and be more open to collaborating with other testers to get coverage between areas and products. Workflow testing is a part of the success Adobe has had in increasing sales and favorable reviews since the Creative Suite first shipped. [3]

4.0 References

[1] Software Defect Reduction Top 10 List By Barry Boehm and Victor R. Basili, January 2001 (Vol. 34, No. 1) pp. 135-137 from http://csdl.computer.org/dl/mags/co/2001/01/r1135.htm.

[2] Definition of Exploratory Testing by James Bach from http://www.satisfice.com/articles/what_is_et.shtml.

[3] http://www.adobe.com/aboutadobe/history/timeline/ October 2003, CS Ships. April 2005, CS2 Ships, March-July 2007, CS3 ships. Each Creative Suite generating more sales than the previous version from CS to CS3 taken from public data including http://news.cnet.com/8301-10784_3-9804379-7.html and http://premium.hoovers.com/subscribe/co/factsheet.xhtml?ID=12518.

Software Testing as a Service (STaaS)

Author: Leo van der Aalst[1]
Solution and Innovation manager at Sogeti Netherlands B.V.

1 Introduction

The importance of IT testing is growing. Some important drivers for this are:

- Higher business demands and expectations on 'first time right' software launches.
- Legislation and regulations (e.g. SOX, SAS70, Basel II act and Clinger Cohen act) put stronger demands on quality assurance and test processes.
- Mergers, chain integrations, globalization and technological developments lead to more complex IT chains.
- Business demands swift, high quality and cost effective IT services that contribute to business processes.

IT becomes a utility. The business departments' demand guarantees from IT services that IT implementations will not threaten business continuity. The business department demands a test process which clearly demonstrates that requirements have been sufficiently met, and that risks for deployment are acceptable. Testing will become a utility also.

Test service providers who can offer the test process as demanded by the business departments will be very successful, especially if these providers always: make the client's objectives highest priority and commit to focusing primarily on the success of the client's business. A robust and successful collaboration with the customer is founded on the skills of the providers' test professionals, highly industrialized test processes, open communication and full transparency regarding objectives, measurable results, responsibility, operation procedures and costs. The model to support this all is called: *Software Testing as a Service*.

2 STaaS definition

Software Testing as a Service (STaaS) is a model of software testing used to test an application as a service provided to customers across the Internet. By eliminating the need to test the application on the customer's own computer with testers on site, STaaS alleviates the customer's burden of installing and maintaining test environments, sourcing and (test) support. Using STaaS can also reduce the costs of testing, through less costly, on-demand pricing.

From the STaaS provider's standpoint, STaaS has the attraction of providing stronger protection of its test approach and establishing an ongoing revenue

[1] Mike Roe, Brian Hansen, Rob Kuijt and Dirkjan Kaper; thank you for your input. You were a great help!

stream. The STaaS provider may test the application on its own server or even use a third-party application service provider. This way, the customer may reduce their investment on server hardware too.

3 Drivers for STaaS adoption

The traditional rationale for test outsourcing is that by applying economies of scale to the testing of applications, a test service provider can test better, cheaper and faster than companies can themselves. STaaS could be the next step in test outsourcing. Several important changes made to the way we work could make a rapid acceptance of STaaS possible:

Everyone has a computer: Most testers have access to a computer and are familiar with conventions from mouse usage to web interfaces. Therefore, the learning curve for new applications is lower, requiring less handholding by the customer.

The testing industry has matured into a standard practice: In the past, executives viewed corporate test centers as strategic investments. Today, people consider testing to be a cost center and, as such, it is suitable for cost reduction and outsourcing. IT is commodity → testing is a commodity!

Testing by companies themselves is expensive: In-source testing activities require expensive overhead including salaries, health care, liability and physical building space.

Standard test approaches are available: With some exceptions, testers can use a standard test approach to test any application. Refer to TMap Next [Koomen, 2006].

A specialized testing provider can target global markets: A testing provider specialized in testing widespread applications (packages) can more easily reach the entire user base.

Security is sufficiently well trusted and transparent: With the broad adoption of SSL, VPN and Citrix, testing providers have a secure way of reaching the applications under test. This still allows the environments to remain isolated from each other.

Wide Area Network's bandwidth has grown drastically: Added to network quality of service improvement, this makes it possible for testing providers to trustfully access remote locations and applications with low latencies and acceptable speeds.

IT as a utility ensures the customer that test environments are no longer scarce and mysterious environments that must be carefully managed. Therefore, test environment capacity can be quickly increased and decreased without upfront investments.

4 STaaS process

As said: Software Testing as a Service (STaaS) is a model of software testing where an application is tested as a service provided to customers across the Internet.

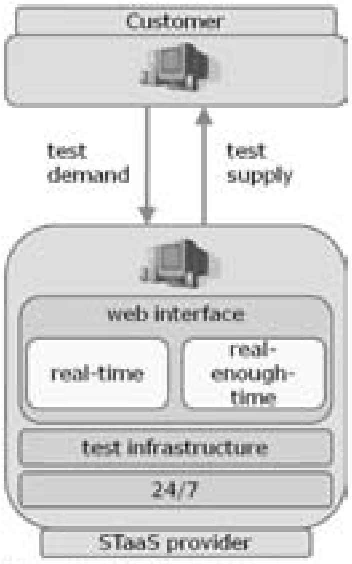


Figure 1: STaaS process.

The customer has a test demand. The demand is sent through the internet to a STaaS provider. After a certain time the STaaS provider sends the customer a test report (figure 1).

What happened in between? How did the provider deal with the test demand? For instance did the provider use a '*real-time STaaS'* or a '*real-enough-time STaaS*'? In addition, how did the provider deal with other challenges like test infrastructure, 24/7 availability and the communication between customer and himself?

Web interface

In a *real-time STaaS* (figure 2) the test demand is implemented without human intervention by the provider. In the ultimate form of the real-time STaaS a test object (e.g. application software), including test bases (e.g. requirements, use cases, set of heuristics), design and architecture model, is offered to the STaaS provider. Without human intervention this is implemented in a test environment. The entire amount of testing is performed by human simulators against the model and a neural network forecasting. A test report is sent to the customer. Is the above-described real-time STaaS science fiction? Yes, for this moment anyway. Perhaps in the future?

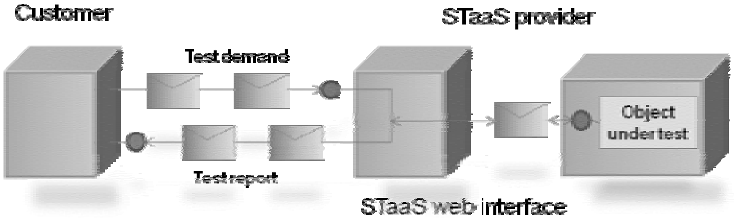


Figure 2: Real-time STaaS.

Examples of today's existing real-time STaaS are:

- Regression subscription to periodically checking the external and internal links on a web site. Are the links for instance still working correctly and not broken?
- Regression subscription for application interfaces in a suite of applications. Monitoring the health and functionality of the application landscape.
- Periodically, from various locations (worldwide), execution of performance measurements of a web site.
- Testing of SaaS applications through STaaS (e.g. web services collecting interest percentages or license plate data).

In a *real-enough-time STaaS* (figure 3) the test demand often requires human intervention in the workflow. The demand is carried out, behind the 'scenes', by many humans, through which it appears as if the test demand is carried out by computers. By its very nature, this introduces a latency and unpredictability to the STaaS process.

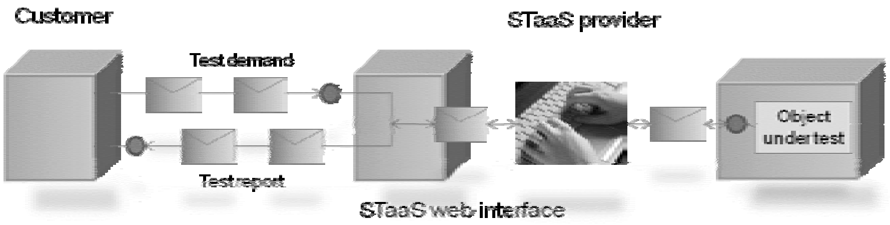


Figure 3: Real-enough-time STaaS.

Examples of existing real-enough-time STaaS:

- Work Package (WoPa) broker. Through a formal test demand mechanism everything about the assignment is specified in a WoPa. This includes items such as; what should be tested, how and when, what criteria should be used, and what knowledge is necessary. The WoPa's are stored in a kind of virtual (digital) cupboard. The WoPa can be pushed by the WoPa broker to, or pulled by, a tester or WoPa team that possesses the sufficient means to carry out the WoPa. The WoPa serves as the contract between customer and provider.
- Managed Testing Services (MTS) is the structured form of a WoPa broker who is specialized in a particular client or application. Through MTS the provider takes full responsibility for test assignments, with clear commitments expressed in KPI's on quality, cost level and time to market. MTS is organized in so-called test lines. A test line is the operational organization to provide test services to one or more customers. A test line has a fixed team of testers, infrastructure, test tools and standardized work procedures. Every test line has a permanent key team of testers that ensure continuity and knowledge retention. There is also a flex team. When the work available in their test line is insufficient, the flex team is assigned to other test lines (temporarily). It is a flexible pool of testers deployed to test lines with the most work pressure.

Test infrastructure

With STaaS it should be possible to test an application from all over the world, regardless of the location of the tester and the customer. This requires special attention to the test infrastructure. Figure 4 “Test infrastructure” below contains an existing and operational infrastructure used by a provider.

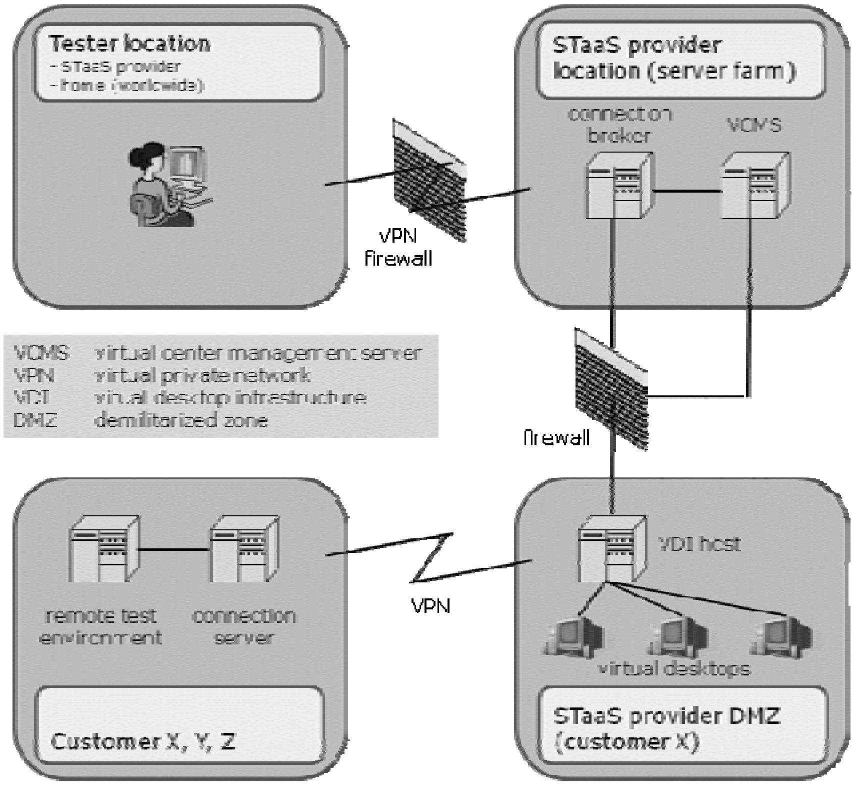


Figure 4: Test infrastructure.

The tester (either from home or provider office) has a remote connection tool on his computer with which he or she can establish a VPN connection to the connection broker of the provider.
Based on a certain classification, the tester is assigned a desktop and the VCMS prepares this desktop ready for use (e.g. VMWare Test lab manager). These virtual workstations are hosted on a server farm at the providers’ location.
In case a connection with the customer is needed, a secure tunnel is set up. E.g. virtual workstations are placed in a DMZ and from the virtual workstation a second VPN connection is established to the customer.
In this way multiple customers can be connected, each with its own virtual desktops. Security is guaranteed in this way.

Other possibilities are:

- hosting of the test infrastructure by the STaaS provider
- outsourcing of the test infrastructure to a third party hosting provider.

24/7

When it is possible to test an application all over the world through the internet, the provider and his testers should be available 24/7. In this situation a test demand is not rejected just because it is night at the location where the provider is situated. The provider needs a broad network of testers spread all over the different time zones or needs testers available 24/7 in a specific time zone. Because the demand for testing services will fluctuate, it is recommended that the provider have a fixed pool of testers and a pool of flex testers (figure 5). In practice students have proven to be very suitable as flex testers; they like to work in virtual environments, are time-independent and location-independent and can be paid per assignment.

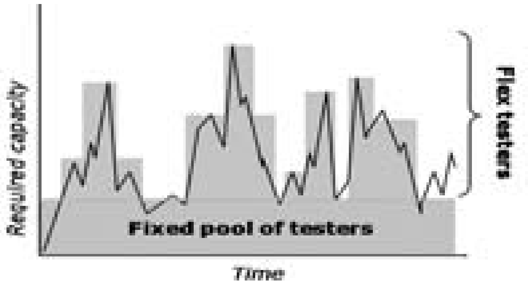


Figure 5: Fixed pool and flex testers.

In addition to the fixed pool the flex testers cover the required flexibility to cope with peaks and variation of required workload. Assignment of the flex testers is based on the planned test capacity demand and agreed reaction times. Learning time of flex testers is relatively short due to the provider's standard working practices. In principle the flex testers leave the fixed pool at the end of the peak.

Of course, using test tools and test automation could also support the STaaS providers' 24/7 availability.

Governance model (test demand ⟷ test supply)

The STaaS provider has to distinguish various interaction points where the customer and provider interact and communicate. Figure 6 "Governance of a test line" gives an overview of the generic governance model used by a STaaS provider.

- Customer contract manager ⟷ Provider delivery manager.
  Agreement on a strategic level regarding contracts and SLA. At this level there is a responsibility for setting up the contracts and SLA.
- Customer manager test services ⟷ Provider test line manager.
  Set up and maintain the standard procedures and KPI's. Initiate and start work packages.
- Customer project manage ⟷ Provider test manager.
  Planning and monitoring progress of test activities. Progress reporting, defect reporting and management. Delivering the conclusive test report after finishing the test.
- Customer development teams ⟷ Provider test coordinator.
  Findings after the testability review are reported. If required testers and

designers meet in a session to clarify the findings. The result is a clear unambiguous test basis.

- Customer development teams ⟷ Provider test coordinator.
  Intake of delivered software. The initial test is performed on the basis of agreed entry criteria. Issues regarding the initial test are reported to project management and development. If required a meeting is set up to clarify issues. The result is a system with sufficient quality to start test execution.
- Customer development teams ⟷ Provider test coordinator / test engineer.
  Retest of resolved defects. Through a delivery document the developer lists which defects are resolved in the new build. This document is the basis of the re-tests.

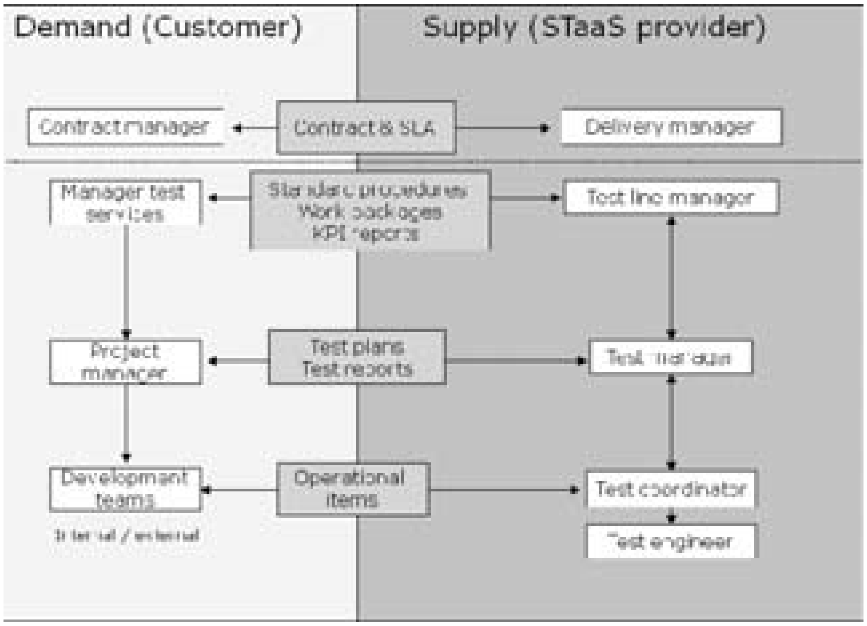


Figure 6: Governance of a test line.

5 STaaS provider services

A service item is a certain element of the test process offered to the customer for which the STaaS provider is responsible. These service items can be highly varied. Moreover, the established service offering can be modified when new services are proposed or existing ones are eliminated. The STaaS provider must deliver a result based on the demand. The delivery must occur within the pre-defined timeframe, at pre-defined costs, and at a pre-defined quality level. The provider is responsible for guaranteeing continuity in delivering the result.

Some typical service items are (in alphabetical order):

- ‘chain integration’ testing
- creating test scripts
- evaluating test basis
- executing tests
- infrastructure testing
- Installation testing
- localization testing
- management of defects
- performance testing
- reporting
- security testing
- setting up and maintaining test data
- setting up, maintaining and hosting test environments
- test automation
- testability review
- testing of standard packages
- testing of web applications.

6 STaaS provider process model

A number of processes have to be set up by the STaaS provider to offer the services. The STaaS provider process model (figure 7) consists of two parallel primary processes:

- The process for the actual execution of the service in an assignment.
- The process that supports and monitors the execution.

The processes serve to support the assigned employees' collaboration needed to accomplish the contracted services. The processes are described in detail below.

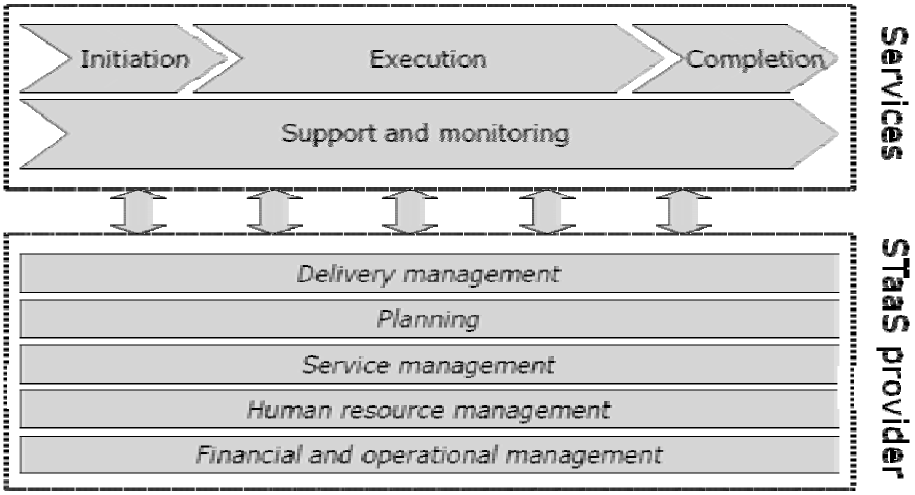


Figure 7: STaaS provider process model.

Initiation

This is the first phase for execution of the assignment. The assignment always comprises one or more services tailored to the customer's specific demand. The initiation phase serves to describe the scope of the assignment accurately. This can be done by creating a so-called *assignment description* and optionally asking the customer to approve it. An assignment description concretely describes:

- the STaaS service asked for, including preconditions and basic assumptions
- clear commitments expressed in KPI's on quality, cost level and time to market
- the agreements on monitoring by the STaaS provider in relation to communication lines, progress reporting and consultation
- the deliverables.

Furthermore, the initiation phase is used to identify what is available in the provider's organization for (re)use on behalf of the assignment. This may include templates, standards, existing test scripts or test design patterns from previous assignments, test environments and tools.

In more detail: Test design pattern

A test design pattern is a generic set up test structure and/or test strategy, which solves a specific common type of test design problem. Test design patterns are generically described, offering the advantage of a recognizable solution pattern, regardless of the implementation details. Using test design patterns accelerates the communication of a test assignment because the solution of a common test design problem has, in fact, been given a "name".

Execution

In this phase, the assignment is executed in conformance with the agreements with the customer as described in the assignment description. Furthermore, the parties communicate via the agreed communication lines on the results, progress, risks and bottlenecks in the execution of the assignment.

Completion

Reuse of resources is one of the success factors of the provider. In this phase, the assignment is assessed and a satisfaction measurement made with the customer. The lessons learned from the assessment are fed back into the provider's organization and incorporated into the (new version of the) service. This results in formal process improvement embedded into the processes of the provider's organization.

Support and monitoring

The provider continuously supports and monitors the assignment process as described above. The progress, risks and bottlenecks involved in the execution of the assignment are monitored. Where and when necessary, the involved parties reach new agreements on the assignment.

Delivery management

This process covers activities that aim to acquire assignments for the provider and manage (long-term) relationships. Examples are maintaining the test environment, repeated testing of releases, and live monitoring of applications.
A contract is created to govern how both parties will handle the assignment. It specifies agreements on the service level provided by the provider.

Planning

The planning process ensures that the right tester is deployed to each assignment. 'Right' in this context means that the knowledge and competencies of the tester match the knowledge and competencies required for the assignment. Other planning aspects are:

- required availability of the tester (during working hours, weekends, 24/7)
- required or available location (office, home, off-shore, near-shore, on-shore)
- technology (bandwidth and processing power availability).

Service management

The range of services provided by the STaaS provider is not set in stone – it may grow or recede. To this end, it must be determined periodically whether the current service offering is in line with the requested services. In addition services must be known (to the customer, assignment management and tester) and the products for the services must be up-to-date and in line with the latest developments.

Human resource management

The process of human resource management aims, among other things, to continuously develop the skills and career of the provider's testers. This requires matters like defined job positions with associated competency, continued training and remuneration levels.

Financial and operational management

Financial management is a continuous process based on budgeting (what are the expected costs and benefits) and monitoring (what are the actual costs and benefits). Operational management can be executed based on many factors. Examples of these factors are:

- the percentage of assignments completed within the agreed key performance indicators (KPI's) on quality, cost level and time to market
- the percentage of test services acquired as compared to test services acquired by competitors.

7 Achieved results by a STaaS(-like) provider

A STaaS managed test services provider with 300 testers, 20 clients and 18 test lines achieved the following results in the first year of its existence:

- proven test cost reduction
- demonstrable improvement of quality of testing, test process, test deliverables, test results and flexibility of test operations.

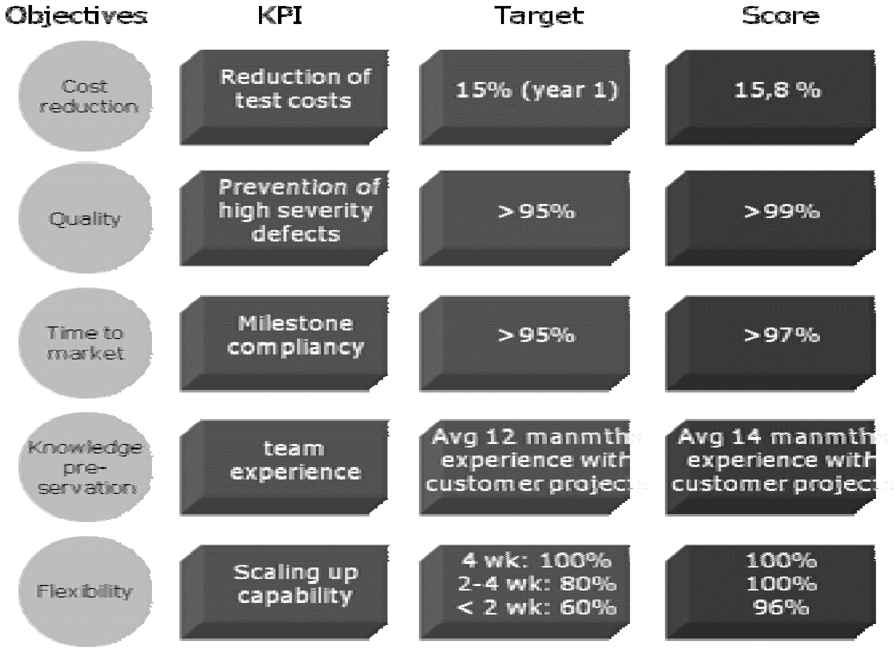
Optimization of costs was expressed in measurable improvements of test costs, measured in agreed units. Units could be test costs per function point, per requirement or could be expressed as a proportion of test costs versus total project costs. After agreeing on the measurement unit the STaaS provider performed a zero-measurement to establish the starting situation.

In the experience of a particular provider, managed testing services have been yielding the following results:

- 10% reduction of test costs per test unit within 6 months
- 15% reduction of test costs per test unit within 12 months
- 25% reduction of test costs per test unit within 24 months (forecast).

The provider committed itself to key performance indicators that were directly related to customers' business objectives.

Figure 8 "STaaS MTS results" shows the business objectives related to agreed ("Target") and achieved ("Score") KPI's.



Objectives	KPI	Target	Score
Cost reduction	Reduction of test costs	15% (year 1)	15,8 %
Quality	Prevention of high severity defects	>95%	>99%
Time to market	Milestone compliancy	>95%	>97%
Knowledge pre-servation	team experience	Avg 12 manmth experience with customer projec	Avg 14 manmth experience with customer projec
Flexibility	Scaling up capability	4 wk: 100% 2-4 wk: 80% < 2 wk: 60%	100% 100% 96%

Figure 8: STaaS MTS results.

The test cost reduction was achieved through the following measures:

Measure	Short explanation	Level of cost reduction
Resource rationalisation	Assign tasks to employees with matching seniority level. Focus on healthy ratio for test management vs. test coordination vs. senior test engineers vs. junior test engineers.	Combined these three measures have led in practice to cost reductions of **5-15%.**
Sufficiently lean core team	The size of the key team is adjusted to the highs and lows of the mid term forecast in such a way that the average level of occupation for the key team is > 95%.	
Alternatives for idle time	Within testing dealing with idle time is a common phenomenon. Idle time for a test team can rise to 20% of the test effort. By using the economies of scale of the test line a very flexible process is set up that allows prompt re-assignment of testers to other projects in the case of idle time. In practice the test line has proven to reduce idle time to a level below 5%.	
Uniform process	Install a uniform test process, with standardized test products and procedures. Maintain a key team to use and re-use the process and test deliverables in multiple	By installing a uniform test process team cost reductions up to **5%** of original test costs have been achieved.

	projects	
Test automation	Proper use of test automating contributes to test cost reduction.	Test automation has resulted in cost reductions up to **5%** of original test costs.
Near shoring and off shoring	Through off shoring and near shoring testing activities are transferred to regions with lower cost rates. In practice the amount of off/near shoring depends on certain conditions and varies from 0% to 70% of all testing activities.	Taking into account the investment cost (translations, remote connections, extra QA and communication effort) test off shoring has resulted in cost reductions varying from **10% to 30%.**

8 Conclusion

Benefits of STaaS

Thanks to survival of the fittest, STaaS is forced to provide its customers the best full test service solution because STaaS providers have to compete with other STaaS providers. Therefore, these providers have to make sure they:

- use the scarce expertise on structured testing, infrastructure and tools optimally
- improve the test processes continuously
- have international professional test capacity available
- industrialize the test services (‘test factory’)
- produce reliable test product quality
- give advance insight into costs and running time.

The STaaS provider takes full responsibility for test assignments, with clear commitments expressed in KPI’s on quality, cost level and time to market. A solution should be available for single applications, full projects, and portfolios.

STaaS leads to cost optimization as well as demonstrable improvement in quality of testing, test process, test deliverables, test results and flexibility of test operations.

Challenges of STaaS

A STaaS provider must devote continuous attention to a number of challenges. Managing these challenges is critical for the provider's long-term success:

The provider must determine on a continuous basis whether the services offered still match the customer’s demand. The customer, not the provider, determines the required quality level.

The professionalism of the provider is based on the knowledge and competency of the testers on the one hand, and the stability of the tester population on the other. If there is a continuous inflow and outflow of people in the ‘pool of testers’, there is no stability and no solid basis for knowledge building.

Often, an important objective defined for the provider is cost savings. One way to achieve this is by archiving test ware, test data, and test infrastructure for reuse.

Continuous attention to optimizing, organizing, and refining test objects as intellectual property is critical.

As a provider, it is important to render an objective assessment of the delivered software or hardware, independently of the customer. On the other hand, the customer may have other interests (less costs, short time-to-market). This is an important challenge that may pose contradictions.

The test basis (e.g. requirements, use cases, design specifications, heuristics) should be available in English or translatable into a language which is understood by the testers. Preferably in a clear and simple form using tools such as QualityCenter Requirements or Rational Requisite Pro.

If the quality of the test basis is inadequate, an alternative way to gain domain knowledge is needed.

And last but not least, test environments should be accessible from various locations.

Reference

- [Koomen, 2006]
  Koomen, T., Aalst, L. van der, Broekman, B., Vroon, M. (2006), TMap Next, for resultdriven testing, ’s-Hertogenbosch: Tutein Nolthenius Publishers, ISBN 90-72194-80-2

Title: Is your Testing Effective and Efficient?

Author: Bhushan B. Gupta
Affiliation: Hewlett-Packard Company, 18110 SE 34th Avenue, Vancouver, WA 98683
Email: Bhushan.gupta@hp.com

Biographical Sketch:

Bhushan Gupta has 23 years of experience in software engineering, 13 of which have been in the software industry. Currently a Program Manager/Test Lead in the RPS group, at Hewlett-Packard, he joined the company as a software quality engineer in 1997. Since then he has led his groups in product development lifecycles, development methodology and execution processes, and software metrics for quality and software productivity.

As a change agent, Bhushan Gupta volunteers his time and energy for organizations that promote software quality. He has been a Vice President, a Program Co-Chair, and a Board Member of the Pacific Northwest Software Quality Conference. He offers a workshop titled “Engineering Software Quality” at the Center for Professional Development, OHSU, for software quality practitioners.

Bhushan Gupta has a MS degree in Computer Science from the New Mexico Institute of Mining and Technology, Socorro, New Mexico, 1985.

Abstract

Adequate testing is much more difficult when your product involves multiple facets such as software, hardware, Other Equipment Manufacturer (OEM) components, and industry compliance including safety and environment. Test coordination is more complex as there are multiple teams engaged in testing the product. It becomes increasingly difficult to ensure that all facets of the product are tested and there is no unintended test effort duplication.

The Retail Photo Solutions (RPS) group at Hewlett-Packard has developed a test planning method termed “Test Landscape” that assures a high level of test effectiveness and efficiency and yields a high quality product. It defines testing scope, identifies test ownership, and tracks test coverage and status across multiple development stages and quality attributes. The method involves identifying the quality attributes, such as Functionality, Usability, Reliability, Installation/Deployment, Safety, and Regulatory, that must be tested. These attributes form the horizontal test vector. To make sure that the product components are adequately tested before they are integrated, a vertical test vector representing development stages including Unit, Module, Component, System, and Solution and Beyond is also established. The two vectors combined yield a test matrix – the Test Landscape. The method is being used by the RPS group and has made test management simpler and efficient, while also enabling management to have more confidence in the testing process.

Introduction - Background and Challenges

The Retail Photo Solutions group at Hewlett-Packard develops solutions that enable the consumer to produce memorabilia such as calendars, posters, and albums from their own photographic work. The solution is comprised of:

- A software component that assembles images into the desired memorabilia format using image input devices (scanners, kiosks, memory cards etc.)
- Printing devices both HP and non-HP
- Production equipment such as CD/DVD for archival.

This is a global business so the solution must be localized and internationalized. Being global, it also has to comply with each country's regulatory requirements. The solution includes the OEM software and hardware components that are subject to the same standards of quality as the in-house components. The software and hardware development is co-located on multiple HP sites in North America and Europe.

This complex nature of the product makes its testing increasingly difficult. In particular the group faces the following challenges:

- Avoiding testing an integrated system before its individual components are sufficiently tested and stable
- Covering both the customer and the international regulatory perspective
- Optimization of overall testing to avoid test duplication
- Conducting the right testing at the right time in the development lifecycle, across the various stages of integration form component to solution.

Relevant Definitions

The following definitions are relevant for the foundation of this work:
Effective Testing: The test plan and its execution assures a minimal high priority defects found in the field
Efficient Testing: There is no unintended duplication of testing efforts

There are situations where some test duplication may be unavoidable. For example, a user interface is included in two different platforms supported by the solution with a slight variation. Both platforms will need testing, resulting in some duplication. We came up with the test landscape concept and used it as the primary method for organizing and communicating our test planning among the multiple involved groups.

Framework for the Test Landscape

The two vectors that define the test landscape are the product quality attributes and the different levels at which these attributes must be tested during product development. The quality attributes are the product characteristics in addition to functionality that a product must posses to provide value to its users. These characteristics include but are not limited to installation, usability, performance and form a sound basis for the product quality. Gupta and Beckman [1] have discussed the prominent software quality attributes. The following table lists the important attributes and their definition from the sources highlighted in the table which the team used for this methodology:

Attribute	Definition	Source
Functionality	The capacity of a solution to provide its required functions under stated conditions for a specified period of time	Webster Dictionary
Usability	The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use	ISO 9241-11
Reliability	The ability of a solution or component to perform its required functions under stated conditions for a specified period of time	IEEE Standard Computer Dictionary, 1990
Installation	The capability of the software product to be installed in a specified environment	http://www.isi.edu/natural-language/mteval/html/222.html
Localization	Means of adapting for non-native environments, especially other nations and cultures	http://en.wikipedia.org/wiki/Internationalization_and_localization
Regulatory	Legal restrictions promulgated by government authority	http://en.wikipedia.org/wiki/Internationalization_and_localization
Security	Condition of being protected against danger or loss	http://en.wikipedia.org/wiki/Internationalization_and_localization
Compatibility	Exist or function in the same system or environment without mutual interference	http://en.wikipedia.org/wiki/Internationalization_and_localization

Table 1. Quality Attributes Used to Define the Landscape for a Photo Kiosk

This work provided the initial framework for the landscape and we also added Safety since it is relevant to the hardware devices. Generally the set of attributes that should be included in defining this vector will vary from product to product and business needs and should be carefully selected to get an optimized set. Wiegers [2] has provided a list of non-functional software quality attributes with the usage guidelines.

The second vector in the Test Landscape is the time during the product development when the testing should be conducted. This vector may vary depending upon the type of product, software vs. hardware, development methodology, iterative vs. sequential, and the specific shop practices followed by an organization The following diagram describes the main elements of this vector for a typical waterfall software development lifecycle:

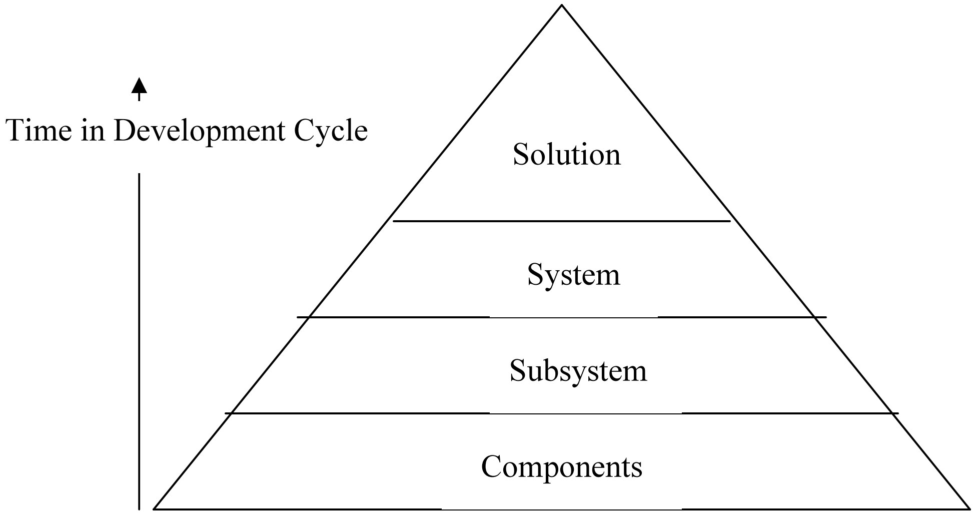


Figure 1. Test Types during the Product Development Stages

In this model integration testing happens at multiple levels as the development proceeds. The development stages can be customized for a particular environment to make the qualification more granular if desired. Craig and Jaskiel [3] and Kaner et al. [4] have discussed various testing stages during software development. The granularity does come at an added cost of qualification for the extra stages.

The two vectors, when combined together, result into the following Test Landscape:

	Functionality	Usability	Installation	Localization
Components	✓	✓		
Subsystem	✓		✓	
System	✓		✓	
Solution	✓	✓	✓	

Table 2. A Simple Test Landscape Showing Horizontal and Vertical Vectors

Our Experience

As discussed earlier, the RPS has a very complex product that includes software, hardware, and the OEM products. In addition, the development is based in the USA, Germany, and UK with each location having its own test team. Since the product is marketed internationally it is important to qualify it against the regulatory requirements. The development is primarily waterfall with multiple test-fix cycles after the "functionality complete" milestone has been reached.

The solution components are tested by the individual development teams at different levels i.e. subcomponent or unit, subsystem or module and system. Printing devices are also individually qualified for performance, reliability, and regulatory as needed. Since the teams are globally dispersed the testing is carried out in multiple places. A high level of coordination is essential for a successful overall solution testing.

The Quality and TCE (Total Customer Experience) director organized a taskforce to develop a test strategy to assure that:

- The testing was effective with no high priority defects found in the field meaning no test escape
- The testing was efficient to optimize the qualification cost
- The product had the intended quality measured against the release criteria.

The taskforce included stakeholders from development, quality assurance, custom product engineering, service and support, regulatory, and human factors engineering. Since the group consisted of development and test managers and leads from multiple test areas a general discussion started around what attributes should be tested and who should own what level of testing.

HP has well established test attributes and it was easy to create a basic attribute list that included Functionality, Localization, Usability, Reliability, and Performance as listed in Table 1. The subject matter experts from Service and Support and Regulatory brought in their perspectives which led to the creation of a broader well rounded list of attributes. An organization can build its own list of attributes that adequately characterizes the product quality.

The group then started to discuss different stages in the in the development when these attributes should be tested. Since the solution is made up of hardware, software and OEM products, the levels had to represent all the stages involved in each development. The OEM products could only be tested at the system level while the hardware and software testing could begin as soon as a component development was complete. There was no clear consensus on the stage names or definitions and the team struggled in getting alignment on characterization of these stages. Finally an agreement was reached to use the simple notion of levels (Level 1, Level 2 etc.) to match the development stages. For example Level 1 represented the Unit/subcomponent, Level 2 the module/subassemblies and so on and so forth. The equivalent of levels is shown in our tables to avoid confusion.

The quality attributes and the test stages together provided the framework for the landscape. We used the landscape table to assign and agree upon ownership of testing for each attribute at each level. The group developed a landscape for each component especially hardware and an overall landscape at the product level to provide efficient test planning at all levels.

Table 3 shows a complete test landscape for a printing device that was a component of the solution.

	Functionality	Reliability	Serviceability	Performance	Regulatory	Safety	Output Quality
Subcomponents	Dev. Team	Dev. Team	CPE	Dev. Team	NA	Dev. Team	Dev. Team
Component	Dev. Team	QA	CPE	QA + Dev. Team	QA	QA	Dev. Team

System	Dev. Team + QA	QA	CPE	Dev. Team + QA	QA	QA	Dev. Team
Solution	QA	UNKNOWN	NA	CPE	NA	NA	CPE
Alpha	CPE	CPE	CPE	NA	NA	NA	NA
Beta	Retailer	Retailer	Supp	UNKNOWN	NA	NA	Supp
Acceptance	Retailer	Retailer	Supp	Supp	NA	NA	Supp

Table 3. Test Ownership of a Printing Device

The abbreviations used in the table are:

Dev: Product Development
QA: Quality Assurance
Supp: Customer Support
CPE: Custom Product Engineering
NA: Not Applicable

Both Beta and Acceptance test stages are focused on testing on the retailer site for the end customer use.

Some typical characteristics of the landscape during component development are:

- The Development Team has a heavy role to play in the beginning and their involvement decreases as the component/product development matures. At the same time, the involvement of other specialty teams increases as we move towards the final product. This is often the case as the components are assembled and the solution starts to exhibit end-product characteristics such as Usability, Performance that require testing by subject matter experts.
- There may be unresolved areas of testing that still need to be finalized. They are highlighted as UNKNOWN and can become potentially critical issues if not resolved early in the program.

The same test landscape can also be used to communicate test status, as shown in Table 4. Once again the component is a printing device (same as in Table 3).

	Reliability	**Performance**	**Regulatory**
Subcomponents			
Finisher	QA – behind schedule	Dev. Team + QA – behind schedule	QA – on track
Engine	QA – on track	Dev. Team + QA – behind schedule	QA – on track
Component			
Printer	QA – behind schedule	Dev. Team + QA – behind schedule	QA – on track

Table 4. Tracking Status for one of the Components of the Photo Kiosk Solution

Table 4 is a snapshot at a milestone in the product development lifecycle where the component was being evaluated. The information was used as a part of the dashboard to inform the upper management.

Table 5 represents the solution test landscape for the product.

	Functionality	Reliability	Serviceability	Performance	Regulatory	Security	Output Quality
Subcomponents	NA	NA	NA	NA	NA	NA	NA
Component	NA	NA	NA	NA	NA	NA	NA
System	NA	NA	NA	NA	NA	NA	NA
Solution	CPE	CPE	UNKNOWN	CPE	NA	UNKNOWN	CPE
Alpha	CPE	CPE	UNKNOWN	CPE	NA	NA	CPE
Beta	SUPP	SUPP	SUPP	SUPP	SUPP	SUPP	SUPP
Acceptance	Retailer	Retailer	Retailer	Retailer	Retailer	Retailer	Retailer

Table 5. Photo Kiosk Solution Test Landscape

The attributes at the levels prior to the solution level have been marked NA since testing at those levels had already taken place during the component development and system testing. For example, Table 4 shows that the Printing Device has been tested at all levels and is now being included in the solution. The Regulatory testing was not shown since the solution components subject to regulations have been tested at one more levels earlier in the development.

Once again the Test Landscape revealed that there were some areas with missing test ownership. The test landscape identified these gaps and raised awareness to the program management teams. This helped us focus on the critical business needs and achieve the desired level of quality and test effectiveness. Using the test landscape, we also discovered that there was a fair amount of overlap between the system and the solution testing. This was primarily due to the lack of clear definition of the two levels and lack of clarity of the roles and responsibilities of the two teams involved in qualification. The test landscape provided a clearly understandable framework which enabled the two groups to align on what the solution testing must accomplish which is different than the system testing. This led the solution team to consider typical use case scenarios such as “Busy Mom” which was portrayed as some one who did not have time to read the instructions and intuitively proceeded to produce her memorabilia. It resulted in an effective user scenario testing which was not being performed earlier. Solution reliability was also another area where testing improved.

Aligning Test Landscape with the Product Development:

To be effective, the test landscape must be designed very early in the development lifecycle. The quality attributes should be determined immediately after the product use cases have been established and the software system requirements are complete. This is equivalent to the requirement definition phase of the waterfall development or the release planning milestone of the agile development.

As the product development proceeds, the landscape must be reviewed and updated if necessary at the various checkpoints and milestones. Our experience was less than perfect with the landscape review. Some component teams proactively reviewed their test landscape while others had to be reminded to complete this activity. There were instances where the review was inadequate. By the time product was released, there was a strong emphasis on the methodology and a better understanding of how it should be utilized.

Benefits of Test Landscape

The Test Landscape has multiple benefits that contribute to a quality product without any additional cost. The following section discusses these benefits in detail.

Test Effectiveness

The landscape builds the test effectiveness by making sure that each applicable quality attribute is tested at an appropriate time during product development thereby providing test coverage from the unit test to the solution test. An example would be to test the performance at component, subsystem, system, and solution level to achieve the intended solution performance. Early testing and defect removal leads to a lower development cost and a superior quality product as the longer a defect stays in the system the more expensive it becomes to fix it [5, 6].

Test Efficiency

Establishing early ownership and clear definition of each test area eliminates duplication of testing, establishes clear roles and responsibilities, and provides a mechanism where testing effort is well understood and is not an afterthought.

Test Coordination

The Test Landscape, after having determined the important critical test areas, can provide effective test coordination to balance resources, assigning testing tasks to appropriate teams, and placing mechanisms in place to analyze test progress and test results.

Scalability

The test landscape is scalable from the component to subsystem, to system and all the way up to the solution level. Depending upon the scope of the product, both the quality attributes and the testing stages can be altered to achieve effective testing.

Customization

The user of the landscape has the liberty of focusing on what is most important to their environment. At times, especially when breaking out into new markets, the functionality is of paramount importance while other quality attributes may not play such an important role. For RPS product it was important to provide excellent usability so that a novice user from the street can get his/her memorabilia while high performance was less critical.

Status Reporting

At every checkpoint or milestone during the product development, the landscape provides a mechanism to track the testing status and thus evaluate the product quality and any schedule risks. At the beginning of the product the landscape can be used to establish the ownership and then as the product development moves along, to evaluate if the intended testing has been performed or not. If, for some reason, the planned testing could not be achieved, a risk analysis can be carried out and the mitigation plans can be put into place.

Conclusion

It all comes down to product quality within the well known constraints – scope, schedule, and resources. Our experience shows that use of the Test Landscape in test planning contributes to the higher product quality, shortens the schedule and optimizes the testing resources. The higher product quality is achieved by testing all the relevant product attributes based upon business needs at the right time in the product development. Identifying and removing unintended duplication contributes to both lower cost and shorter schedule. In most cases testing is the last activity in the product development and is on a critical path. Establishing the testing gaps early in the lifecycle and along the product development helps risk mitigation and potential schedule slip especially in the large organization where each group is focusing on a component or a subsystem.

Acknowledgement

The author would like to recognize the RPS Quality and TCE director, Lee Mason and the members of the taskforce, Amy Battles, David Boal, Tim Dummer, Bernardo Gutierrez, Rick Johnson, Shelly Reasoner, Todd Walker, and Wayne Westly for their enormous contribution in the development of the Test Landscape concept and their support for its adoption by the RPS Group at Hewlett-Packard.

References:

1. Gupta, Bhushan B. and Beckman, Orhan Ph. D., Quantifying Software Quality – Making Informed Decisions, Pacific Northwest Software Quality Conference, Portland, Oregon, 2006
2. Wiegers, Karl E., Software Requirements, 2nd Edition , Ch. 12, Page 216, Microsoft Press, ISBN 0-7356-1879-8
3. Craig, Rick D. and Jaskiel, Stefan P., Systematic Software Testing, Artech House, 2002, ISBN 1580535089, 9781580535083
4. Kaner, Cem, Falk, Jack and Nguyen, Hung Quoc, Testing Computer Software, 2nd Edition, ISBN: 0-442-01361-2
5. Boehm, B. and Basili, V., "Software Defect Reduction Top 10 List," IEEE Computer, IEEE Computer Society, Vol. 34, No. 1, January 2001, pp. 135-137.
6. Cigital, Case Study: Finding Defects Early Yields Enormous Savings, http://www.cigital.com/solutions/roi-cs2.php

Microsoft
patterns & practices
proven practices for predictable results

Acceptance Testing: A Love Story in Two Acts

Grigori Melnik , Microsoft Corporation
Jon Bach, Quardev, Inc.

For presentation at the 2008 Pacific Northwest Software Quality Conference in October 2008

Abstract

This report is about the experiences of five software professionals who set out to produce a book on acceptance testing for other software professionals around the world. This team, headquartered at Microsoft's Redmond campus in the *patterns & practices* group set out to collaborate not only with each other, but with other Microsoft internal teams and external reviewers. After research, we discovered two major "acts" or phases in play when preparing software for acceptance by a customer: the Readiness Assessment and the Acceptance Decision. This may seem obvious, but we found the existing literature and guidance in these areas to be sparse. This paper is about what we did to elucidate activities inherent in each of these phases.

Author Biographies

Dr. Grigori Melnik is a Senior Product Planner in the patterns & practices group at Microsoft, leading the Process & Engineering Practices focus area. Prior to that, Grigori was a researcher, software engineer, coach and educator with 15+ years of meaningful industrial and research experience. His areas of expertise include agile methods, empirical software engineering, software testing and test automation, and software economics. Grigori is a regular contributor and speaker to software engineering conferences and workshops around the world. Grigori was Program Chair of the Agile 2008 conference and a member of the IEEE Software Advisory Board.

Jon Bach is Manager for Corporate Intellect and lead consultant at Quardev, Inc – a Seattle outsource test laboratory that also provides technical writing, training, and consulting services in software development. In his 13 years' experience in software testing, Jon is most famous as an exploratory testing expert and frequent speaker about the cognition and management involved in testing – most notably as co-inventor of Session-Based Test Management. Jon was also president of the 2007 Conference of the Association for Software Testing in Seattle and a recent recipient of Best Presentation at the 2008 Software Testing Analysis and Review (STAR East) conference.

Rationale

In 2007, the Microsoft patterns & practices team did some research into the existing body of knowledge on acceptance testing practices. We found a noticeable lack of consistency and actionable guidance. Furthermore, we found IEEE's definition limiting:

> *"Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system."* [1]

The main problem with this definition was that it was recursive -- it used the word "accept" in its own definition without explaining what it means. So after some brainstorming, we came up with something we felt was more helpful:

> *"Planned evaluation by a customer (or customer proxy) to assess to what degree a system meets their expectations."*

With help from a group of 10 Microsoft internal and external experts assembled into an advisory board, we had carefully crafted each word of the definition until it made sense to all of us. We took each word and decided to make a chapter out of it to explain exactly what we meant by our definition.

For example:

- **"Planned"** -- What does it mean to plan? Who does it? What considerations are there?
- **"Evaluation"** -- What's involved in evaluation? Who does it? What techniques can be used?
- **"By a customer"** -- Who is the customer? Are they different than the user? Should they be an expert in the domain?

- **"Or customer proxy"** -- Who is an acceptable customer substitute? How might their expectations vary from the customer?
- **"To Assess to What degree"** -- What are the methods to determine whether something is acceptable? Under what conditions might it be more or less acceptable? Is it possible for something to *almost* meet acceptance?
- **"A system"** -- Is it an application or a service? What comprises the system? What is being delivered? Will it change over time?
- **"Meets expectations"** -- What's the difference between implicit and explicit expectations? Are requirements the same as expectations?

These questions not only framed our vision and scope, they provoked even more questions. Answering those questions became an anchoring mission for us because, like software testing, questions often drive the effort. It drove our planning, writing and research.

Our first bit of research (see Appendix) consisted of an anonymous online survey and resulted in data from 127 software professionals of varying titles and responsibilities. It helped us discover the following:

- Acceptance testing was not being addressed as an engineering discipline.
- There was no coherent story about how to plan and execute notions of "acceptance".
- There were no specific "how-to's" centered on how to actually plan and execute acceptance testing activities.
- There was no guidance for knowing when you're done with acceptance testing.
- There was no guidance around how to get a good customer (or customer proxy) to participate in providing acceptance tests.
- There were minimal testing checklists.
- There were lots of non-actionable items from IEEE standards, which tend to include recursive definitions.

- We found some research on acceptance testing, but results were not validated by the industry.
- Most engineering teams look at acceptance testing as a "necessary evil".
- With existing notions of acceptance, tester and customer roles are not well-defined, leading us to be confused about who does what and what Functional Acceptance vs. Customer Acceptance means.

From this, we decided to create another non-scientific online survey to get more depth. It contained a mix of multiple choice, multi-select and open–ended questions like:

- *Choose the definition that is most closely aligned with your understanding of what -acceptance testing is.*
- *On your current team, who specifies the acceptance criteria for features/stories?*
- *On your current team, who specifies the acceptance criteria for product/service releases?*
- *How are acceptance criteria communicated to the development team?*
- *How would you prefer the acceptance criteria to be communicated to the development team?*
- *How do you know that you are done, i.e. the product is ready for release?*
- *What do you expect our guide on acceptance testing to have to be useful to you and your team?*
- *Which of the following types of testing should not be included in acceptance testing?*
- *Please share a typical acceptance test or acceptance criteria.*
- *In your opinion what is the biggest challenge with acceptance testing on your project?*

- *What is the level of customer involvement on your project?*
- *How much do you outsource or offshore part or all of your development, testing, or acceptance testing?*

From this feedback[2], we created 3 guiding missions:

1) Inject Acceptance Testing into all phases of a project.
2) Promote it as a discipline (not a series of tasks but a context-driven concept).
3) Promote the notion or "readiness testing" as well as acceptance.

Furthermore, we wanted to have how-to's and checklists – artifacts around planning and execution that were actionable and scalable, and we wanted to ground them to specific examples using a sample application.

Process

We took an Agile development approach, first creating a project wiki for team members to communicate, post content, and keep track of progress.

Here is a diagram of our process:

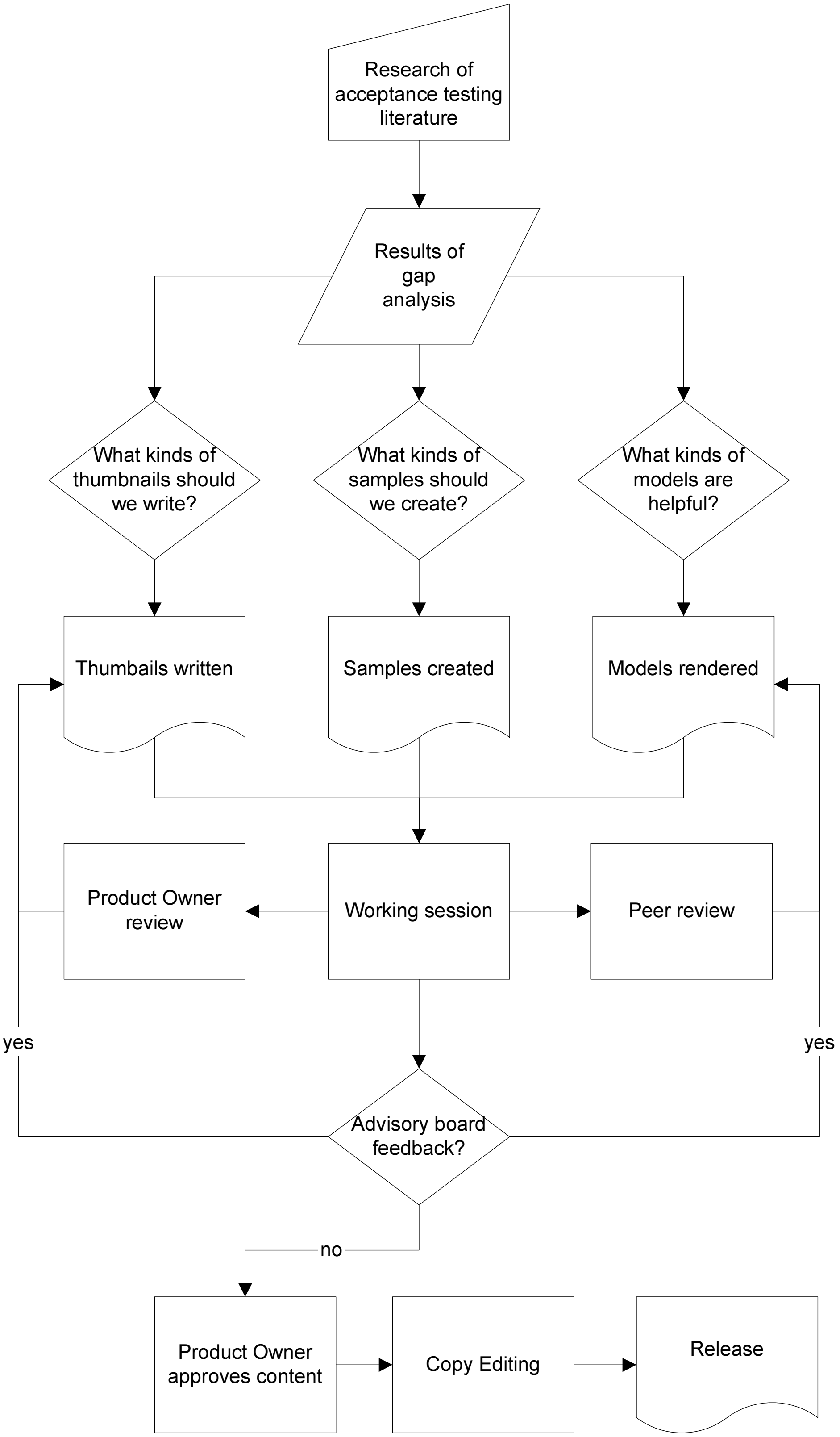

Research of acceptance testing literature
Results of gap analysis
What kinds of thumbnails should we write?
What kinds of samples should we create?
What kinds of models are helpful?
Thumbails written
Samples created
Models rendered
Product Owner review
Working session
Peer review
yes
yes
Advisory board feedback?
no
Product Owner approves content
Copy Editing
Release

Figure 1: Dataflow diagram

What the diagram doesn’t show very well is that we produced content in weekly iterations and had daily stand-up meetings where team members reported their progress for the last 24 hours.

Our story backlog was a list of questions we thought the readers would have about acceptance testing, which became post-it note tasks on a whiteboard. When at least 2 pages of content were created to answer each “story” (and when we could tell a good narrative about that topic), we considered that topic a “thumbnail sketch” and marked it as done, meaning it was ready for someone else on the team to review.

Content structure

The following is a list of some of the approximately 50 thumbnail sketches we produced that we wanted readers to consider in planning and doing their readiness and acceptance testing activities. They cover test processes, requirements, and test design practices, test management, test automation, functional and parafunctional testing, and test oracles:

Test Processes

Exploratory Testing – an approach to testing where testers are in control of the design and execution of their testing and use of techniques as they learn about the product. The point is to react to new or emerging information, so the tester is encouraged to change tactics to follow hunches and discover important issues.

Acceptance Test-Driven Development – a way to write software, starting with the customer requirements and the customer's specified acceptance criteria/tests for those requirements, and using them as the basis for all development.

Incremental Acceptance Testing – when the development of functionality is organized so that individual features can be acceptance tested as soon as they are deemed ready by the supplier team instead of waiting until the end of the project when all of the features are done.

Regression Testing – testing to ensure that perceived quality is not going down due to new features implemented or bugs fixed. Regression testing minimizes risk of new problems by running a standard set of tests on each release candidate.

Script-Driven Testing – preparing tests in written form well ahead of their execution.

Test-Last Acceptance – when acceptance testing is done at the same time as the decision -- after all development and readiness assessment activities have been completed.

Vision-Scope – a way to frame the objectives of the mission by meeting with stakeholders about the following 5 elements: Customer, Needs, Product, Value, and Purpose.

Risk Assessment – identifying things that could go wrong on the project and classify by likelihood and impact to help prioritize the risk mitigation activities including, but not restricted to, testing.

Requirements Practices

Requirements – explicit, stated expectations of a user or customer – the desires for functions that solve some kind of problem or set of problems – but they may also be implicit, assumed and unstated.

Use Case Modeling – a way to describe the functional requirements of a software-intensive system. It focuses on the goals of what the system's users would like to achieve while using the system and what the system needs to do to help them achieve the goals.

User Stories – a way to manage highly-incremental development by deriving and documenting user actions and behaviors from requirements and establishing those as stories that serve as a unit of development work.

Test Design Practices

Regulatory Considerations – sometimes the customer or supplier is bound by forces that affect the way they approach acceptance testing. Government agencies have their own criteria, standards and rules that govern the way software can function.

Scenario Testing – unlike functional tests based on use cases, scenarios typically incorporate behavior from many use cases into the same test based on actual or possible usage behaviors. Scenarios are typically expressed in natural, ubiquitous language.

Soap Opera Testing – a test technique that get its name from fictional daytime television shows that have their roots in the 1950's and 60's when sponsors were often soap companies. An opera is an epic story, either a long series of events or a short series of very dramatic events happening to fictional characters.

Test Management

Cycle-Based Test Management – a method for managing testing effort where the test phases of the project are subdivided into two or more test cycles separated by periods of time set aside for bug-fixing.

Session-Based Test Management – a method for managing testing effort by compartmentalizing testing activity into time-boxes called sessions. It is most commonly used to manage exploratory testing but could be used for managing any kind of test execution.

Test Automation

Test Automation – an approach to testing that leverages the speed and power of computers to run tests that benefit from repetition or complex calculation.

Data-Driven Testing – a technique for reusing the same test logic on many sets of data values. The test is structured to read the input and corresponding expected output data values from a file or table and runs the same test logic with each set of data.

Hand-Scripted Test Automation – when automated test scripts are hand-coded in a scripting or programming language by people with enough technical skills to do some programming and debugging.

Keyword-Driven Testing – a technique for separating the specification of tests from the underlying mechanism to execute the tests by structuring test steps as action keywords followed by action-specific arguments.

Record-Refactoring – a common way to leverage the strengths of recorded tests without taking on the weaknesses involves refactoring. Refactoring is a way of re-organizing code script to remove duplication and make the code script simpler and easier to maintain without effecting affecting what it does.

Testing Functional Requirements

Ubiquitous Language – effective communication between business users of software and the technical builders and testers of software requires a common language. Since business people are not likely to learn technical jargon, the technical people must learn to speak "business". This ubiquitous language should form the basis of all communication including the acceptance tests that describe what done looks like.

Workflow Testing – a test technique designed to verify how the system supports or implements a business process by executing a series of user actions toward a given task or objective. They often include tasks carried out by multiple users exercising different part of the system in a business workflow from a beginning state to an ending state.

Business Unit Testing – verifies the behavior of a business algorithm or business rule outside the normal context in which the algorithm or rule is utilized.

Combinatorial Testing – putting attributes of test criteria together to see if there are harmful interactions.

Installation Testing – once a software system is created, you need a way to get the components of the system deployed. Installation testing is ensuring that the deployment and removal of the components works.

Testing Parafunctional Requirements

Parafunctional Testing – testing that goes beyond confirming function, especially if the customer has implicit requirements, or requirements that go beyond a specific function, or expectations that emerged well after they were interviewed about requirements.

Usability Testing – a test technique designed to find out whether the product meets the needs of real users by watching users operate the product while trying to accomplish a specific task in a (nearly) realistic setting.

Security Testing – testing designed to reveal risks, vulnerabilities, attacks, and threats that would compromise valuable user or company data.

Performance Testing – a determination of whether or not the system under test meets the baselines with respect to throughput, bandwidth, or expected response times for user inputs.

Fuzz Testing – a way to test the robustness of an API or program input fields and entry points by feeding it random or pseudo-random data in many forms, usually in an automated manner.

Accessibility Testing – the act of preparing your software to be compliant with principles designed to help by people with varying degrees of capabilities.

Test Oracles

Comparable System Test Oracle – where the pass/fail status of a test is determined by comparing the actual results from the system under test with the result produced by a system with comparable functionality.

Hand-Crafted Test Oracle – when pass/fail status of a test is determined by comparing the actual results from the system under test with an expected result that was previously hand-crafted by a Human Test Oracle.

Human Test Oracle – when pass/fail status of a test is determined by a human subject matter expert inspecting the actual results from the system under test and deciding whether they are acceptable.

Previous Result (Regression) Test Oracle – when the pass/fail status of a test is determined by comparing the actual results from the system under test with the result saved when the same test case was run against the same system at some point in the past.

It's important to note that not all of these thumbnail ideas were created from the start. We let the list evolve and expand based on our research and feedback from the advisory panel; mostly because our best work came from 3-hour discussions we called "working sessions" – dedicated times when the team collaborated on something, whether it is a model, a strategy for production, or a research topic.

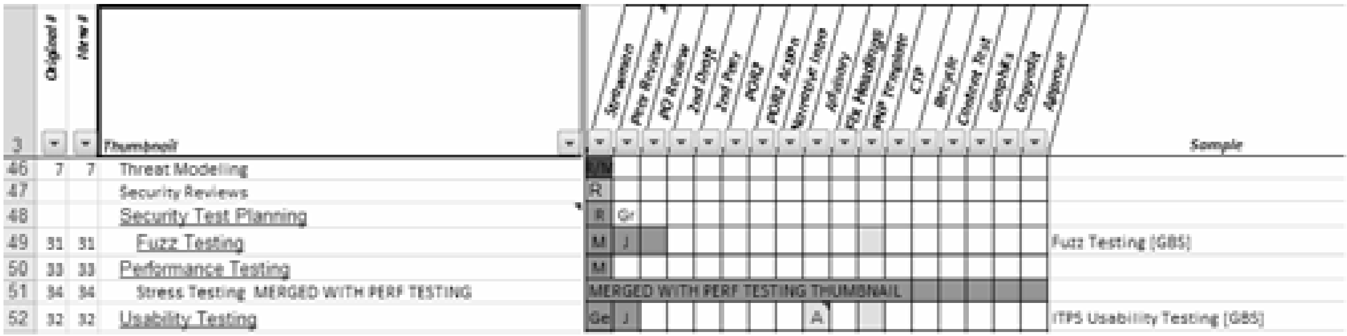
Ultimately, all new stories and ideas for content (thumbnails) were presented and approved by the content owner (Grigori Melnik) and project sponsor.

Table of Contents as project compass

Guiding the working sessions was our table of contents which we kept in an Excel spreadsheet. It evolved from a live TOC on the wiki page, then to a live document in Microsoft Word, then to a live document on Sharepoint exposed through an extranet site so that the reviewer committee could see it.

Each row of the sheet was a thumbnail topic on which we tracked progress. There were 17 levels (columns) of "done-ness", each level getting us closer to final release. This was our main vehicle to gauge whether or not we were still in readiness (ready for a team review) or ready for acceptance (product owner review and release to the web).

Below is a graphic which shows a snippet of a few thumbnail names and its level of done-ness.



3	Original #	New #	Thumbnail	Strawman	Peer Review	PO Review	2nd Draft	2nd Peer	PO#2	PO#2 Accepts	[illegible]	[illegible]	[illegible]	[illegible]	CTP	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	Sample
46	7	7	Threat Modelling	[illegible]																	
47			Security Reviews	R																	
48			Security Test Planning	R	Gr																
49	31	31	Fuzz Testing	M	J																Fuzz Testing (GBS)
50	33	33	Performance Testing	M																	
51	34	34	Stress Testing MERGED WITH PERF TESTING	MERGED WITH PERF TESTING THUMBNAIL																	
52	32	32	Usability Testing	Ge	J								A								ITPS Usability Testing (GBS)

Figure 1: Table of Contents: our main readiness / acceptance gauge for each artifact we produced.

Where did the ideas for thumbnail come from? We created a list of 120 questions we knew readers would likely ask, including:

- What are the common reference models?
- What are the kinds of acceptance testing?
- What is the lifecycle of an acceptance test?
- What are the possible phases?
- What is Readiness / Acceptance and how does it help me conceptualize AT?
- What are the existing standards for AT?
- What is the difference between Agile (incremental) and Waterfall?
- Who is going to do the testing?
- How do you select proxies?
- What are some questions to ask test outsourcers?
- What are the strategies for AT?
- What does an AT plan look like?
- How much detail is needed in the plan?

- Does it cover enough risks?
- How are the risks represented?
- How to get buy-in from stakeholders?
- What does a Vision Statement look like?
- What does a Scorecard look like?
- What gets tested?
- What kind of test bed should we build for data?
- What tools should be used?
- How does a maintenance contract affect acceptance?
- What kinds of customers might there be?
- How much testing do we need to (plan) to do?
- Who writes the acceptance tests?

From this, we looked for affinities. For example, was there a test technique or approach that would answer more than one question? If so, it might be worth writing a thumbnail about.

Samples

To ground our answers to these questions, we also created samples to illustrate the topics in our thumbnail sketches. Borrowing from Microsoft's Visual Studio Team System's example, we used a fictional software service called the Global Bank Identity Theft Protection System (ITPS).

The sample Vision/Scope for the ITPS feature read as follows:

> *"For current Global Bank premium account holders who need to monitor their accounts for suspicious activity like identity theft, fraud, and infiltration the Identity Theft Protection Service (ITPS) will allow customers to sign up for notification of suspect transactions by email, IM, text, and/or voice that provide general information and a URL for secure login to review transaction details unlike that for non-premium account holders (less than $50,000 in assets) or premium account holders at other competing banks."*

The samples we produced around this to help ground the content in our thumbnail sketches included:

- A sample vision-scope
- An all-pairs modeling sample

- A bug chart sample
- A risk assessment sample
- An exploratory session report
- A sample performance testing report
- A sample test plan
- A workflow example
- A soap opera test
- A scenario sample

It was our hope that creating samples like this would help readers understand why a thumbnail sketch of an n approach or a technique could be meaningful, either during readiness or acceptance.

Models

Other than questions, thumbnail sketches, and samples, we knew we had to create ways to illustrate our research and ideas to have a strong visual component to the book. Models served as a way to represent guiding principles for us in our work, that sometimes framed discussions for specific topics.

For example, from one working session came the idea for a working model about how decisions get made:

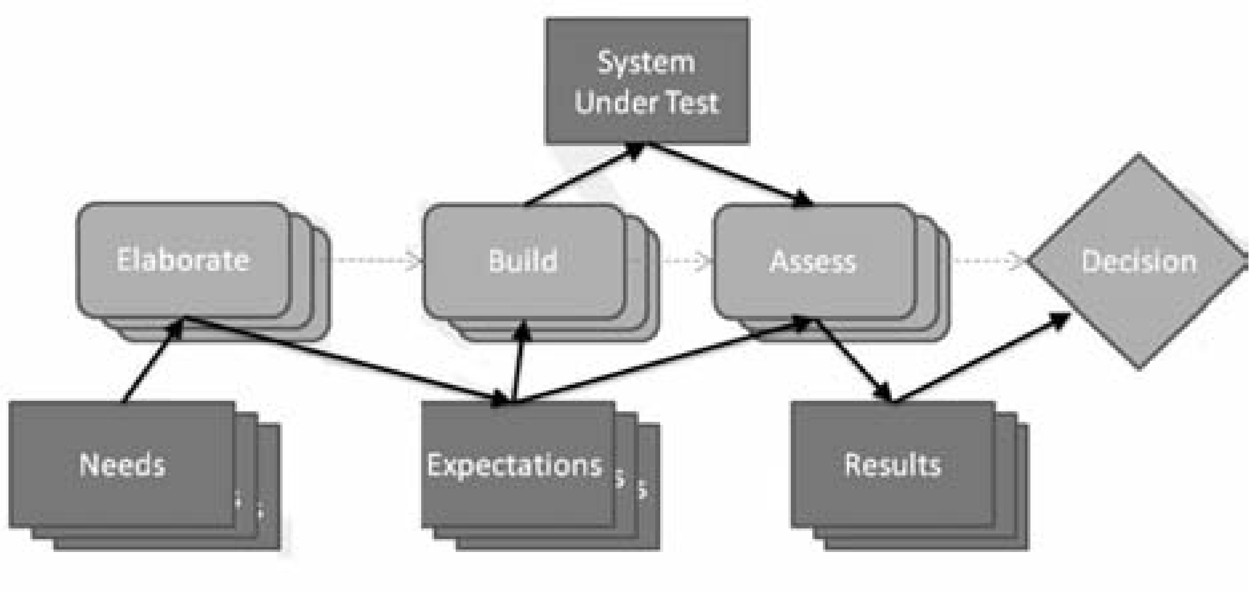


Figure 2: Decision-Making Model

The diamond on the right represents the decision which is made based on the test results. The test results are based on the testing / assessment activity which assesses the system-under-test against the expectations. The expectations of the system-under-test were defined based on the needs / requirements of the users. While many of our thumbnail sketches describe how to do the assessment activity, others describe ways to define the expectations based on the needs. That is one of the reasons our guide has a number of requirements-related practices; it's not about testing, it's about acceptance.

But it was only when expanding on this did in another working session did we get the biggest epiphany. It had to do with a simple question: How would we apply this decision-making concept to our own project?

Our answer took the form of what we now call the Gating Model.

(Notice that the figure below is a hand drawn sketch which the authors of this paper decided to leave as-is to talk about the value of low-fidelity prototyping we used on this project. That is, sometimes a hand-written model can be just as effective as one produced with Publisher or Excel when explaining something.)

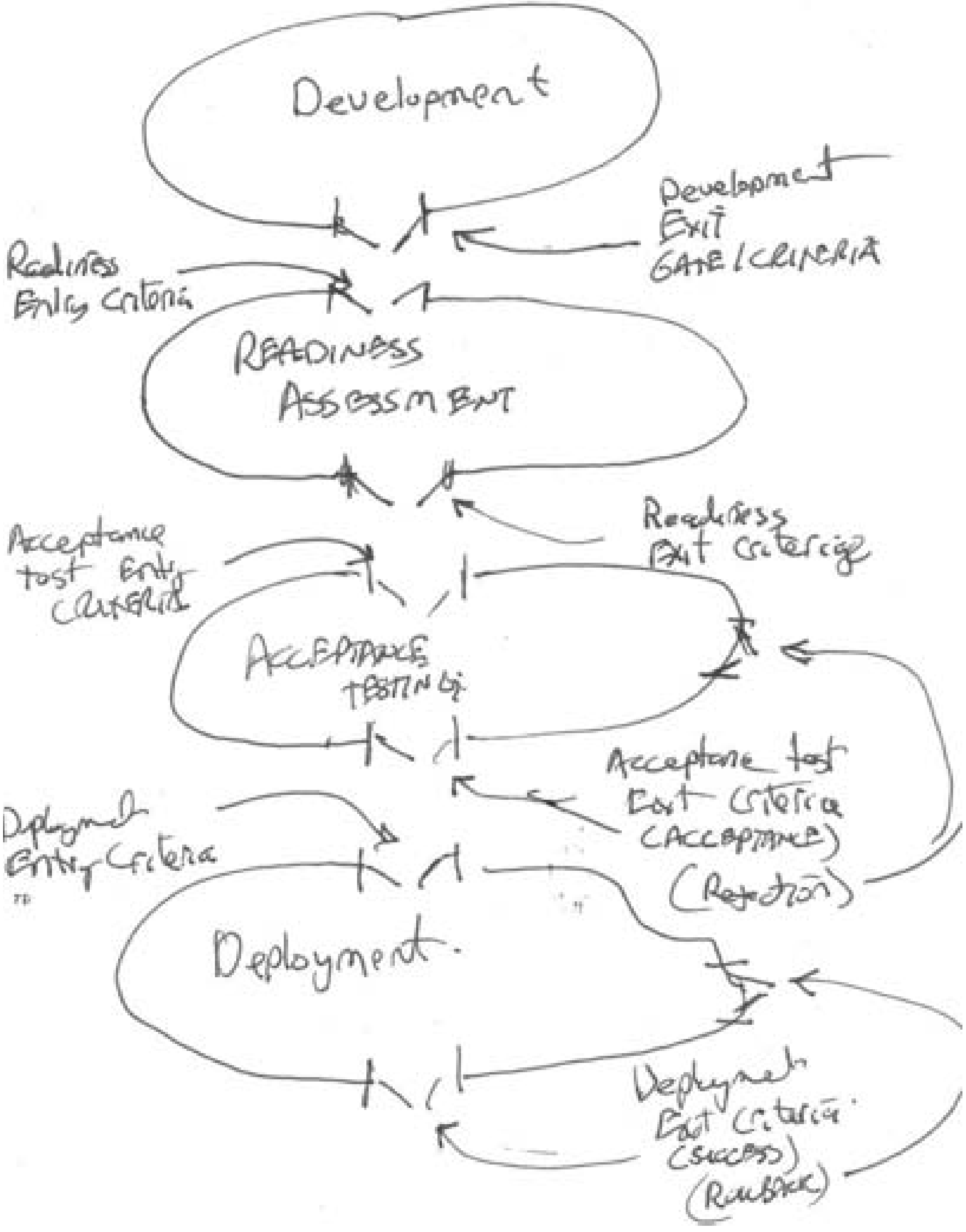

Development
Development Exit GATE/CRITERIA
Readiness Entry Criteria
READINESS ASSESSMENT
Readiness Exit Criteria
Acceptance test Entry CRITERIA
ACCEPTANCE TESTING
Acceptance test Exit Criteria (ACCEPTANCE) (Rejection)
Deployment Entry Criteria
Deployment.
Deployment Exit Criteria (SUCCESS) (RollBack)

Figure 3: Gating Model

Readiness Assessment Phase

Readiness assessment is done by the supplier of the software before declaring the system "ready for acceptance testing". As a result, the gate between

Readiness and Acceptance Testing phases is largely staffed by gatekeepers belonging to the supplier.

For the supplier to feel confident that their product will pass muster with the customer, it may require that they do a lot more testing than the customer might do during acceptance testing. As a result, the testing done as part of Readiness assessment is likely to be much more exhaustive and rigorous, and employ a much wider array of testing techniques than that done during the actual acceptance testing.

Acceptance Testing Phase

The acceptance testing phase is when the customer (or their proxy) is actually executing[i] tests that will help them make the decision to accept or not accept the software. The main entry criteria for the acceptance phase are that the supplier has deemed the software ready for acceptance testing. Secondary criteria may include whether or not the customer is sufficiently prepared to conduct the acceptance testing.

Exit criteria are primarily focussed on whether or not the “accept software” decision has been made. The software is considered in acceptance testing until either

- The customer has accepted it

Or

- The customer has found it so inadequate as to reject it outright. At this point, the ball is back in the supplier’s court until they have done further development and readiness assessment based on their revised understanding of the customer’s expectations.

Challenges

Like most development projects, our effort to produce a book on acceptance testing guidance was not without its challenges.

First, not everyone on the team was a dedicated full-time resource. Other projects competed for our attention and made progress slow at times. The dedication problem was the same for our advisory board, as most of the people who signed up to be on it had other commitments that kept them from helping us flush out important errors and omissions in our content.

The presence of the advisory board made it necessary to abandon the wiki pages we had started at the beginning of the project because it was an internal resource and reviewers could not get access to it. We adopted SharePoint as a hosted content management system for our extranet site.

We also could not identify our velocity -- delivery of some unit over a given time period -- as can be done on most projects that use Agile methods based on a unit of work like story cards or code production. The classic way of doing velocity in Agile is to put an estimate on every piece of content that is meant to pass acceptance and then measure how much of that got done per iteration. There was a time when we did not have the concept of a thumbnail, so it was hard to know what "done" looked like. Even when we had the concept of the thumbnail, they were not equal units of work – some topics took longer to research and write than others. We consciously decided not to estimate how long it would take to complete a given thumbnail topic, but in retrospect, we could have used a first order approximation in the spreadsheet (e.g. squares in the Table of Contents spreadsheet). We also had a tough time predicting what would get done in a given time frame because we did not have the notion of a burn-down chart.

Another challenge was keeping our focus on acceptance testing. The more we researched and produced artifacts about testing, the more the book started to feel like a book on general testing techniques and approaches, not just about how those techniques could be used for acceptance. Creating a template for the thumbnails helped us stay focused and consistent, even though there were several content contributors on the project.

Summary

Based on a new definition of acceptance testing after surveying people in the industry, we used an Agile-style iterative and incremental development process. We strived for a continuous stream of value that started with a backlog of questions which served as "stories" about aspects of acceptance testing we wanted to write about. We used a live table of contents to guide our thumbnail sketches for content, to serve as project status reporting, and for planning future iterations. In addition, we hosted weekly meetings with an advisory board consisting of people in software development who served as our first consumers of our guidance. We used a sample application to produce grounded artifacts to our thumbnails. We held working sessions, which helped us create models and new topics to go into the list of artifacts we thought would be helpful.

We found acceptance testing to be more than just creating a checklist of tests for the customer to run. The more we realized how thoughtful we were trying to be while producing a meaningful book on acceptance testing, the more we saw a parallel between producing software and producing a book – that is, sometimes you have to use a variety of techniques to anticipate how a customer will formally sign-off on their acceptance.

We found the relationship between Readiness (supplier-focused development) and Acceptance (customer-focused testing) to be a reinforcing synergy – a love story, if you will – which frames many kinds of test approaches and techniques to increase the likelihood that customers get their requirements, expectations, and needs met.

References

[1] (* Source: SEI/CMU and Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.)

[2] See Appendix for survey

Appendix

This is a public survey we created and posted online at www.codeplex.com at the beginning of the project to get a sense of how “acceptance” was being performed on a variety of software projects. The objective was to form a foundation of acceptance testing and focus on strategies of readiness and acceptance.

Acceptance Testing Survey

Jun 16, 2008 11:18 AM PST

1. Choose one the following definitions that are most closely aligned with your understanding of what acceptance testing is		
Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system.	52	50%
Planned evaluation of a system by a customer (and / or customer proxy) to assess to what degree it satisfies their expectations.	19	18%
Formal test of a system performed after installation on the customer's premises by an authorized entity with the participation of the supplier to determine whether the contractual obligations are met.	12	11%
Evaluation of a system by a technical tester as a way to transfer ownership of the application from the developers.	5	5%
Verification that requirements will always be met and will be acceptable by all users.	13	12%
Other, please specify	4	4%
Total	105	100%

2. On your current team, who specifies the acceptance criteria for individual features/stories? [Select all that apply]		
Customer	52	50%
Customer proxy	36	35%
Business analyst	34	33%
Program/Project manager	42	40%
Test lead/QA manager	29	28%
Tester	28	27%
Developer	18	17%
Development Lead	16	15%
Architect	12	12%
Release manager	4	4%
Compliance manager	5	5%
Usability specialist	8	8%
External consultant	0	0%
End-users	11	11%
CTO	2	2%
CFO	0	0%
CIO	1	1%
COO	0	0%
Legal counselor	1	1%
Other, please specify	5	5%

3. How are acceptance criteria communicated to/documented for the development team? [choose all that apply]		
Requirements Spec	39	38%
Statement of Work (SOW)	10	10%
Acceptance Document	16	15%
Use cases	25	24%
User stories	28	27%
Scenarios	17	16%
Legal contract/SLA	3	3%
Test plan	26	25%
Design documentation	18	17%
Manual test cases	22	21%
Automated test cases	18	17%
Wireframe	3	3%
Storyboard	8	8%
Face-to-face conversations	24	23%

Email/IM	13	12%
IEEE/FDA/ISO/DOD/SEI or some other standard or regulation	2	2%
Not at all	6	6%
Other, please specify	4	4%

4. How would you prefer the acceptance criteria to be communicated/documented? [choose your top three]		
Requirements Spec	43	41%
Statement of Work (SOW)	6	6%
Acceptance Document	33	31%
Use cases	29	28%
User stories	36	34%
Scenarios	26	25%
Legal contract/SLA	2	2%
Test plan	36	34%
Design documentation	13	12%
Manual test cases	14	13%
Automated test cases	29	28%
Wireframe	1	1%
Storyboard	9	9%
Face-to-face conversations	14	13%
Email/IM	1	1%
IEEE/FDA/ISO/DOD/SEI or some other standard or regulation	2	2%
Not at all	0	0%
Other, please specify	4	4%

5. How do you, in your role, know that you are done, i.e. the product, application, or service is ready for release? [choose all that apply]		
When customers realize enough value	15	15%
When user stories/requirements are complete	40	39%
Development task is completed, tested, demoed to and approved by customer	60	58%
Scorecard criteria are met	6	6%
Product has no critical problems	33	32%
Product has sufficient benefits	11	11%
The value of the benefits in the product outweigh the problems	10	10%
Impact of the last code modification is minimum and does not require full test pass	3	3%
Last test pass resulted in bugs that were relatively low severity and in low priority features	16	16%
Overall trend of bugs found in test passes indicates that code is becoming more and more stable	12	12%
Test coverage of the last successful test pass is acceptable	23	22%
All functional and non functional quality gates identified for the release have been met	38	37%
All deferred active bugs have been triaged / documented as known issues and shared with customer	28	27%
Regression test pass doesn't result in reactivation of fixed bugs	27	26%
Documentation is written/updated and edited	19	18%
Risks have been identified and mitigation strategies formulated	18	17%
Further time spent testing, designing, building is deemed more expensive (harmful) than helpful	12	12%

When the checklist is completed	16	16%
When there is no more time	17	17%
When there is no remaining budget	8	8%
Other, please specify	4	4%

6. How do you address bugs found during acceptance testing?		
Discard the current release candidate and request a new one with the bugs fixed	57	56%
Defer the bugs and address them as part of the service level agreement	13	13%
File a change request	19	19%
Other, please specify	13	13%
Total	102	100%

7. How would your acceptance testers define a bug?		
Non-conformance to the written specification or contract (SOW)	8	8%
Does not meet an explicit, written requirement in a requirements doc	24	24%
Does not meet an expectation (implicit requirement)	25	25%
Does not adhere to a story card	12	12%
Anything the customer isn’t happy with when they see the software	27	26%
Other, please specify	6	6%
Total	102	100%

8. At what phase of the project does your acceptance tester get involved?		
During project planning	17	17%
During requirements gathering	20	19%
During system design	7	7%
When development starts	15	15%
When development is finished	28	27%
When the customer sees it for the first time	7	7%
Other, please specify	9	9%
Total	103	100%

9. From the perspective of defining requirements and performing acceptance testing, which of the following applies?		
Customer performs both	19	18%
Customer and in house test team perform both	34	33%
Customer defines requirements only, and in house test team does acceptance testing	26	25%
Customer defines requirements and outsources acceptance testing to an external vendor	2	2%
Customer defines requirements and end users perform acceptance testing	16	16%
Other, please specify	6	6%
Total	103	100%

10. What tools do you use for acceptance testing? [Select all that apply]		
None	43	43%

Commercial tools (Visual Studio Team System, Rational suite, Seapine products, etc.), please specify below	42	42%
Open source tools (Fit/Fitnesse/Selenium or similar), please specify below	17	17%
In house tools	21	21%
Other, please specify ____________________	1	1%
Specify:	8	8%

11. What do you expect our guide to contain to be useful to your team? [choose three]		
Descriptions of the various acceptance testing practices/techniques?	82	80%
Case studies/stories from the trenches	40	39%
Checklists	50	49%
How-to's	44	43%
Examples : test case/test strategy/test plan/test script	70	68%
Sample reports	18	17%
Metrics and how to use them	35	34%
Tips and Tricks / Heuristics	40	39%
Anti-patterns and known mistakes	59	57%
Other, please specify:	6	6%

12. Please share a typical acceptance test or acceptance criteria. [Do not worry about giving the context, we would like to simply see the style and structure of your acceptance tests]
56 Responses

13. In your opinion what is the biggest challenge with acceptance testing on your current/last project?
69 Responses

General Questions

14. Which best describes the process your team follows?		
Agile (Scrum, Crystal, Lean, Extreme Programming, and so on)	54	55%
Formal (Tayloristic, waterfall, spiral, and so on)	30	30%
Other, please specify	15	15%
Total	99	100%

15. What is (are) your role(s) on the team? [Choose all that apply]		
Customer	1	1%
Customer proxy	9	9%
Business analyst	7	7%
Program/Project manager	24	23%
Test lead/QA manager	18	17%
Tester	19	18%
Developer	32	31%
Development Lead	37	36%
Architect	30	29%
Release manager	9	9%

Compliance manager	0	0%
Usability/User experience designer/specialist	6	6%
External consultant	8	8%
Technical writer	5	5%
End-user	1	1%
Executive (CTO, CFO, CIO, COO and so on)	6	6%
Legal counselor	1	1%
Ethnographer	1	1%
Other, please specify	2	2%

16. What is the level of customer involvement/commitment on your project?		
Very High (on-site customer)	18	18%
High (near site customer or customer proxy)	28	28%
Moderate (customer present at weekly status updates)	22	22%
Low (customer gets only involved once a month)	15	15%
Very Low (customer gets only involved at the beginning of the project and at the final demo)	11	11%
What customer?	4	4%
Other, please specify	3	3%
Total	101	100%

17. How many people are on your team [again, please think of the most recent project]?
98 Responses

18. What percentage of your development and testing effort does your team outsource?
96 Responses

19. If you have a story or a lesson learned about acceptance testing that you'd like to share, feel free to write it below or provide an email address if you'd like to be contacted and interviewed.
26 Responses

Building a Software Testing Strategy

Karen N. Johnson

Who am I?

Karen N. Johnson	
Independent Software Test Consultant	www.karennjohnson.com
Hosted on Tech Target	http://searchsoftwarequality.techtarget.com
My blog	http://www.testingreflections.com/blog/3804
Co-founder of WREST workshop	http://www.wrestworkshop.com/Home.html
Director	http://www.associationforsoftwaretesting.org/drupal/executives

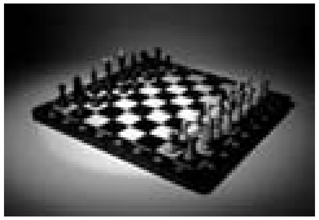
About this presentation

- There are many ways to build a test strategy.
- Take these ideas as ideas not as absolutes.
- A look at the principles of the context-driven school of software testing and how the principles apply to building a strategy.
- A list of components you might include in your strategy.

The Seven Basic Principles of the Context-Driven School

1. The value of any practice depends on its context.
2. There are good practices in context, but there are no best practices.
3. People, working together, are the most important part of any project's context.
4. Projects unfold over time in ways that are often not predictable.
5. The product is a solution. If the problem isn't solved, the product doesn't work.
6. Good software testing is a challenging intellectual process.
7. Only through judgment and skill, exercised cooperatively throughout the entire project, are we able to do the right things at the right times to effectively test our products.

Checklist to Consider



- Project stakeholders
- Product
- Context
- Project scope
- Risk analysis
- Types of testing
- Test environment
- Test data
- Resources
- Estimates
- Project plans & milestones
- Users

Building a Testing Strategy © Karen N. Johnson, 2008 Slide 5

Project Stakeholders

- Determine who they are
- Collect their opinions
- Uncover their assumptions
- Enlist their support
- Keep them informed

Building a Testing Strategy © Karen N. Johnson, 2008 Slide 6

Project Scope

- Testing objectives
- Testing in/out for release
- Features in/out for the release
- Full release, partial rollout
- Communicating the testing scope

Building a Testing Strategy © Karen N. Johnson, 2008 Slide 7

Product

- Features & Functions
- Core product elements for regression
- Product documentation
- Interface with other applications
- Components outside of the core product

Building a Testing Strategy © Karen N. Johnson, 2008 Slide 8

Context

- The product
- The release
- Constraints
- Formal vs. informal
- Users
- Initial product release or subsequent release

Risk Analysis

- Formal:
  - Failure mode and effects analysis (FMEA)
  - Regulations
  - Compliance
- Informal:
  - Polling team members
  - Based on product history
  - Based on customer support and field tech input
- Think beyond functional risks: content, security, data, multi-user, performance, multilingual, cross browser, cross operating system, upgrading customers

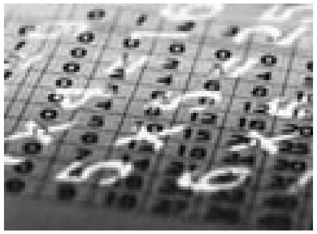
Types of Testing

1. Black box
2. White box
3. Gray box
4. Functional
5. Automated
6. Regression
7. Security
8. Data
9. Exploratory
10. Performance
11. Stress
12. Multi-user
13. Cookies
14. Compatibility
15. Interfaces
16. Developer
17. Unit
18. Integration
19. System
20. User Acceptance
21. Installation

Test Environment

- For some projects, getting an environment is easy.
- And for some projects, getting an environment established can be one of the most challenging aspects of the project.
- Maintaining the environment: the software release and the data.

Test Data



- Depending on the product, test data can be an essential factor.
- In other cases, getting a copy of production data might be restricted.
- If test automation is a type of testing, building test data might be needed.

Resources

- Existing staff
- Contract staff
- Test lab
- Test Equipment
- Test Data
- Tools
- Utilities

Estimates

- Methods of building estimates.
  - Top down
  - Bottom up
  - WBS
- Include other people in estimating.
- Look at a calendar to plan out of office time.

Project Plans & Milestones

- Project plans
- Development milestones
- Testing milestones
- Milestones for elements of the strategy, such as hiring staff, acquiring an environment, purchasing test tools

Users

- Intended use
- Potential misuse
- Happy paths and rainy days
- Thinking about users and their needs help drive us to recall: What are we building? And why.

Are we done yet?

- Release or acceptance criteria
- Go, no-go release meeting
- Post-release monitoring
- Learning from the project
- Planning your next strategy

Keeping the Strategy Alive

- Review the strategy throughout the project.
- Keep stakeholders informed through status updates.
- Revise and distribute the strategy at key intervals.

References

- Webcast: "How to plan your software test projects" Hosted by Tech Target. Presenters: Karen N. Johnson and Michael D. Kelly. Link: http://searchsoftwarequality.bitpipe.com/detail/RES/1196440596_441.html?bucket=WC&topic=306121
- Satisfice Heuristic Test Planning Context Model: http://www.satisfice.com/tools/satisfice-cm.pdf
- Satisfice Test Planning Guide: Building the Plan: http://www.satisfice.com/tools/build-the-plan.pdf
- Satisfice Test Plan Evaluation Model: http://www.satisfice.com/tools/tpe-model.pdf
- "Developing a Project Test Strategy," Michael D. Kelly, http://www.informit.com/articles/article.aspx?p=355875
- "Documenting your software test project," Karen N. Johnson and Michael D. Kelly http://searchsoftwarequality.techtarget.com/tip/0,289483,sid92_gci1284632,00.html
- "Building a Software Test Strategy," Karen N. Johnson http://www.informit.com/articles/article.aspx?p=1146504
- Blog post, James Bach, see: http://www.satisfice.com/blog/archives/63
- Webcast: "Making Sense of Software Tests," Hosted by Tech Target. Presenter: Karen N. Johnson http://searchsoftwarequality.techtarget.com/guide/allInOne/category/0,296296,sid92_tax308063,00.html
- Risk Analysis for Web Testing, Karen N. Johnson http://www.karennjohnson.com/pdf/Risk_Analysis.pdf
- The Seven Basic Principles of the Context-Driven School see: http://www.context-driven-testing.com/

Test Automation Design Pattern

A pattern for automatic test case generation

Lian Yang
liany@microsoft.com

Abstract

Test automation itself is a software engineering project and its effectiveness, efficiency, and maintainability quite often present the same engineering challenges as other software projects. The development community has been embracing the "design pattern" concepts for 15 years, in an effort to address software engineering difficulties and formalizing design and coding process. While mentioned in various papers ([3]), design pattern practice is still not well practiced in the software test automation community. Ill-planned, and roughly designed test automation projects are still wide-spread. This paper tries to define a "test case automatic generation" design pattern, which addresses the most common and fundamental test automation engineering issues facing most testers today. The concepts presented in this paper has been implemented by the author and his team in testing several Microsoft© products.

Glossary

Terminology or Acronym	**English**
SUT	Software Under Test
SPEC	(product) Specification
TTS	The test state
TCGEP	Test Case Generation and Execution Pattern

1. Test Automation

1.1. Definition

A Test automation system is a software system that exercises a target software product (or SUT) in order to detect defects in SUT. Software defects can be any of the following things:

- Mal-functions that cause the user to be unable to use the software, such as hanging, Access Violation (in Windows), and system crashing
- Not SPEC compliant
- Bad user experience that may or may not be in conflict with the SPEC.
    - Slow response
    - Using too much resource

    - All kind of things that don't make users job easy
- Unwanted side effects
    - Data corruption
    - Security issues

Most of software defects (or bugs) fall into the above categories. An ideal test automation system should be able to catch a fair percentage of the above kind of bugs. From the above definition, test automation should consist of the following elements (in red):

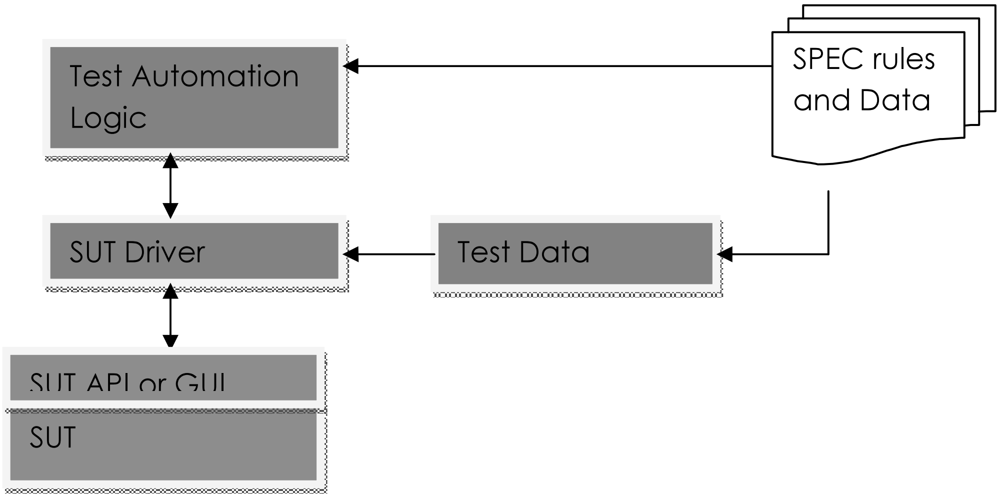


Fig. 1: components for a generic test automation

The above diagram can be thought as a high-level design pattern, from which other design patterns can be derived. The most important concepts introduced in the above generic pattern are:

1) Separation between test logic and SUT driver
   This allows the design patterns such as TCGEP to express the design concepts in abstract terms while leaving all SUT specific details to the SUT Driver
2) Derive test data directly from SPEC in form of data template or using advanced SPEC to case generation tools (such as Microsoft Research's SPEC#)
   Hard-coded test data is as bad as hard-coded magic numbers or strings in product code. There is really very little justification for test automation applications to continue using hard-coded magic data.

1.2. A sample SUT application

We need to use a sample SUT application to illustrates concepts and points through out the paper. Our sample application is code named ***Happy-Fish***. Happy-fish is a simple on-line banking system with the following components:

- A WEB UI Front End
    - Customer pages
        - A logon page with user name, password fields, and an OK button
        - A user page with account names as link buttons, and a logoff button
        - A user account detail page with a data grid control and a logoff button
    - Admin pages with a button "add user", that leads to a "Add New User Page"
        - Assume that admin pages are only known by the Admin and privately accessed (without password protection) in order to simply the demo application
        - Add New User Page with user name, password fields, and an Add button

- A business logic mid-layer behind the front end on the web server
- A SQL DB that stores customer account information and personal data

The following diagram shows the components of the system:

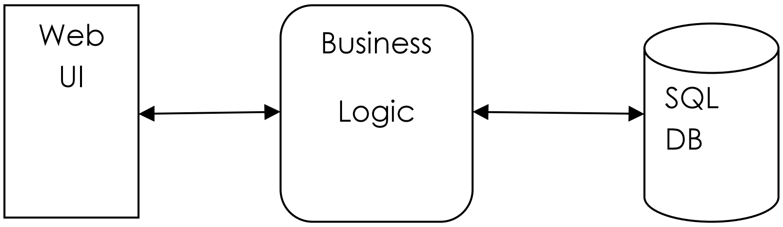


Fig. 2: components for a sample SUT application code name

2. Design Pattern

First introduced by the "gang of four" in 1994 [1], Design Pattern Concepts have received wide acceptance in software development community. In software engineering, a design pattern is a **general reusable solution** to a **commonly occurring problem** in software design (WIKIPEDIA). In this paper, the essence of Design Pattern contains the following three elements:

- The intent

States the purpose, problem domain, applicability, and limitation of the pattern (problem)

- Design constraints
  States the implementation requirements, adaptability, and extensibility of the pattern (reusability prerequisites)
- Supporting framework
  States the utility, helper functions, and inherited attributes, which helps pattern user to implement the working system (reusable objects and concepts)

I will use the above elements in describing the "Test Case Generation Pattern".

3. Test Case Generation and Execution Pattern (TCGEP)

Think of your automation as a baseline test suite to be used in conjunction with manual testing, rather than as a replacement for it. [2] To make the automation worthwhile, we need to understand what test cases really are, come up with an abstract and formal description, and form some basic test case generation design patterns. We will not cover specific test tool areas such as code coverage data collection.

3.1. Test case definition

3.1.1. Test Case Template

A test case in test automation terms can be defined as the following two elements:

1) *Applying a set of actions in form of function calls to SUT. The functions are called in a given order with their parameters precisely defined.*
2) *The result of the actions can be clearly predicated and measured by a set of other function calls to SUT.*

In reality, a test case resembles a state transition in a finite state machine. However, in describing this pattern, we use results from function calls to signal the pass or fail of a test case, rather than use state machine. This model makes it simple to model a test case and is also effective when combining with run time "*test state*".

Also note that the second element of test case is the "testability" requirement of a test case. In reality, it can be done as black box or white box depending on the nature of SUT and its testability level.

For our pattern to work, we have defined the following "test case template" class, based on which one or more actual test case can be formed:

A ***test case template*** is a class that contains a set of Action instances, and can be executed by calling the *DoTest* method of each action until one of the action signals failure or state change. The class definition as well as pseudo code for ***execute*** is shown below:

```
class MyTestCaseTemplate: BaseTestCaseTemplete
{
   List<Action> _myActionList;
   Void AddAction(Action action) {…}
   Virtual GetNextAction() (//default get test case sequentially ….)
   ….

   Void Execute()
   {
      While (true)
      {
              Action action;
            While (action = GetNextAction())
            {
                        Try
                        {
                                action.PrepareTestData();
                                action.Dotest();
                                action.Validate();
                        }
                        Catch(TestStateChange ex)
                        {
                                 // handle the state change
                                 StateChanged();
                                 break;
                        }
                        Catch(TestException ex)
                        {
                                 // handle error
                                 HandleError(ex);
                                 // decide to continue or quite
                                 ……………
                        }
            }
      }
……
}
```

Table 3.1: pseudo code for test case template

The above class is named as ***TestCaseTemplate***, instead of ***TestCase*** for a reason. To satisfy the 2nd elements of a test case definition, the above function *Execute* shall be predictable and can be measured for state change, pass, or failure accurately. This would not be possible given that each action has to prepare for its data at runtime. At the same time, ***GetNextAction*** may not return the same action as the last time the test was executed. Hence the "predicable" test case is only available after you play back the previously generated action/data sequence. This is why in the pseudo code, it's extremely important to log every action (***LogAction***) after running "***DoTest***".

The essence of the test case template idea can be summarized by the following points:

1) Avoid using hard coded test steps and test data early in test case generation by randomizing the actual steps and data as much as possible. For example, to test the user logon page of Happy-fish, your test steps can be either enter user name, enter password, and click OK, or enter password, enter user name, and click OK. On the other hand, you should randomly select a legal or illegal user name, rather than always use the same user name.
2) Separate test steps (actions) from test data and dynamically generate data at runtime
3) Log (serialize) the test case run and make it possible to replay test cases based on previously logged step sequence and data
4) Test case is confined by the SPEC (step and data requirement) but by no means static and inflexible. This makes test automation not only a perfect tool for regression but also introduces some explorative testing elements to test automation. For example, Happy-fish "SPEC" dictates that a user name must be longer than 4 characters and shorter than or equal to 10 charaters. This information is used to generate both positive and negative cases for "Add New User Page".

3.1.2. Action definition

A test action for a test case can be defined as the following three elements:

1) A specific function call made to SUT.
2) A group of parameters, with each parameter a set of values
3) Each parameter defines a *template* from which real value derives. The template is derived from the product SPEC.

The following pseudo code describes a test case action:

```
class TestCaseStep: BaseTestStep
{
    // Will call SUT API f1 with two parameters: an integer and a string
   public void DoTest() { ….}

   // Log this action
   public void LogAction();

   // Derive actual test data from data provider
   void PrepareTestData();

   // Validate the action
   void Validate();

   // Can the action be run now?
   public virtual bool ActionCanRun() {…}

   // first parameter needed for calling SUT function
   String _param1_template = "integer:0-100";

   // Second parameter needed for calling SUT function
   String _param2_template="string:charset:all illegal:~!@#$ length:1-256"
   …
}
```

Table 3.2: pseudo code for test action

From the above test action pseudo case, we learn that the actual action will be realized only when we run "***PrepareTestData***" because the parameter values are not hard-coded and only their templates are given. Regardless of the result of the test run, the action is logged and serialized to an XML file, in which the entire test case is serialized, makes it a concrete step within a concrete test case.

3.2. Test Case Generator

In the pseudo code listed in Table 3.1, ***GetNextAction*** as a virtual method, is also a variant factor in test case generation. It is difficult to predict what factor at run time will influence the decision of picking an action from a set of predefined actions. But to make it simple and practical, the pattern provides 3 implementation choices:

- Sequential action
  User is responsible to make sure the order of the action is the same as the order of the action list. This is suitable for simple test cases in which a set of well defined test steps can be used to simulate a usage scenario and the SUT behavior is predictable.
- Action/Data combined Permutation
  This method uses multiple-dimension permutation algorithm to calculate the action/data combination for each test case run. This is suitable for small steps/data value combination and guarantees to hit most if not all test cases automatically. It is also a great stress, long-haul test case generator.
- Randomly picking an action
  This method randomly picks up an action from an existing action list and randomly generates input data from data template.

No matter what choices you use, the derived class could override the decision making by calling ***ActionCanRun*** function to determine whether an action is relevant at the moment. This requires us to define the runtime decision making mechanism as a restriction. "Test State" is a simple interface we define for this purpose. It is a bridge between the framework and the SUT Test Drivers and helps the test action interact well with real world usage scenarios.

3.3. Test State

We have used a notion of test context or test state, to capture the SUT state, with respects to test applications. It is implemented as a simple string-object Hash-table. The idea is powerful even though it is extremely simple:

- The captured SUT states can be serialized to an XML file at any moment, making it easy to debug the test code as well as SUT code in case of test failure.
- The SUT states serve as an important mechanism for choosing the next test action. The following pseudo code from one of our test application demonstrates it:

```
Action GetNextAction()
{
   TargetList tgtLst =_testStates.GetNamedValue(“TargetList”) as TargetList;
   If (tgtLst.Count > 0)
   {
      return new ActionCreateVDS();
   }
      return new ActionCreateTarget();
   }
```

Table 3.3: Pseudo Test Step Generation Code

In the above pseudo-code, we choose “CreateVDS” action if there exists a “TargetList” object in the test state, while choosing “CreateTarget” action otherwise.

The Test State (TTS) can be viewed as an in-memory object model maintained by test automation which reflects the real SUT state from test automation’s point of view. Whenever, there is a conflict between TTS and SUT, it’s either a SUT product bug or a test bug and if it’s latter we need to improve our test automation.

3.4. Pattern Description

Name

Test Case Generation and Execution Pattern

Intent

- Reusability
- Standardize Basic Test Automation Components to guarantee the basic test coverage regardless of SUT’s problem domain
- Maximize the randomness in test case generation thus maximize the opportunity of finding real SUT bugs as early and as much as possible using the minimum efforts
- Possibility to auto-generate test program for the upper layer of test automation logic
- Expedite test automation development process

Constraints

- Data provider interface
- Test action interface
- Runtime state interface

Support Framework

- Standard data providers
- Standard error handling mechanism
- Basic permutation and combined permutation algorithm implementation

3.5. Pattern Framework Design

Although this paper mainly presents the concepts and ideas behind TCGEP pattern, we did provide an implementation and a framework for further expanding it. The following is the diagram which describes the basic OOP design architecture for TCGEP.

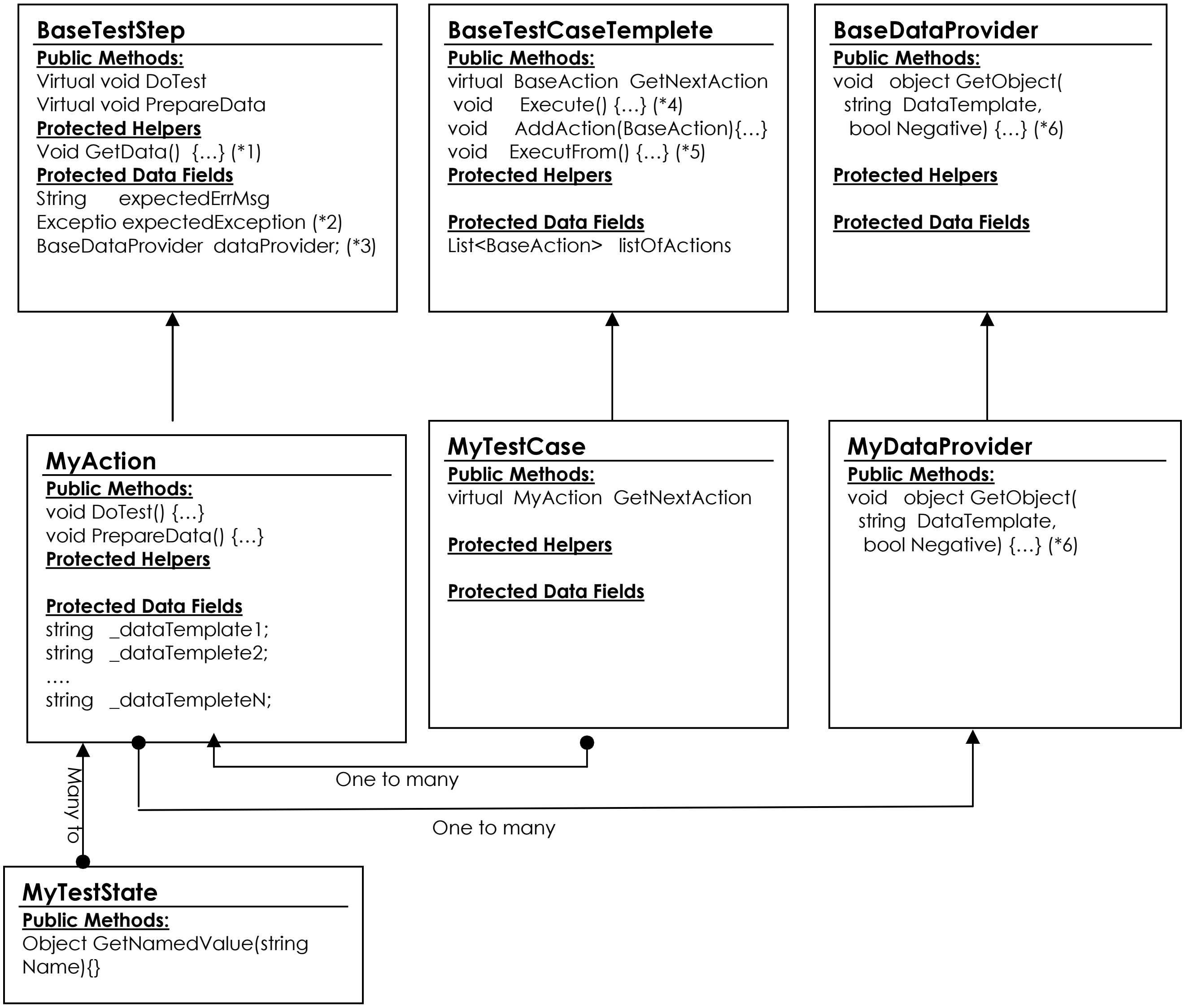
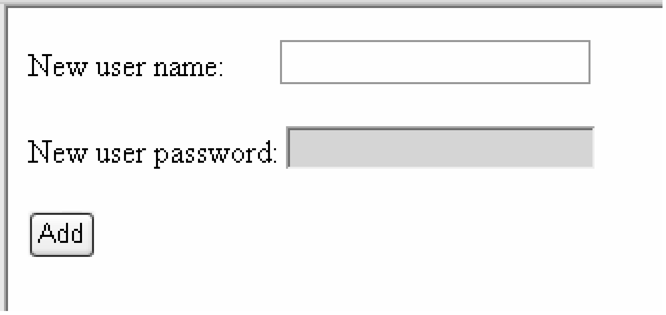


Fig. 3: TCGEP support framework class hierarchy

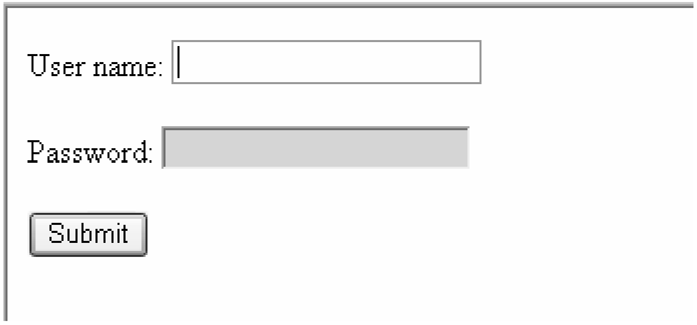
4. Demo Test Cases for Happy-fish Test Application

We will focus on our test efforts on admin page and user logon page. The following two UI pictures show admin page and logon page:

Admin page:



User logon page:



Although from UI perspective, the above two pages looks very similar, they are very different web pages in terms of functionality and testability:

- Admin page has much less restriction for name and password fields, as long as they meet the product SPEC regarding user name format and password requirement.
- User login page name and password fields can use random data only for fuzzing and security verification.
- User logon page functionality test data is provided by Admin page test and the data can only be retrieved from the "test state".

The following pseudo code illustrates the test steps for admin page:

```
Class AdminStep_PutName : BaseTestStep
{
    // test data template:
   string nameTemplate = "[char: any] [forbit:@%$&*()] [length:4-10]";
   void DoTest()
   {    // get the name from pattern and put it into the name field:
       String name = _myDataProvider.GetString(nameTemplate);
       // UI call
       SetData(_contrIDName, name);
   }
}
Class AdminStep_PutPassword: BaseTestStep
{   // test data template
  string pwTemplate = "[provider: cusPWPriv.dll]";
  // similar to the above DoTest function
}
```

```
Class AdminStep_Add: BaseTestStep
{
    void DoTest()
    {
        // call UI driver tool:
        clickOn("ctrlD_add");
    }
}
```

In the above pseudo case implementation, the Admin page has lots of freedom to randomly generate test data based on a well defined data pattern for user Name and uses a customer data provider for password. On the user logon page, the data source is more restricted and can come from the three sources:

1. From previously generated Admin test pages for "good" user credentials
2. From randomly generated data provider for random (mostly bad) data
3. Mixed data sources

The following lists the pseudo code for log on page (with a simplified one step):

```
class  UserLogon: BaseTestStep
{
    void DoTest()
    {
       // get name from two sources
       Name1 = _testState.GetAddedUserName();
       Name2 =  _myDataProvider.GetString("any");
       // get PW from two sources
      Pw1 = _testState.GetPWForUser(Name1);
      Pw2 =  _myDataProvider.GetString("any");
      // Randomly choose:
      ChooseNameAndPWCombo(out Name, out PW);

        // call UI driver tool:
        LogOn(Name, PW);
    }
}
```

In reality, we can have three test step classes form the above code to randomly choose one method. We mixed the three methods into one function just for demo purpose.

4.1. Test case generation

Some test automation applications only generate the obvious test cases due to lack of design and time. For example, the following would be the test case steps for testing the user log on page:

Step1: admin add a new user

Step2: the new user use his name and password to log on and he should be able to log on to "Account" page

Step3: log off from account page and back to user log on page

We use the default test case generator – "random test case generator", which form test cases by randomly picking a test step defined above. On the surface, this is less efficient than the "basic case", however, we have covered much more basic usage scenarios and discovered several Happy-fish bugs. The following user scenarios are some of the generated cases:

- Enter random user name/password before there is any user added by Admin user (a crashing bug found for this case).
- Enter a valid user to the DB; then from the user-logon page enter the right name but invalid password
- On the user logon page enter a invalid user name and a valid password of another user
- Enter a valid user and password and we succeeded in landing in user account page
- Add a new user while an old user is trying to log on
- Remove a user and the same user tries to log on again
- Enter invalid user name or password format (for both admin and user logon pages)

4.2. Conclusion

From the above example and pseudo code, we can observe the following important principals for using the test case generation design pattern:

- Focus on creating reusable actions with data template
- Use a standard test case generator for simple and effective test case generation by calling the defined action as test steps
- Use test state manager to keep track of SUT state and help decision making during runtime
- Log test state as well as test action details for regression

The practice has taught us some valuable lessons in implementing the design pattern. Nevertheless, we found that this design pattern is very effective in test coverage, stress, and explorative test case generation.

5. References

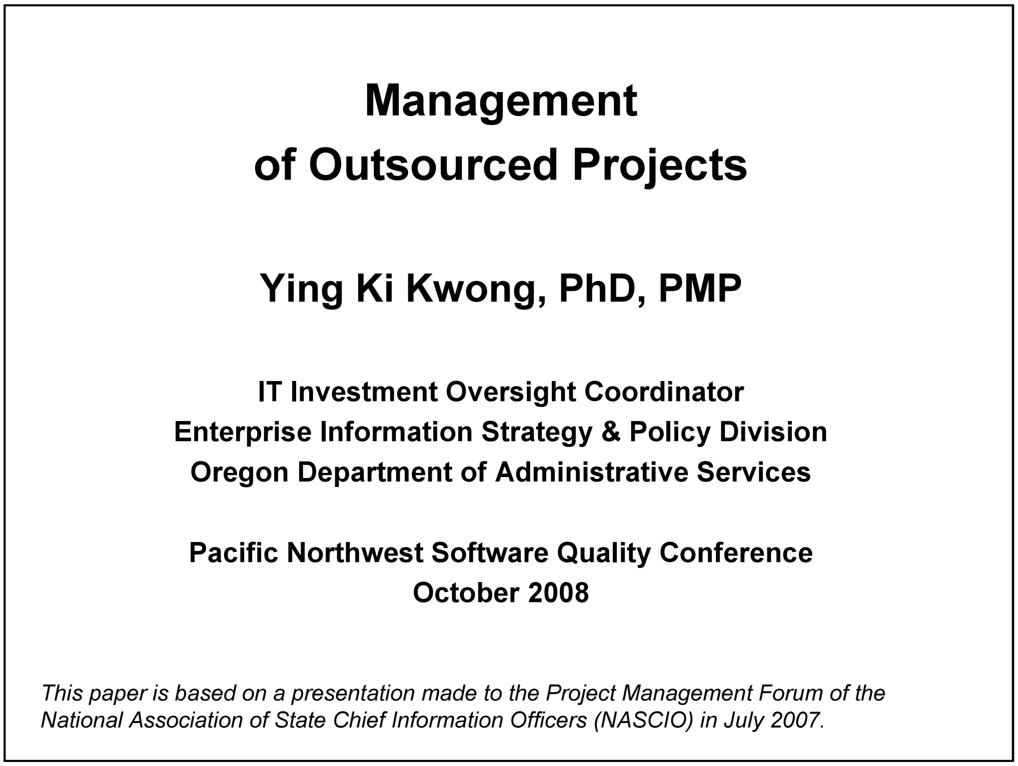
[1] Gamma, Helm, Johnson, Lissides. 1994. Design Patterns: Elements of Reusable Object-Oriented Software

[2] Bach, James. 1996. "Test Automation Snake Oil." *Windows Technical Journal* , (October): 40-44. http://www.satisfice.com/articles/test_automation_snake_oil.pdf

[3] Robert V. Binder "*Testing Object-Oriented Systems: Models, Patterns, and Tools* ".

6. About the Author

Lian Yang has been a developer, tester, and test lead at Microsoft for 13 years. He owns two patterns in test automation areas and helped Microsoft © shipped products such as Windows Media, Smartphone, Vista, and Windows Storage Server.



Management of Outsourced Projects

Ying Ki Kwong, PhD, PMP

IT Investment Oversight Coordinator
Enterprise Information Strategy & Policy Division
Oregon Department of Administrative Services

Pacific Northwest Software Quality Conference
October 2008

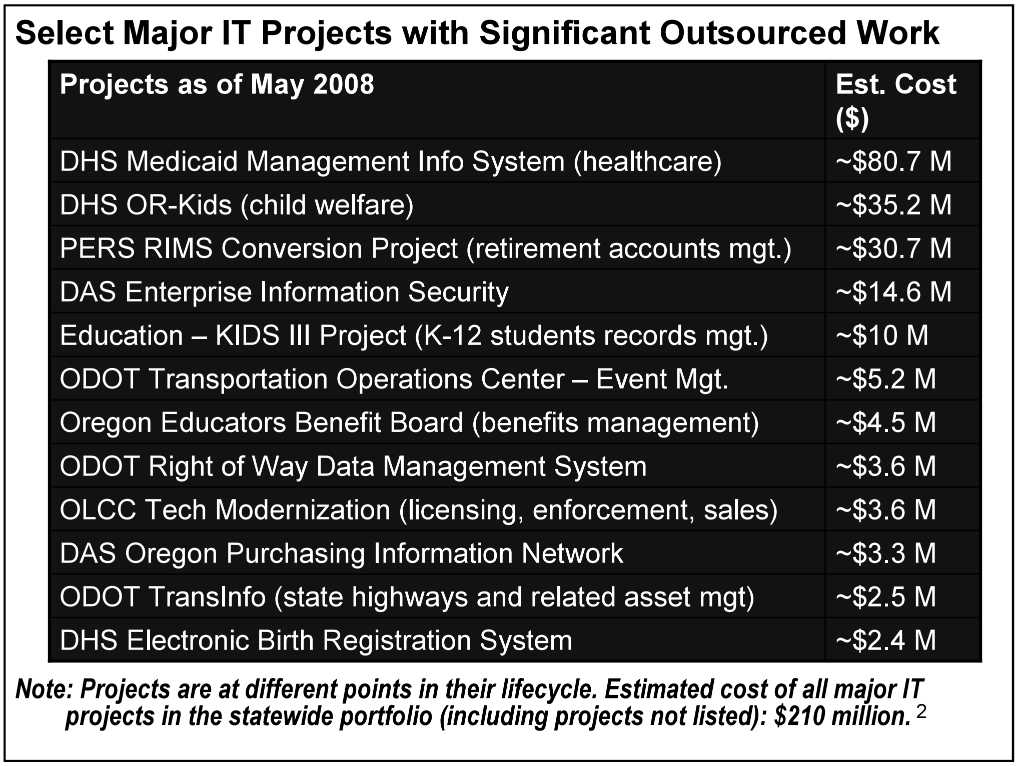
This paper is based on a presentation made to the Project Management Forum of the National Association of State Chief Information Officers (NASCIO) in July 2007.

The Enterprise Information Strategy & Policy Division (EISPD) is the office of the State Chief Information Officer (CIO), which is a division of the Oregon Department of Administrative Services. In addition to a variety of enterprise information technology (IT) programs, EISPD is responsible for oversight of all major IT projects for state agencies in the executive branch of Oregon state government.

The author of this presentation has been IT Investment Oversight Coordinator for the State of Oregon for about three years, currently reporting to the Deputy State CIO. In this role, he is the primary point of contact for lifecycle quality and risk management of major IT projects.

Before this role, the author was Project Office Manager of the Medicaid Management Information System Replacement Project — Oregon's largest IT project to date — during the Project's planning & procurement phase. Before joining the State of Oregon, he was CEO of a Hong Kong based internet B2B portal for online trading of commodities futures and metals. Prior to that, he was a program manager in the Video & Networking Division of Tektronix (now part of Thomson), responsible for worldwide applications and channels marketing for a line of video servers for broadcast television applications. In these roles, he was involved with the management of quality in software systems/applications, products, or software-enabled business processes.

In this presentation, the author will use examples from the State of Oregon to illustrate specific points. This presentation provides a useful perspective for outsourced IT projects in large enterprises and should be applicable to both the public and the private sectors unless otherwise stated in the notes of a slide.



Select Major IT Projects with Significant Outsourced Work

Projects as of May 2008	Est. Cost ($)
DHS Medicaid Management Info System (healthcare)	~$80.7 M
DHS OR-Kids (child welfare)	~$35.2 M
PERS RIMS Conversion Project (retirement accounts mgt.)	~$30.7 M
DAS Enterprise Information Security	~$14.6 M
Education – KIDS III Project (K-12 students records mgt.)	~$10 M
ODOT Transportation Operations Center – Event Mgt.	~$5.2 M
Oregon Educators Benefit Board (benefits management)	~$4.5 M
ODOT Right of Way Data Management System	~$3.6 M
OLCC Tech Modernization (licensing, enforcement, sales)	~$3.6 M
DAS Oregon Purchasing Information Network	~$3.3 M
ODOT TransInfo (state highways and related asset mgt)	~$2.5 M
DHS Electronic Birth Registration System	~$2.4 M

Note: Projects are at different points in their lifecycle. Estimated cost of all major IT projects in the statewide portfolio (including projects not listed): $210 million. [2]

This is a background slide regarding major IT projects in the State of Oregon…

At any one time over the last three years, the State of Oregon may have between 10 to 20 major IT projects. These projects have various characteristics, including but not limited to the following:

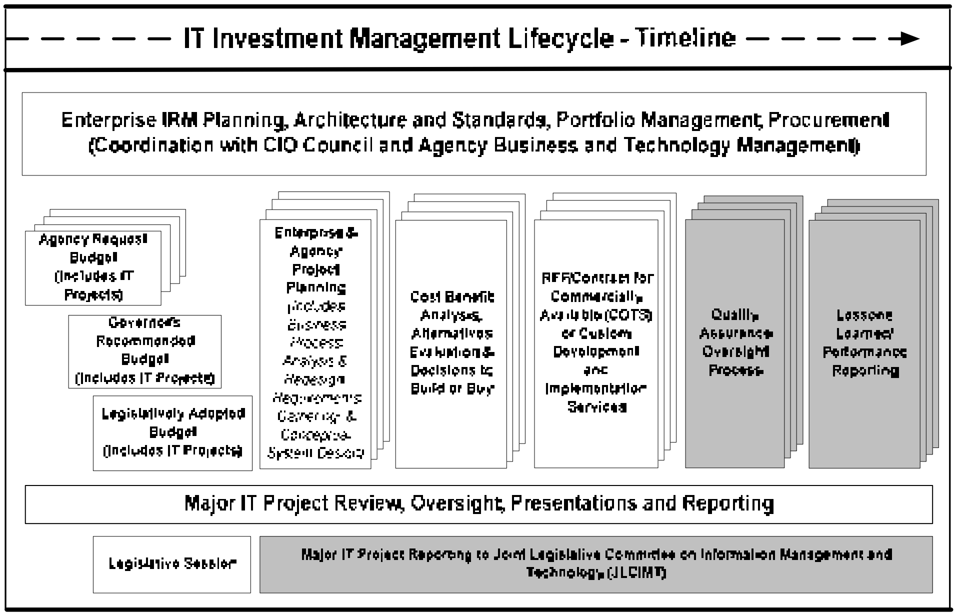
• They have budgets above US$1 million.

• They are mission critical and/or enable major change in the state agencies where the work are undertaken, both in terms of their operations, staff, and stakeholders. These stakeholders usually consist of internal and external stakeholders; both in and out of state government and other government jurisdictions.

• They affect citizens or the public in important ways.

The State's major IT projects portfolio has a total value of about $210 million in May 2008, as seen in this chart. Most major IT projects listed are planned, designed, developed, and implemented by private contractors working closely with State personnel. As such, most technical work is outsourced to contractors.

Oregon IT Investment Management Lifecycle



IT Investment Management Lifecycle - Timeline

Enterprise IRM Planning, Architecture and Standards, Portfolio Management, Procurement (Coordination with CIO Council and Agency Business and Technology Management)

Agency Request Budget (Includes IT Projects)

Governor's Recommended Budget (Includes IT Projects)

Legislatively Adopted Budget (Includes IT Projects)

Enterprise & Agency Project Planning (includes Business Process Analysis & Redesign Requirements Gathering & Conceptual System Design

Cost Benefit Analysis, Alternatives Evaluation & Decisions to Build or Buy

RFP/Contract for Commercially Available (COTS) or Custom Development and Implementation Services

Quality Assurance Oversight Process

Lessons Learned/ Performance Reporting

Major IT Project Review, Oversight, Presentations and Reporting

Legislative Session

Major IT Project Reporting to Joint Legislative Committee on Information Management and Technology (JLCIMT)

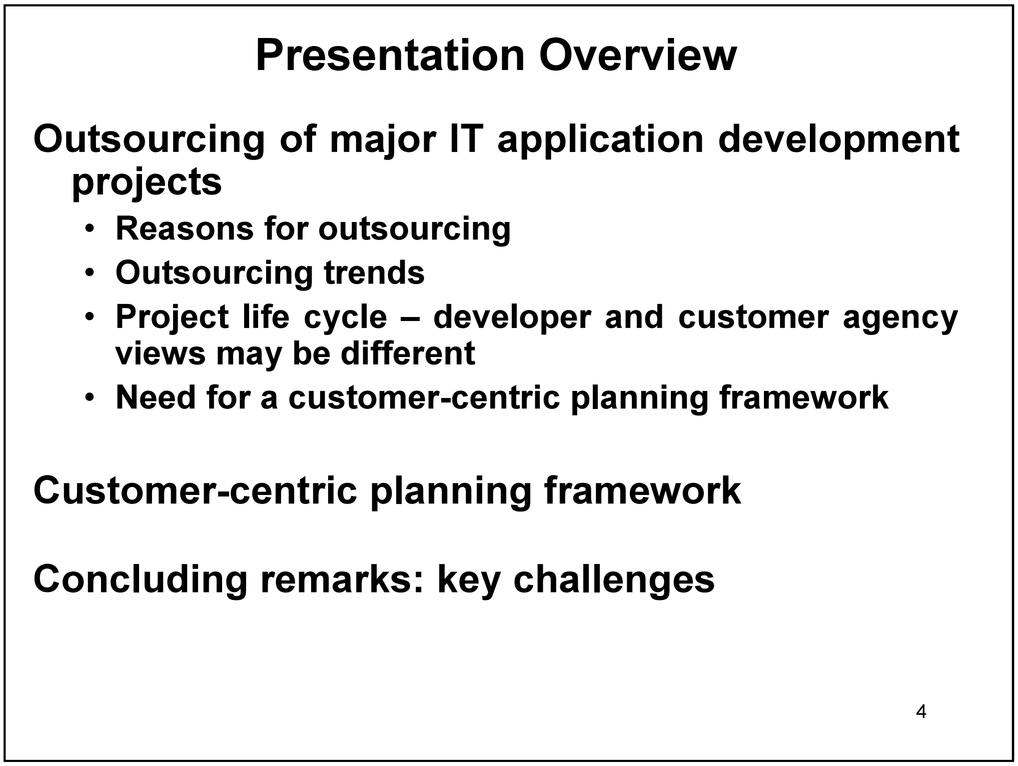
3

This is another background slide on major IT projects in the State of Oregon…

The State of Oregon views major IT projects as investments with a lifecycle. This lifecycle typically begins as an agency concept that leverages IT to improve or re-engineer business process, increase capacity to meet stakeholder needs, or improve operational efficiency. Agency concepts drive the development of the budget process, which occurs once every two years. A project's budget becomes part of an agency's budget, which is then incorporated into the Governor's recommended budget and become legislatively adopted during a biennial legislative session. Off session requests are authorized by the Legislature through the Emergency Board process.

With the legislatively adopted budget in place, a state agency must prepare an Information Resource Request (IRR). The IRR is the vehicle for State CIO approval of a project and is supported by a detail business case, which analyzes the relative costs, benefits, and risk of available solution alternatives. An approved IRR is required before procurement of hardware, software, and professional services.

Because of the importance of a major IT project to a state agency and its stakeholders, quality assurance is an important aspect of every major IT project. Statewide quality processes exist to assure product quality, process quality, and management accountability. The staff of the State CIO participates in many facets of quality planning and has broad oversight authority traceable to state statutes, administrative rules, and policies. As will be discussed, the State use independent QA contractors to provide independent assessment of major IT projects. The findings of these assessments are periodically shared with decision makers in both the executive and the legislative branches of Oregon state government.



Presentation Overview

Outsourcing of major IT application development projects

- **Reasons for outsourcing**
- **Outsourcing trends**
- **Project life cycle – developer and customer agency views may be different**
- **Need for a customer-centric planning framework**

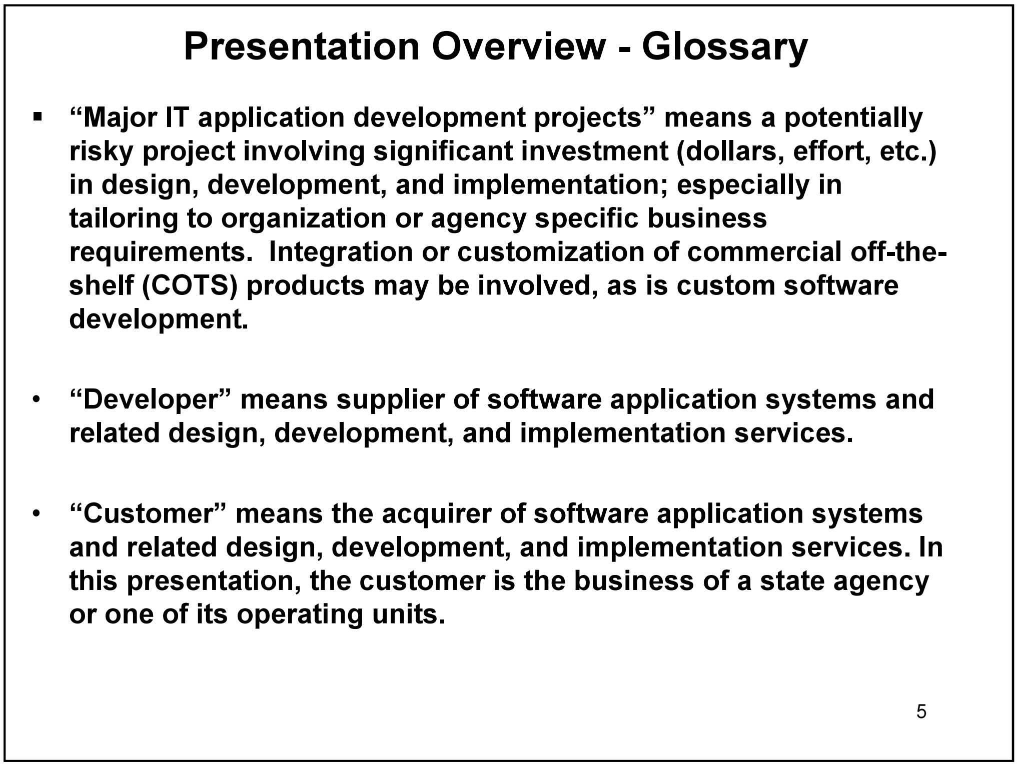
Customer-centric planning framework

Concluding remarks: key challenges

4

This presentation will begin by discussing why organizations outsource major IT projects. We will put forth the perspective that the objectives of an acquiring enterprise are *not* the same as those of the developer (contractor). As such, the thinking with respect to project management and quality management are also different, in general.

This presentation describes a "customer-centric" framework to plan and execute outsourced IT project. In the context of this presentation, "customer" is defined as the enterprise acquiring the software system or application in question.



Presentation Overview - Glossary

- **“Major IT application development projects” means a potentially risky project involving significant investment (dollars, effort, etc.) in design, development, and implementation; especially in tailoring to organization or agency specific business requirements. Integration or customization of commercial off-the-shelf (COTS) products may be involved, as is custom software development.**

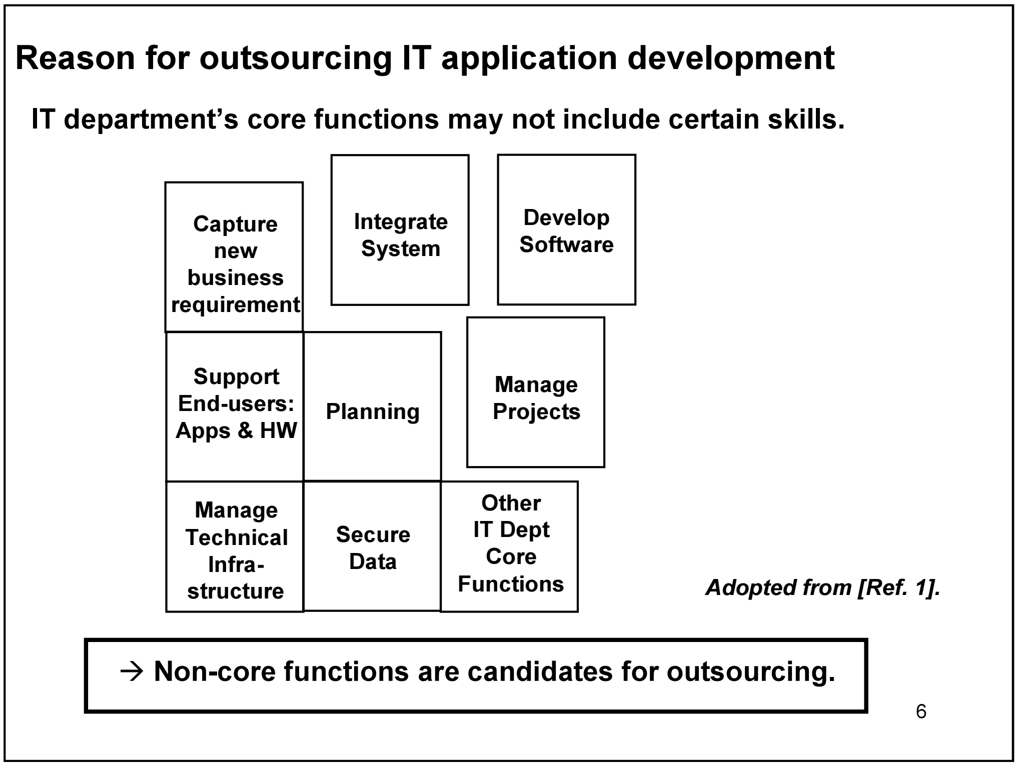
- **“Developer” means supplier of software application systems and related design, development, and implementation services.**

- **“Customer” means the acquirer of software application systems and related design, development, and implementation services. In this presentation, the customer is the business of a state agency or one of its operating units.**

5

This slide defines terms that will be used throughout this presentation. In addition to the above definitions, please note the following:

1. As this presentation discusses outsourced application development projects, the term “developer,” “development contractor,” and “contractor” will be used interchangeably.
2. In this presentation, we use the term “customer” to refer to the enterprise acquiring the software system or application in question, which is the customer of the developer. A large enterprise may have an IT department that is primarily responsible for applications development and managing IT contractors. Such an IT department would have internal stakeholders that may be referred to as its “internal customers,” or simply “customers.” To avoid confusion in this presentation, we will only use the term “customer” in the sense defined in the slide above.



Reason for outsourcing IT application development

IT department's core functions may not include certain skills.

Capture new business requirement

Integrate System

Develop Software

Support End-users: Apps & HW

Planning

Manage Projects

Manage Technical Infra-structure

Secure Data

Other IT Dept Core Functions

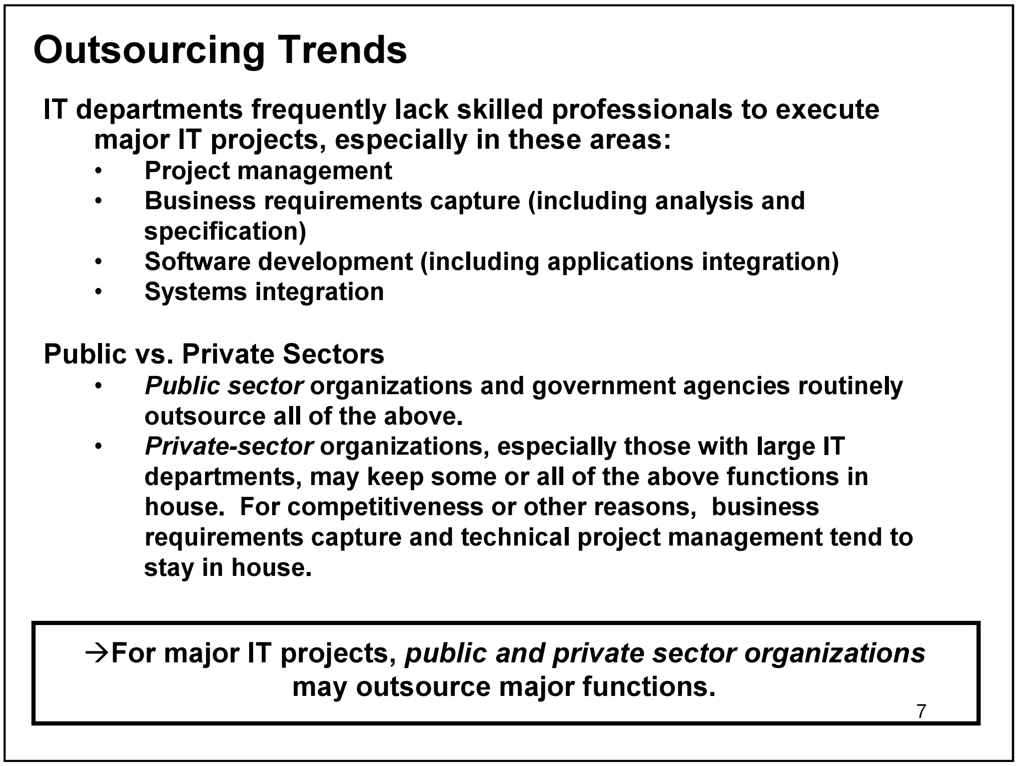
Adopted from [Ref. 1].

→ Non-core functions are candidates for outsourcing.

6

Depending on the enterprise, what is considered "core" functions or core competency may be different from enterprise to enterprise. As an example, companies such as Nike does not consider manufacturing to be core functions and use contract manufacturers extensively to fulfill its manufacturing needs.

For IT, enterprises tend to view support (for hardware, network, applications) and information security as core. Increasingly, enterprises view project management, software development, and system integration as non-core. As such, the design, development, and implementation of major IT projects are increasingly outsourced, with in-house development by internal IT staff becomes correspondingly less common.



Outsourcing Trends

IT departments frequently lack skilled professionals to execute major IT projects, especially in these areas:

- **Project management**
- **Business requirements capture (including analysis and specification)**
- **Software development (including applications integration)**
- **Systems integration**

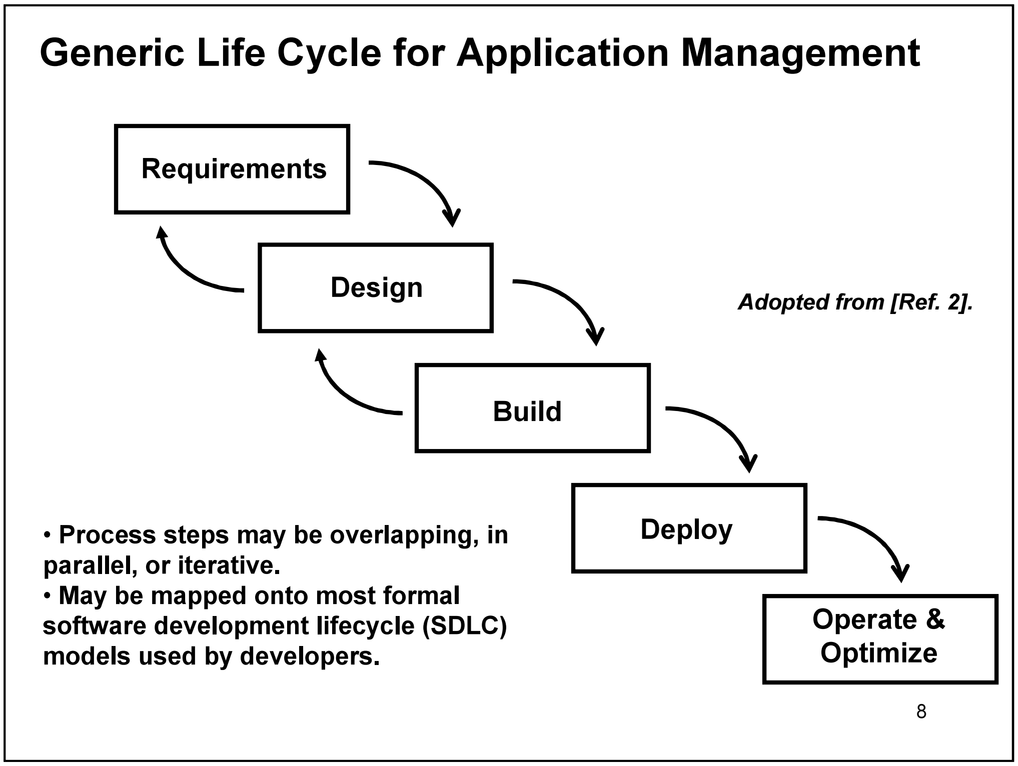
Public vs. Private Sectors

- ***Public sector* organizations and government agencies routinely outsource all of the above.**
- ***Private-sector* organizations, especially those with large IT departments, may keep some or all of the above functions in house. For competitiveness or other reasons, business requirements capture and technical project management tend to stay in house.**

→For major IT projects, *public and private sector organizations* may outsource major functions.

Outsourcing of work by the IT department of an enterprise is usually driven by limitation of internal resources, the desire to focus on core business functions, and the desire to reduce cost.

The view as to whether requirements definition should be considered core function of an enterprise varies. Private enterprises usually views this (especially the definition of business functional requirements) as part of its core competency, because business requirements are closely related to an enterprise’s competitive differentiation in the marketplace. Increasingly, in both private and public sector enterprises, even the capture/analysis of business requirements increasingly involves contract-based personnel. Contract-based personnel may include domain experts and facilitators in “joint application requirements” sessions or equivalent activities in iterative or Agile system development lifecycle models.



Generic Life Cycle for Application Management

Requirements

Design

Build

Deploy

Operate & Optimize

Adopted from [Ref. 2].

- Process steps may be overlapping, in parallel, or iterative.
- May be mapped onto most formal software development lifecycle (SDLC) models used by developers.

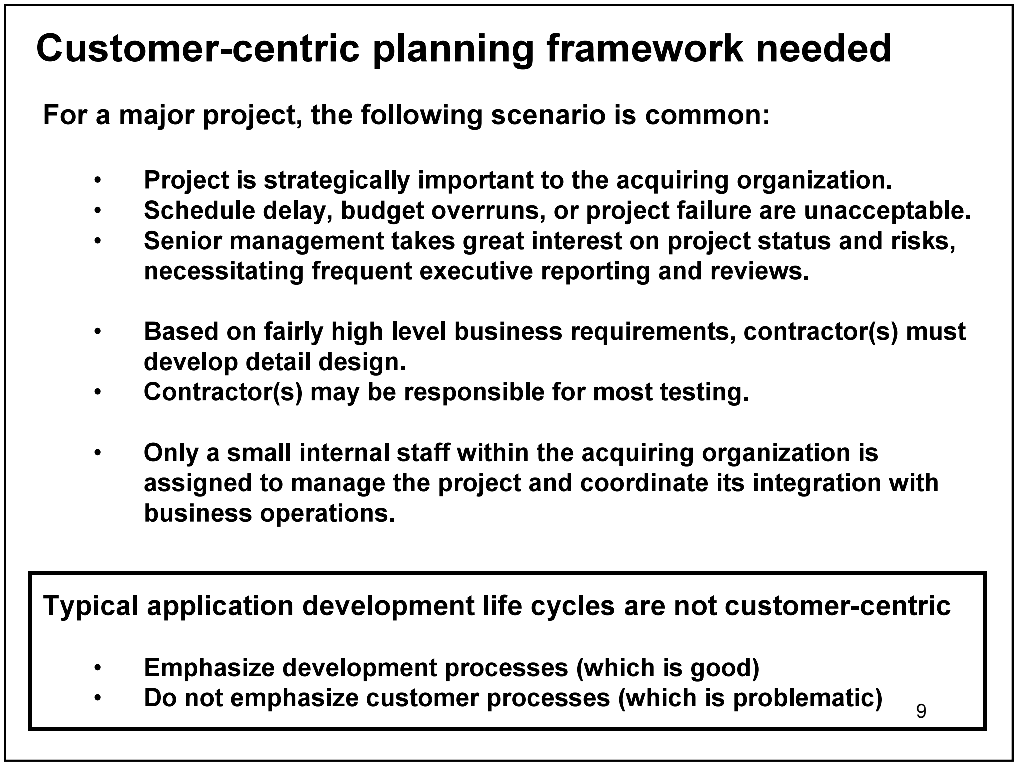
8

Effective ways to outsource IT projects are related to an organization's model for applications management. This generic lifecycle model is based on ITIL and can be mapped to all common SDLC models, as we will discussed later.

Especially important to the quality of an enterprise's effort to outsource major IT projects are:

• the process for capturing, communicating, or documenting requirements. (By necessity, this needs to include considerations for change control of scope, schedule, and budget.)

• the process for quality and risk management during Design, Build, and Deploy. (By necessity, this needs to include considerations for status tracking/reporting, customer reviews at major milestones, acceptance testing of iterations or subsystems, and processes for acceptance and/or payment for major releases.)



Customer-centric planning framework needed

For a major project, the following scenario is common:

- **Project is strategically important to the acquiring organization.**
- **Schedule delay, budget overruns, or project failure are unacceptable.**
- **Senior management takes great interest on project status and risks, necessitating frequent executive reporting and reviews.**

- **Based on fairly high level business requirements, contractor(s) must develop detail design.**
- **Contractor(s) may be responsible for most testing.**

- **Only a small internal staff within the acquiring organization is assigned to manage the project and coordinate its integration with business operations.**

Typical application development life cycles are not customer-centric

- **Emphasize development processes (which is good)**
- **Do not emphasize customer processes (which is problematic)**

9

An outsourced IT project is usually tied to strategic improvements that an enterprise considers important – sometime critically important. As such, project failure is usually unacceptable, and senior management takes particular interest in the project's status, performance, and management.

For many projects, the requirements available to the contractor at the start of the contract may be very high level or conceptual. As a result, the contractor frequently plays a significant role in defining detail requirements, in performing data conversion, and in testing.

From the perspective of the acquiring enterprise, typically a small staff is responsible for managing the project, the overall implementation of a system, work products acceptance, and integration with business operations. This situation is made more challenging by the fact that many system development lifecycle (SDLC) models emphasize development processes and do not adequately emphasize the customer's own project life cycle processes for requirements, procurement, contract administration, quality and risk, and organizational change.

This presentation will overview a framework that emphasizes these topics from the perspective of the enterprise outsourcing major IT projects to development contractors.

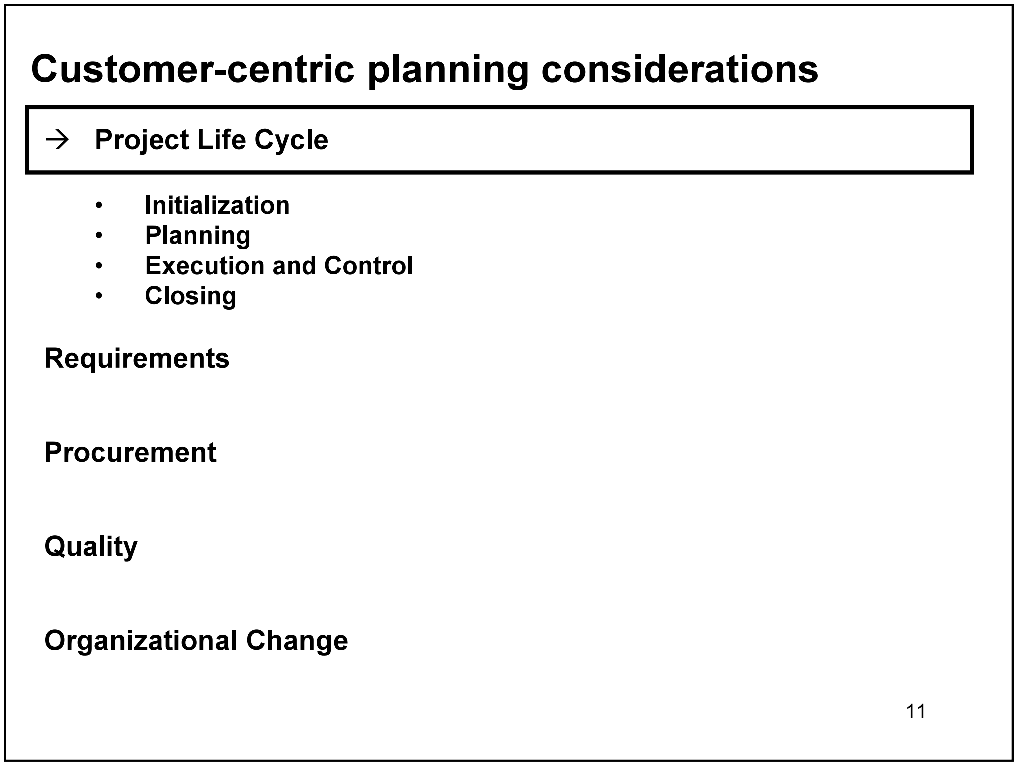
Customer-centric planning considerations

- Project Life Cycle
- Requirements
- Procurement
- Quality
- Organizational Change

10

The planning framework being presented here is from the point of view of the customer of the development contractor, i.e. the acquiring enterprise.

We start with reviewing aspect of a project’s lifecycle, followed by the management of requirements, procurement, quality, and organizational change.



Customer-centric planning considerations

→ **Project Life Cycle**

- **Initialization**
- **Planning**
- **Execution and Control**
- **Closing**

Requirements

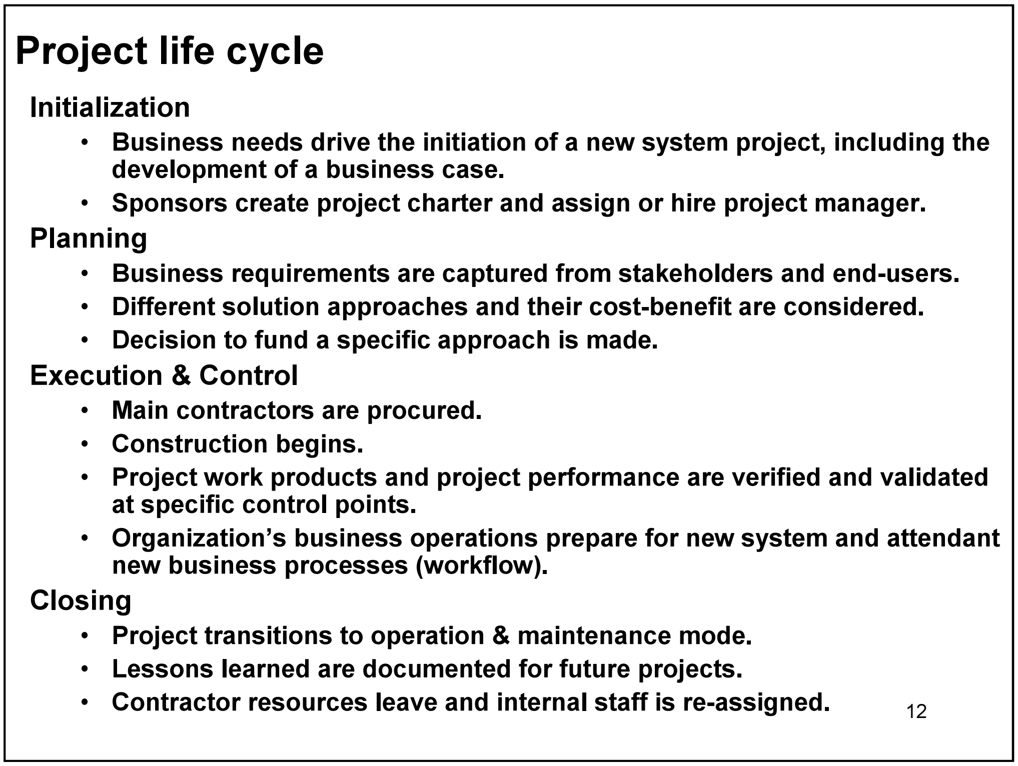
Procurement

Quality

Organizational Change

11

We will start with a discussion of the Project Life Cycle.



Project life cycle

Initialization

- **Business needs drive the initiation of a new system project, including the development of a business case.**
- **Sponsors create project charter and assign or hire project manager.**

Planning

- **Business requirements are captured from stakeholders and end-users.**
- **Different solution approaches and their cost-benefit are considered.**
- **Decision to fund a specific approach is made.**

Execution & Control

- **Main contractors are procured.**
- **Construction begins.**
- **Project work products and project performance are verified and validated at specific control points.**
- **Organization's business operations prepare for new system and attendant new business processes (workflow).**

Closing

- **Project transitions to operation & maintenance mode.**
- **Lessons learned are documented for future projects.**
- **Contractor resources leave and internal staff is re-assigned.**

12

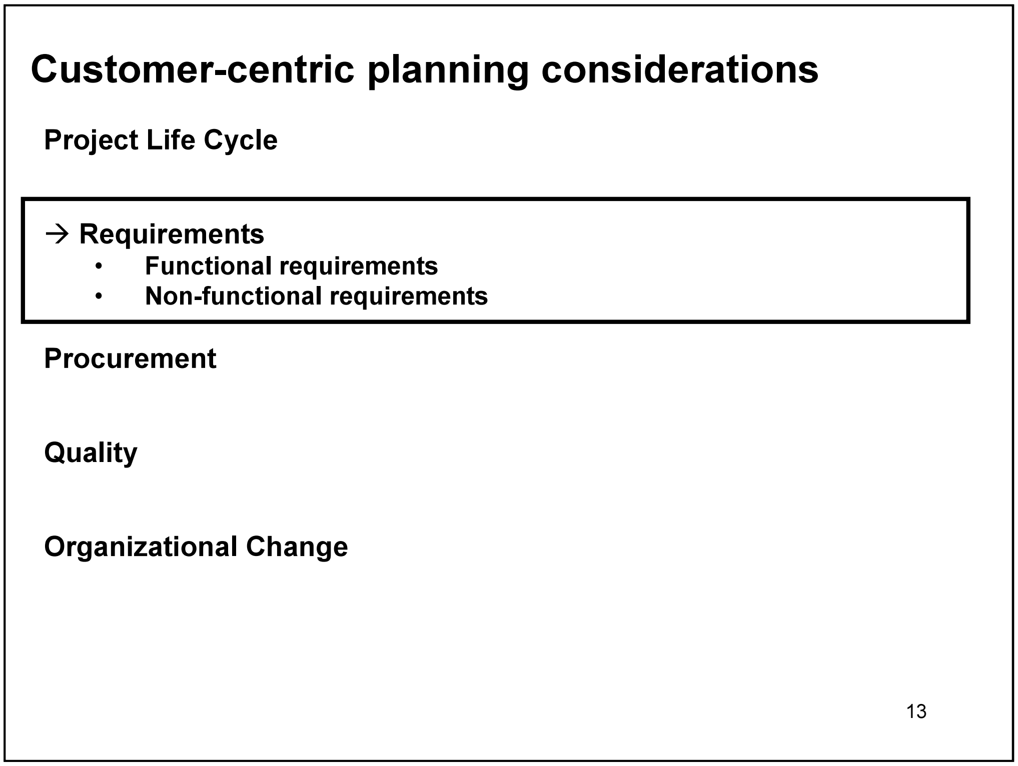
A significant degree of formality is typically required in the management of major IT projects in large enterprises. This is so for two main reasons.

First, a major IT project is typically complex, both in business and technical terms. The required resources and personnel across various functions of an enterprise must be coordinated. This is so at different points of the project; notably at times of requirements definition, user acceptance, implementation, and integration and operationalization of the production system into actual business operations. Key stakeholders from across the enterprise must also be involved in on-going project governance, requirements change control, and quality and risk management. These efforts need to be coordinated, and formal project management is a useful tool for this coordination and communication.

Second, a major IT project is important to the future of an enterprise. Stakeholders from both within and outside the enterprise care and need to know about the status of the project. As such, a formal approach to project management is typically the foundation for communication with diverse stakeholders (including senior management) on project status and associated project risks and issues.

Like many large organizations, the project management approach that is the standard for the State of Oregon is the Project Management Body of Knowledge (PMBOK) Guide, third edition, as published by the Project Management Institute. In the PMBOK view, a project consists of distinct phases, as outlined in this slide.

The PMBOK approach emphasizes processes, planning, and performance measurements against baseline plans; e.g. an integrated project plan that baselines project scope (requirements), schedule, and budget, as well as supporting plans for the management of quality, risk, procurement, human resources, and communication. The PMBOK can be made consistent with iterative and Agile SDLCs, a point that we will return to later.



Customer-centric planning considerations

Project Life Cycle

→ Requirements

- Functional requirements
- Non-functional requirements

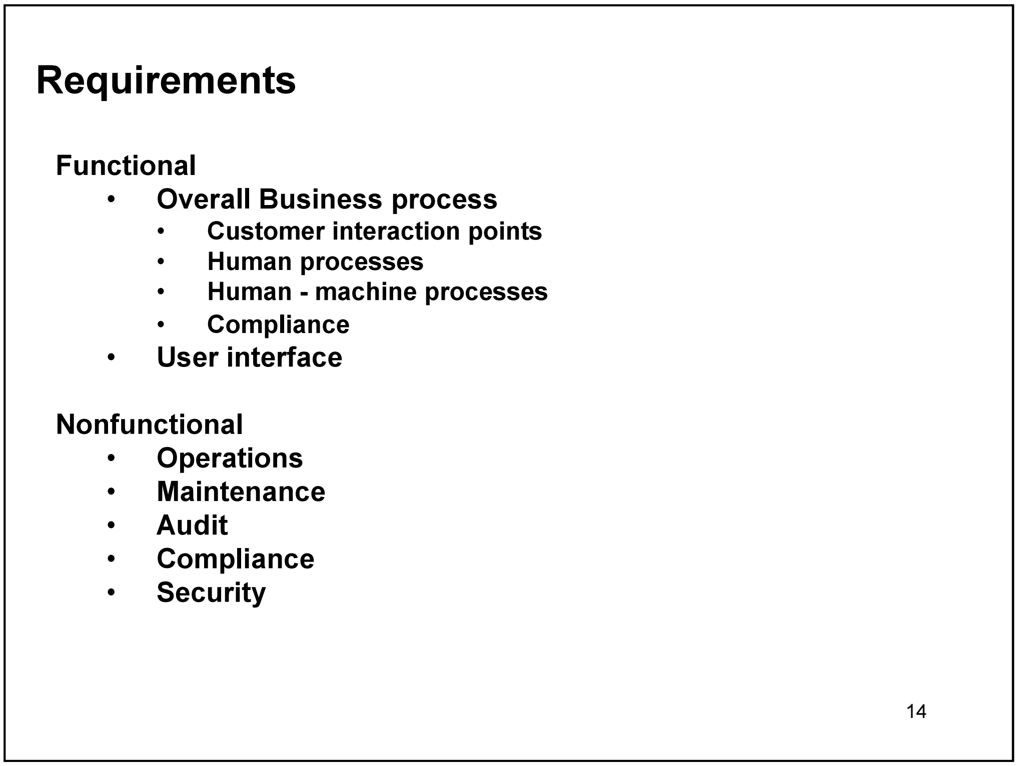
Procurement

Quality

Organizational Change

13

We now discuss Requirements.



Requirements

Functional

- **Overall Business process**
  - **Customer interaction points**
  - **Human processes**
  - **Human - machine processes**
  - **Compliance**
- **User interface**

Nonfunctional

- **Operations**
- **Maintenance**
- **Audit**
- **Compliance**
- **Security**

14

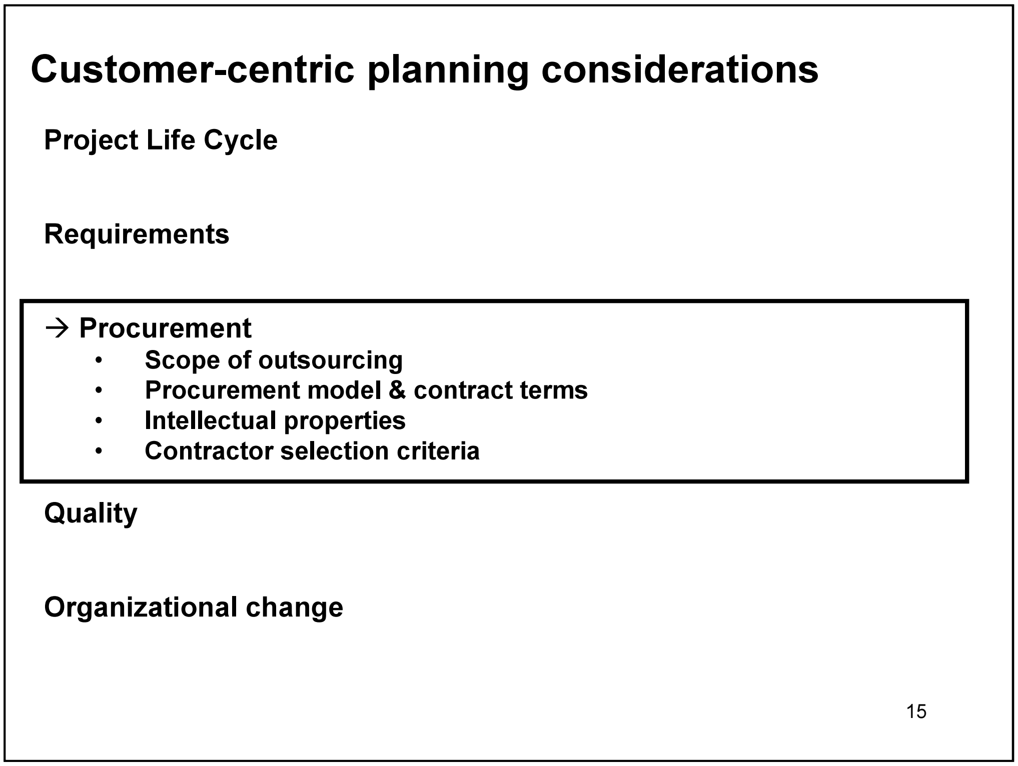
At the highest level, requirements can be described in terms of business and non-business requirements.

Business requirements are also known as functional requirements. Typically, an enterprise acquires a system to fulfill specific business functions. The management and the users of an enterprise system, especially non-technical users, typically do not care about the underlying technology.

Non-business requirements are also known as non-functional requirements, which include requirements for technology platforms and for on-going support and maintenance. (In the public sector, procurement rules may require government agencies to *not* favor specific technology platforms, especially for specific brand named products, unless there is a business need.)

Large enterprises must increasingly pay attention to regulatory compliance. In North America, HIPAA (for the healthcare industry in both the public and private sectors) and Sarbane-Oxley (for publicly traded companies in the private sector) have focused the need of enterprises to build IT systems and associated business processes that are secured, compliant with applicable regulations, and support internal and external audits. Although depicted in this slides as non-functional requirements, many enterprises begin to view information security, regulatory compliance, and auditability as business requirements.

Managing requirements well is both a science and an art. It is central to high quality software application, but it is also an important factor for high-quality procurement and contract administration of outsourced IT projects.



Customer-centric planning considerations

Project Life Cycle

Requirements

→ Procurement

- **Scope of outsourcing**
- **Procurement model & contract terms**
- **Intellectual properties**
- **Contractor selection criteria**

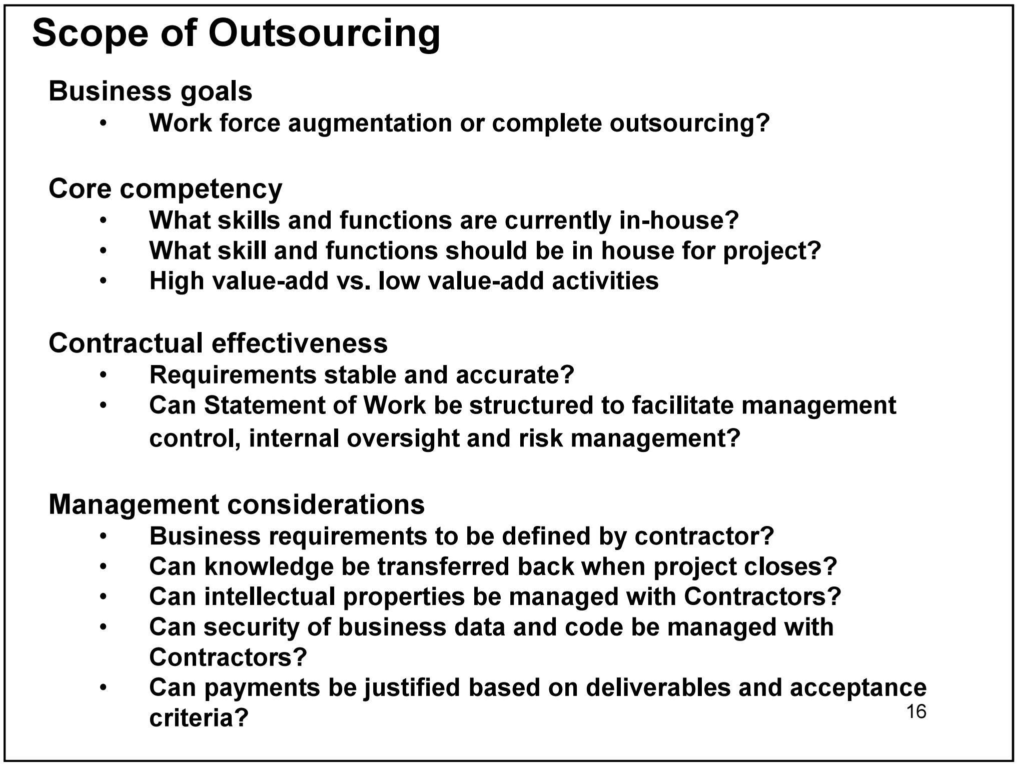
Quality

Organizational change

15

We now discuss procurement and related contract administration considerations.

Even experienced project managers sometime view procurement as "something for the legal department or the purchasing departments." In the author's opinion, this is incorrect.



Scope of Outsourcing

Business goals

- **Work force augmentation or complete outsourcing?**

Core competency

- **What skills and functions are currently in-house?**
- **What skill and functions should be in house for project?**
- **High value-add vs. low value-add activities**

Contractual effectiveness

- **Requirements stable and accurate?**
- **Can Statement of Work be structured to facilitate management control, internal oversight and risk management?**

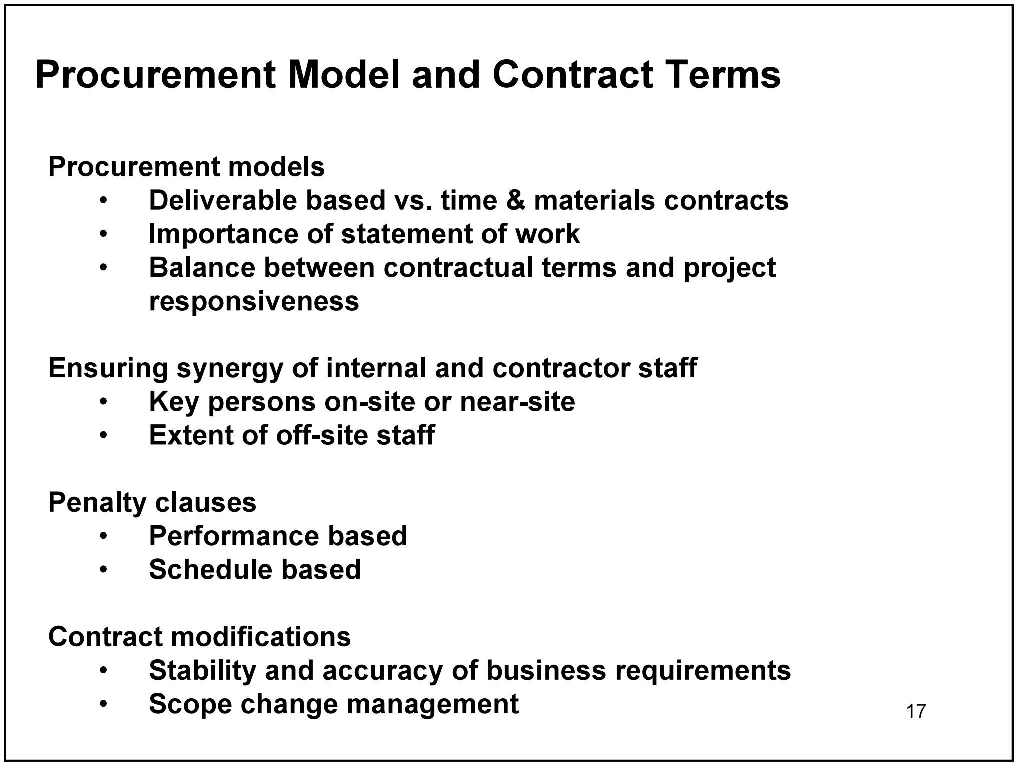
Management considerations

- **Business requirements to be defined by contractor?**
- **Can knowledge be transferred back when project closes?**
- **Can intellectual properties be managed with Contractors?**
- **Can security of business data and code be managed with Contractors?**
- **Can payments be justified based on deliverables and acceptance criteria?**

16

When outsourcing, it is important for an enterprise to keep in mind the business goals and objectives. Major projects frequently require entire projects to be outsourced for two reasons. First, the internal organization simply lacks resources. Second, a single point of responsibilities may enable more effective transfer of risk (technical, schedule, and possibly even budgetary) from the acquiring enterprise to the developer.

Outsourcing usually involves some sort of contractual arrangement between the developer and its customer. Key questions that should be discussed among stakeholder of a major IT project (ideally before the start of the procurement process) are listed in this slide.



Procurement Model and Contract Terms

Procurement models

- **Deliverable based vs. time & materials contracts**
- **Importance of statement of work**
- **Balance between contractual terms and project responsiveness**

Ensuring synergy of internal and contractor staff

- **Key persons on-site or near-site**
- **Extent of off-site staff**

Penalty clauses

- **Performance based**
- **Schedule based**

Contract modifications

- **Stability and accuracy of business requirements**
- **Scope change management**

17

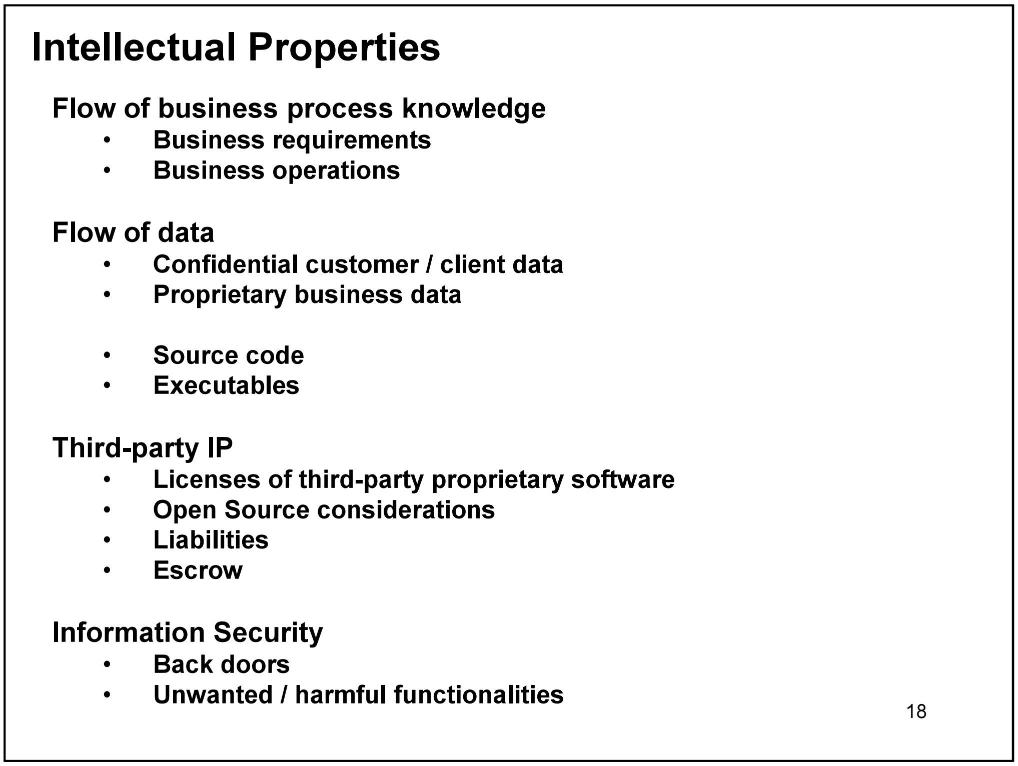
From the perspective of the acquiring enterprise, project success is at least partly tied to the effectiveness of the contract. In this respect, it is important for the acquiring enterprise to look at four things: procurement models, the roles & responsibilities of the contractor's staff vs. the customer's staff, penalty clauses, and the degree of contract modifications anticipated during the project's lifecycle.

Procurement models primarily refer to whether the contract will be deliverables based or time-and-materials based. If the requirements are well understood, stable, and can be well documented, a deliverables based contract is preferred. Many enterprises (especially private ones) may feel that time-and-materials contracts are more easily administered and requires less upfront work associated with statement of work (SOW) development. However, this is so only if the acquiring enterprise is willing to accept the risk associated with contractor non-performance and potentially flawed hours estimates.

The roles & responsibilities of the contractor staff vs. the staff of the acquiring enterprise need to well understood by the developer and its customer. Part of this understanding includes an agreement on the level of on-site staff requirements, especially contractor key personnel such as project managers and subject matter experts.

Terms and conditions in the contract that awards performance (such as early completion) could be a useful incentive for the contractor; as are clauses that delay, retain, or otherwise reduce payment due to non-performance (such as late completion).

A high quality SOW that balances specificity (of tasks and deliverables) with the need for the contractor/customer to dynamically respond to project unknowns is important. For requirements that may not be stable due to rapidly changing business conditions or the lack of thoroughly analyzed requirements, it may be necessary for the customer to budget extra funds for on-demand work by the contractor (possibly through task orders) or through contract amendments.



Intellectual Properties

Flow of business process knowledge

- **Business requirements**
- **Business operations**

Flow of data

- **Confidential customer / client data**
- **Proprietary business data**

- **Source code**
- **Executables**

Third-party IP

- **Licenses of third-party proprietary software**
- **Open Source considerations**
- **Liabilities**
- **Escrow**

Information Security

- **Back doors**
- **Unwanted / harmful functionalities**

18

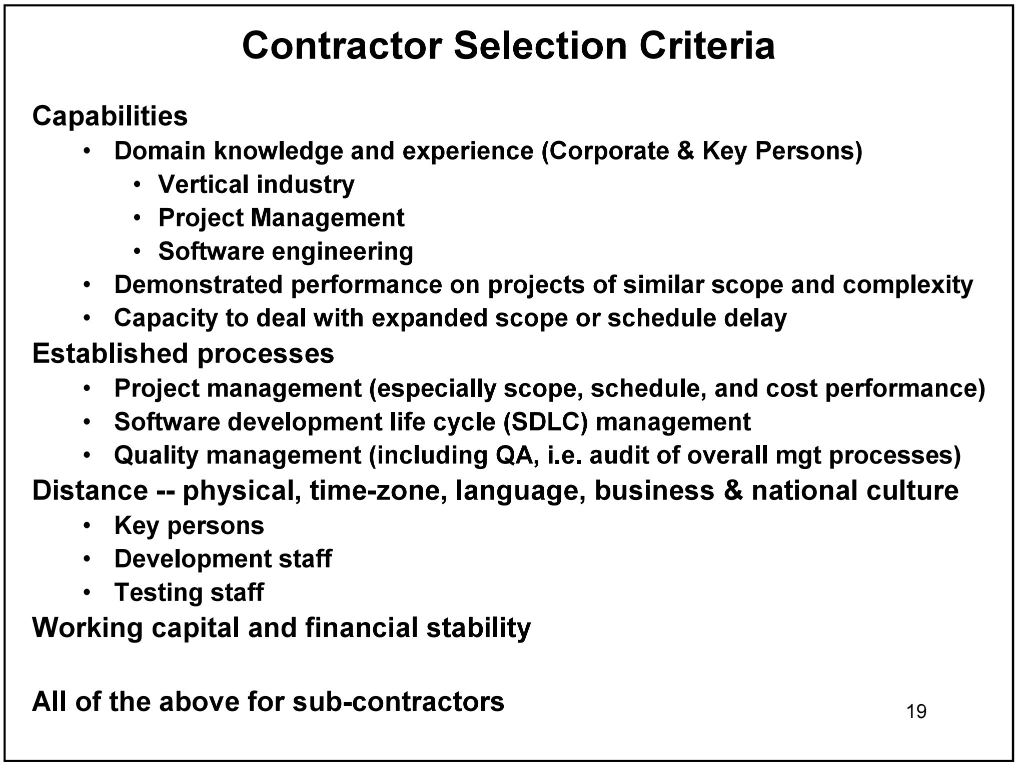
The management of intellectual properties (IP) is an especially tricky issue for offshoring (outsourcing to offshore locations), especially to countries where IP protection may be weak. However, IP in the context of outsourcing does not refer simply to unauthorized access, use, or distribution of source codes or executables.

The flow of business process knowledge is a key IP concern for the acquiring enterprise, especially when the contractor's subject matter experts (SME) are primarily responsible for defining To Be (future) business process and associated business rules and requirements. How to effectively transfer this IP from developer back to the customer is not always clear, even when the contract SOW makes provision for extensive training and/or warranty.

The potential flow of confidential or proprietary data is also a major concern. Today, confidentiality agreements are routinely expected. Many acquiring enterprises also require on-site contractor personnel to be subjected to background checks by law enforcement authorities. The security of testing or training environments is also important, not only to secure customer data but also to secure source codes and executables.

Contract language must increasingly cover reuse of third-party codes and to assure proper pass-through of proprietary software licenses. (With Open Source, the concern may be safeguards against unintentional pass-through of general licensing agreements that are not desirable from the customer's perspective.) The goal for the customer is legal protection (indemnification) against liabilities associated with third-party IP.

Large enterprises are also concerned with access to source codes in the event the contractor goes out of business. To mitigate this risk, the contract may include an escrow requirement for source codes.



Contractor Selection Criteria

Capabilities

- **Domain knowledge and experience (Corporate & Key Persons)**
  - **Vertical industry**
  - **Project Management**
  - **Software engineering**
- **Demonstrated performance on projects of similar scope and complexity**
- **Capacity to deal with expanded scope or schedule delay**

Established processes

- **Project management (especially scope, schedule, and cost performance)**
- **Software development life cycle (SDLC) management**
- **Quality management (including QA, i.e. audit of overall mgt processes)**

Distance -- physical, time-zone, language, business & national culture

- **Key persons**
- **Development staff**
- **Testing staff**

Working capital and financial stability

All of the above for sub-contractors

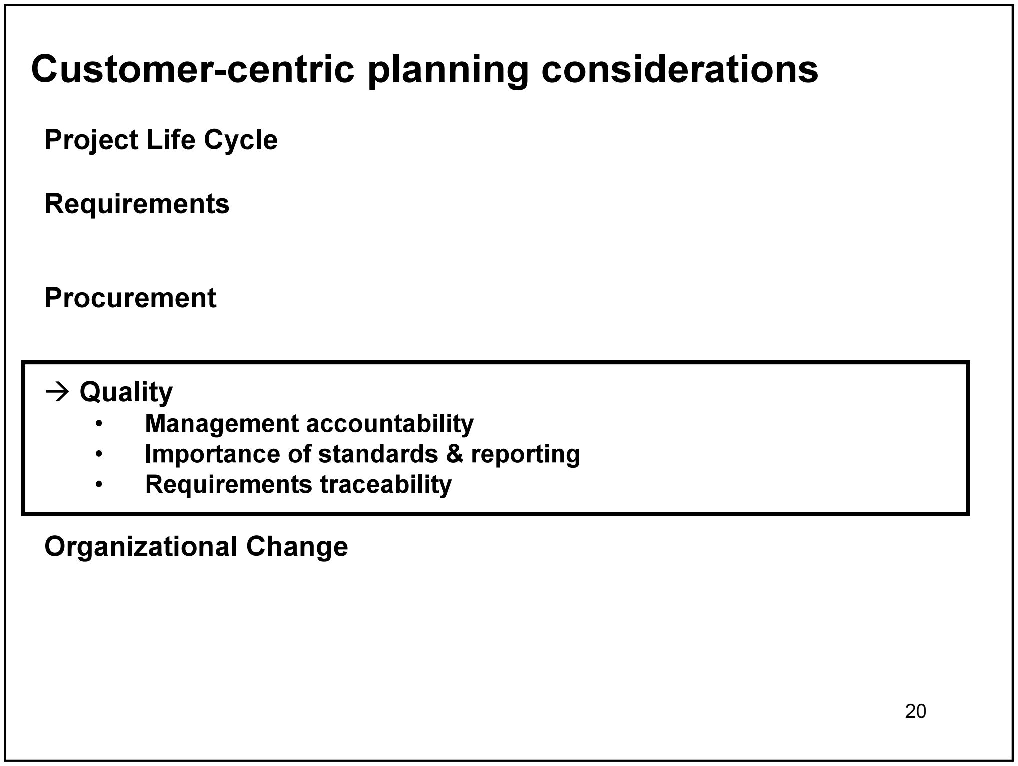
19

Whether a formal or informal procurement process is used, the choice of contractors ultimately comes down to considerations of the contractors' capabilities, process maturity, distance, and its financial stability; and the same for sub-contractors.

Capabilities refer to knowledge and experience in specific vertical industries (like healthcare, finance, or government), project management, and software engineering. A developer may be very experienced as a company, but the proposed team of personnel being proposed by the company may not. Corporate capacity to add staff and resources quickly (in the event of scope expansion or schedule fast-tracking) is an important consideration in the evaluation of developer capabilities.

Large projects may involve large teams of personnel. Expertise with project management, system development lifecycle methodology (such as for RUP or Agile), software engineering practices, tools for development and testing are especially important. Likewise for the skill levels and the accessibility (including physical distance) of the assigned contractor staff. In this sense, accessibility difficulty of contractor personnel may be geographic, time-zone, language, and cultural in nature.

Lastly, the financial stability of a contractor is important. No customer wants to see its developer goes out of business during a major project. The State of Oregon routinely require proposals of major IT projects to be accompanied by financial data for the contractor and may conduct searches to assure that a contractor is in good financial standing and is not currently involved in litigations or contract disputes.



Customer-centric planning considerations

Project Life Cycle

Requirements

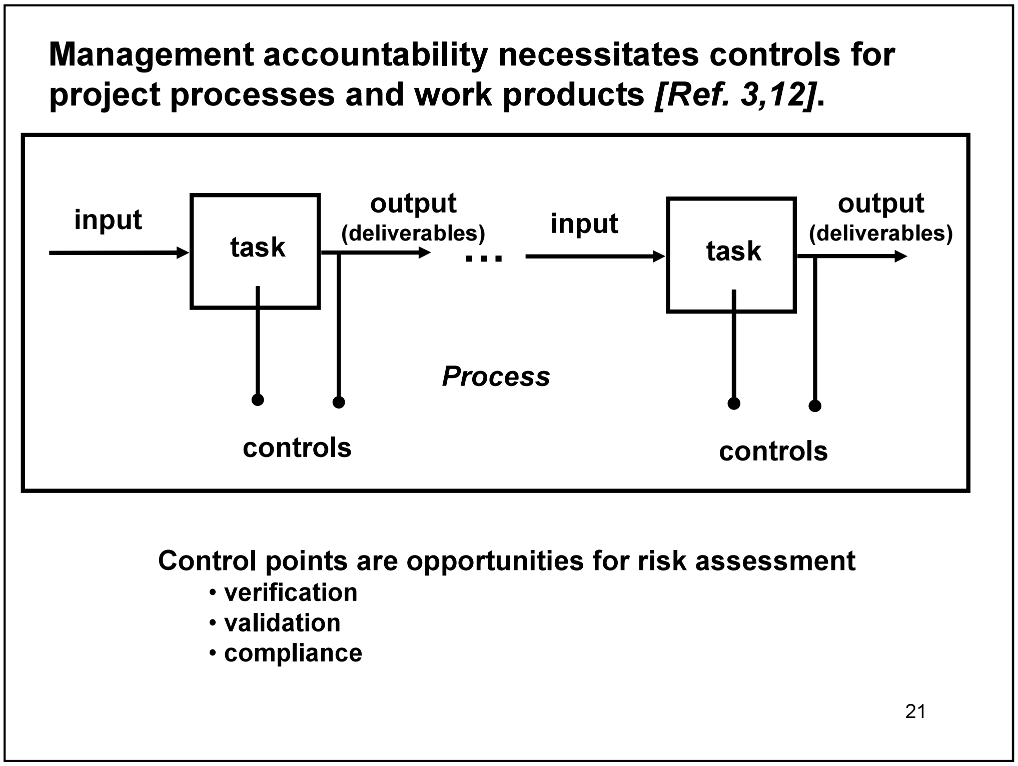
Procurement

→ Quality
- Management accountability
- Importance of standards & reporting
- Requirements traceability

Organizational Change

20

We now look at quality management.



Management accountability necessitates controls for project processes and work products *[Ref. 3,12]*.

Control points are opportunities for risk assessment

- **verification**
- **validation**
- **compliance**

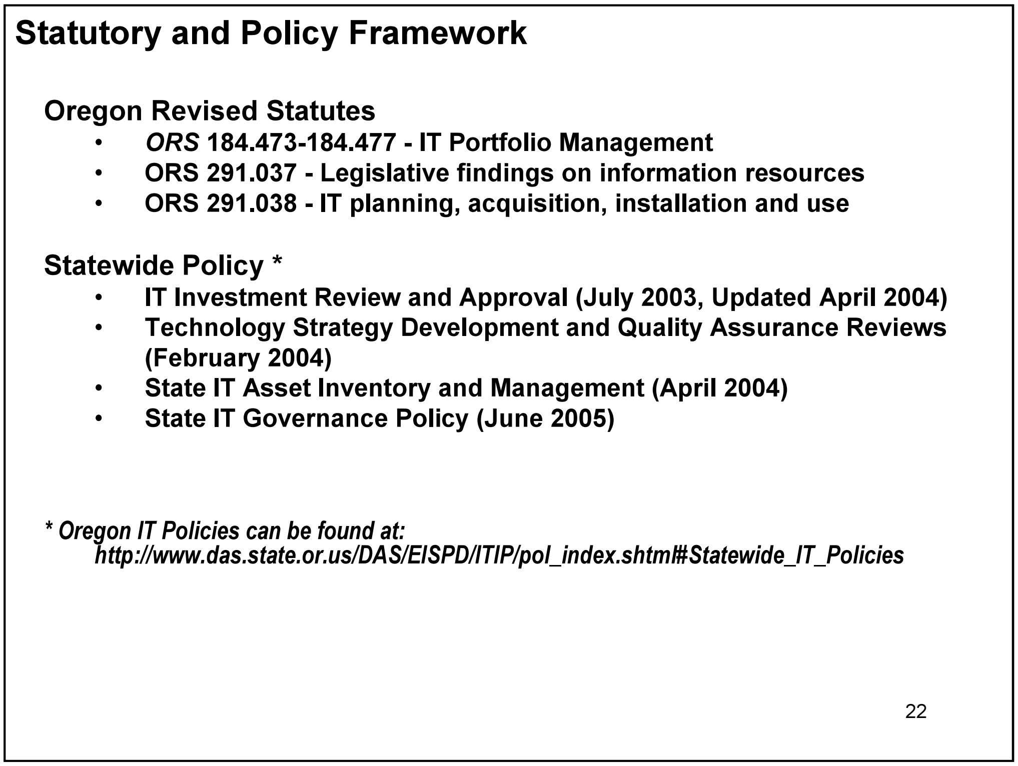
21

Obviously, enterprises outsourcing IT projects desire high quality contractor work. In quality management paradigm such as ISO 9000, quality usually refer to the quality of work products and the quality of processes for performing and managing work.

For major IT projects in large enterprises, a project and its management may be "under the magnifier" at all times. Stakeholders, including senior management, expect to be informed frequently about project status, quality, and risks. The fact that management accountability necessitates management oversight means that projects must be managed in a way where project performance can be transparently assessed or audited at all times, sometime by independent quality assurance personnel.

The process diagram above depicts a prototypical project plan for which a "task" may denote a specific iteration in an iterative SDLC, a phase in a spiral SDLC, or a task in a waterfall-like SLDC. Management control points can be imposed during the execution of a "task" to review work in progress or work already completed.

From a management standpoint, control points are opportunities for assessing work product quality usually by means of verification and validation (V&V); usually associated with testing, code review, and other means to establish that work products are "fit to use" and compliant with applicable regulations. These management control points are also opportunities for assessing and reporting project performance, such as percent of completion for a task and for the overall project and the actual amount of resources (time and budget) used vs. planned. In formal project management method such as PMBOK, Earned Value Analysis (EVA) is employed to measure budget and schedule variance relative to a baseline plan, both at the time of reporting and as estimated (forecasted) at project completion. In EVA, the percent completion of a task is usually tied to specific, discrete milestones having been reached.



Statutory and Policy Framework

Oregon Revised Statutes

- ***ORS* 184.473-184.477 - IT Portfolio Management**
- **ORS 291.037 - Legislative findings on information resources**
- **ORS 291.038 - IT planning, acquisition, installation and use**

Statewide Policy *

- **IT Investment Review and Approval (July 2003, Updated April 2004)**
- **Technology Strategy Development and Quality Assurance Reviews (February 2004)**
- **State IT Asset Inventory and Management (April 2004)**
- **State IT Governance Policy (June 2005)**

**** Oregon IT Policies can be found at:***
http://www.das.state.or.us/DAS/EISPD/ITIP/pol_index.shtml#Statewide_IT_Policies

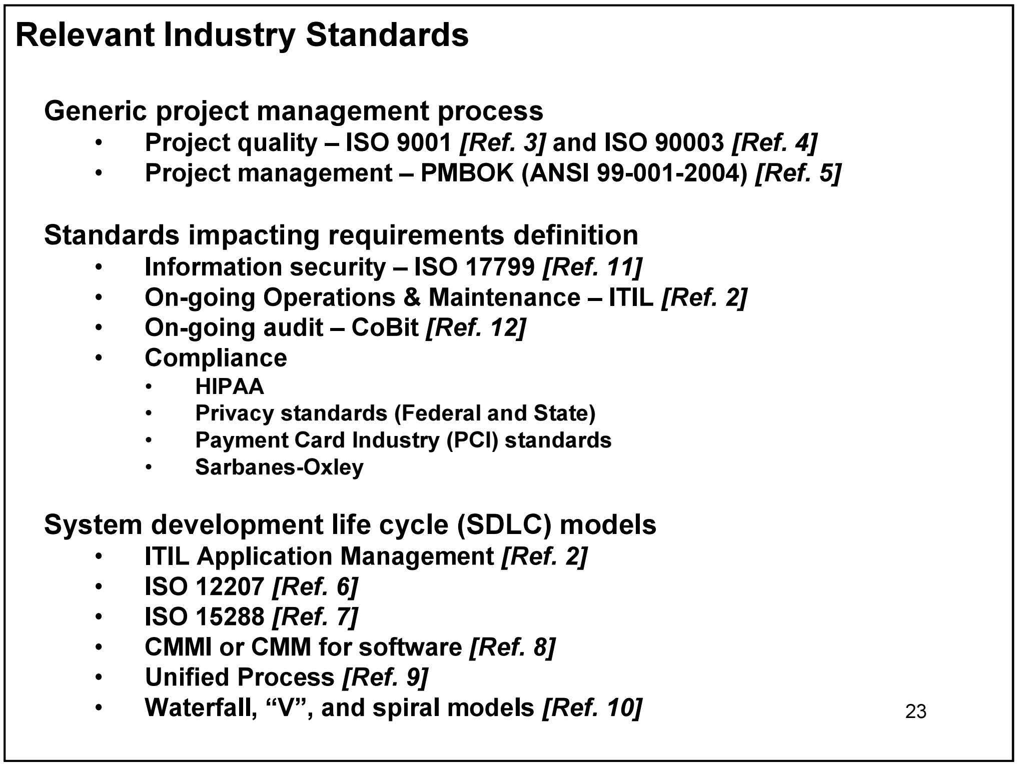
22

Another slide with information specific to the State of Oregon…

In Oregon state agencies, statewide policies traceable to state laws exist to govern quality of major IT projects.

As already mentioned, projects with value greater than certain dollar thresholds requires a detailed business case and State CIO approval before project execution.

During execution, the use of professional project management is expected; as are project status reporting, on-going oversight by the staff of the State CIO, and the use of independent QA contractor.



Relevant Industry Standards

Generic project management process

- **Project quality – ISO 9001 *[Ref. 3]* and ISO 90003 *[Ref. 4]***
- **Project management – PMBOK (ANSI 99-001-2004) *[Ref. 5]***

Standards impacting requirements definition

- **Information security – ISO 17799 *[Ref. 11]***
- **On-going Operations & Maintenance – ITIL *[Ref. 2]***
- **On-going audit – CoBit *[Ref. 12]***
- **Compliance**
  - **HIPAA**
  - **Privacy standards (Federal and State)**
  - **Payment Card Industry (PCI) standards**
  - **Sarbanes-Oxley**

System development life cycle (SDLC) models

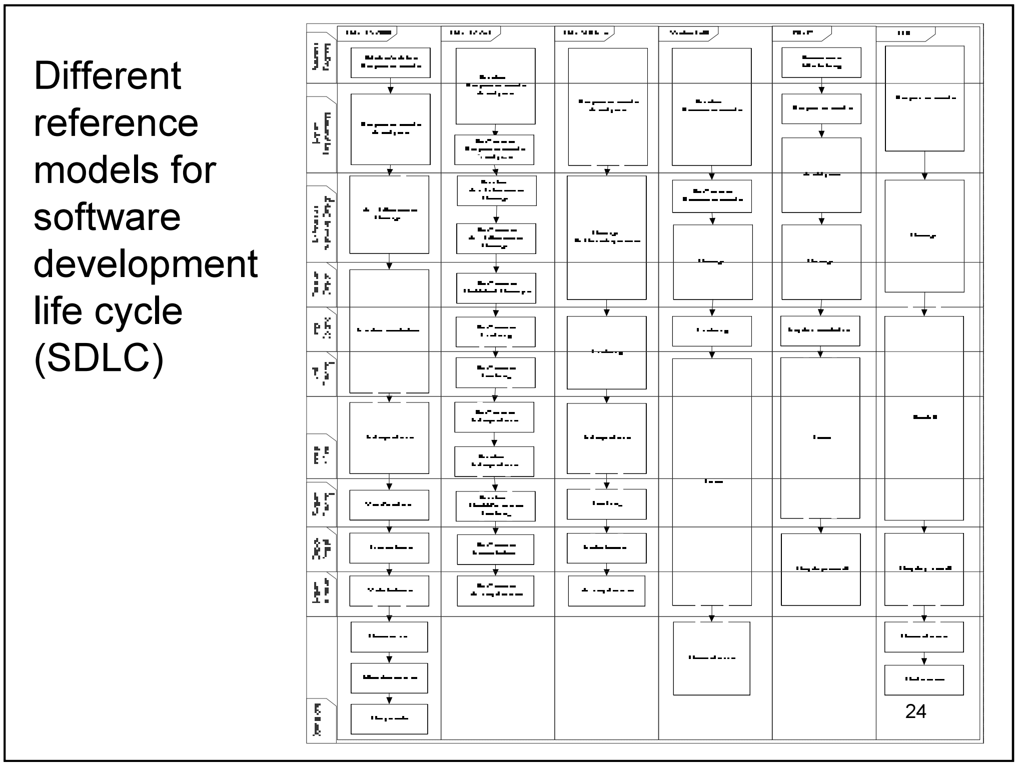
- **ITIL Application Management *[Ref. 2]***
- **ISO 12207 *[Ref. 6]***
- **ISO 15288 *[Ref. 7]***
- **CMMI or CMM for software *[Ref. 8]***
- **Unified Process *[Ref. 9]***
- **Waterfall, "V", and spiral models *[Ref. 10]***

23

From a business perspective, the assessment of quality is tied mainly to the functional requirements. As such, functional requirements should be the main criteria for determining if work products are "fit to use" and should be accepted and paid for by the customer.

The assessment of process quality is usually not as straightforward. A variety of industry and regulatory standards are useful here. This slide depicts relevant standards or approaches that may be valuable to project management, requirements management, and system development lifecycle (SDLC) models.

Standards associated with HIPAA, PCI, and Sarbane-Oxley may prescribe work product functionalities and may be useful for assessing quality of the work products.



Different reference models for software development life cycle (SDLC)

24

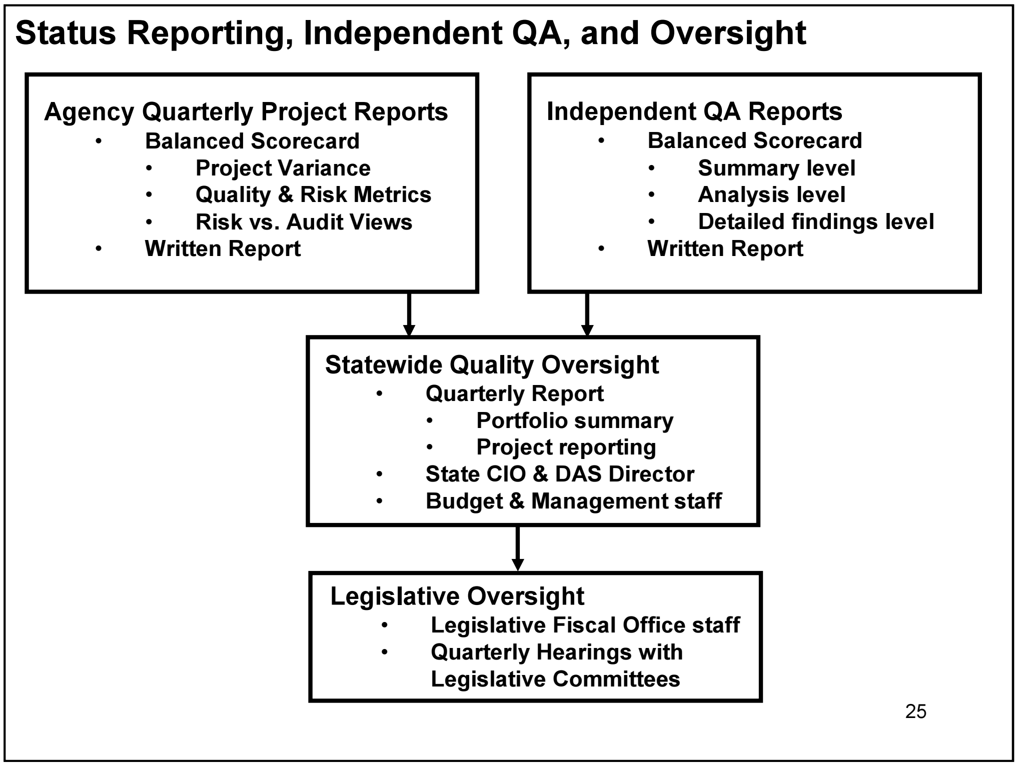
A word on SDLC…

It is often impractical or impossible to impose requirements on a developer's choice of SDLC, but the acquiring enterprise would likely have certain preferences. An enterprises may prefer RUP or Agile when requirements are not stable or captured in detail, or when formal planning or documentation is not possible. An enterprise that understands its requirements well and where contracting approaches are very formal (as in government agencies) may prefer a waterfall like methodology with formal control points or review milestones that decouple requirements, design/development/testing, implementation, and support & maintenance.

This slide maps common SDLCs (waterfall, RUP, and ITIL) to three ISO standards for software engineering:

- ISO 90003 (Software engineering -- Guidelines for the application of ISO 9001:2000 to computer software)
- ISO 12207 (Systems and software engineering -- Software life cycle processes)
- ISO 15288 (Systems and software engineering -- System life cycle processes)

The main point here is that all useable SDLCs have certain core similarities that are consistent with formal project management and quality management.



Status Reporting, Independent QA, and Oversight

Agency Quarterly Project Reports

- Balanced Scorecard
  - Project Variance
  - Quality & Risk Metrics
  - Risk vs. Audit Views
- Written Report

Independent QA Reports

- Balanced Scorecard
  - Summary level
  - Analysis level
  - Detailed findings level
- Written Report

Statewide Quality Oversight

- Quarterly Report
  - Portfolio summary
  - Project reporting
- State CIO & DAS Director
- Budget & Management staff

Legislative Oversight

- Legislative Fiscal Office staff
- Quarterly Hearings with Legislative Committees

25

Another slide with information specific to the State of Oregon…

As mentioned, major IT projects in the State can be thought of as having multiple levels of oversight. Typically, a project is under the oversight of the following entities:

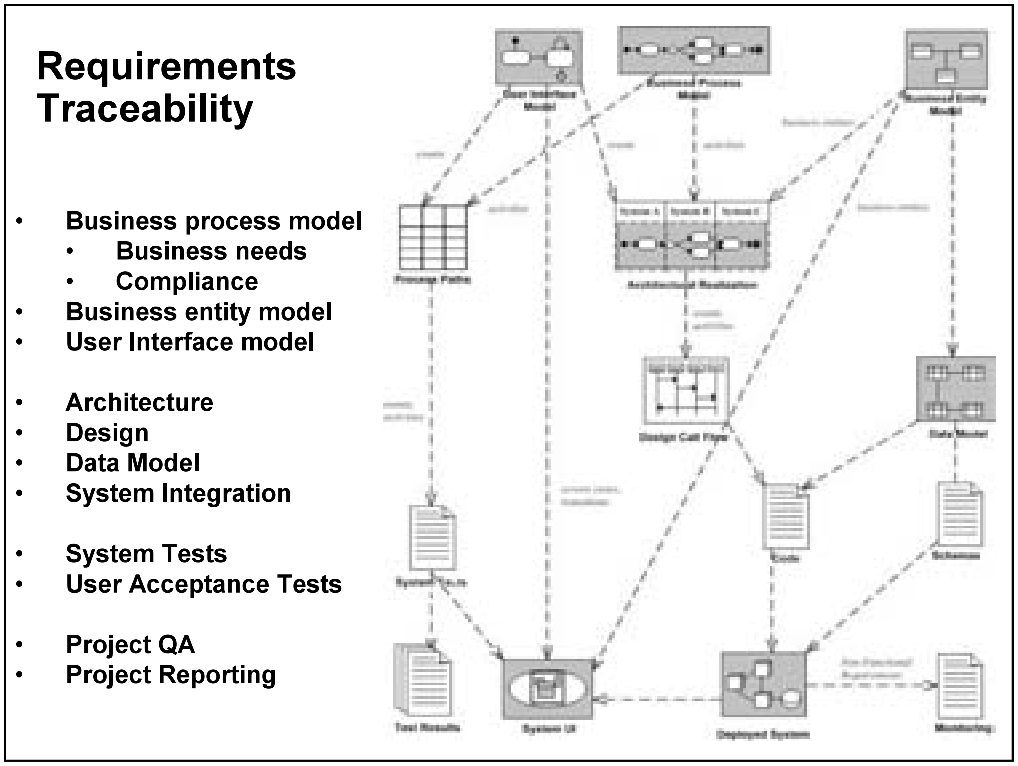
1. the management of the agency planning and executing the project;
2. independent QA contractor retained to provide independent assessment of project status, performance, and risks;
3. the staff of the State CIO;
4. legislative oversight (from a budgetary or fiscal perspective).

In addition, all projects are subject to audits by the Secretary of State, which is constitutionally independent from all executive branch agencies.

With the exception of (2), this is not too different from a large private enterprise in which a project or program may report into a director or VP of an operating division but is under the oversight of the various C-level managers, such as the CIO and the CFO. Finally, there may be external audits by an independent auditor.

The use of independent QA contractors is expected for major IT projects greater than $1 million. The goal of independent QA is to assure the independence of assessment but also to assure project performance is measured against industry best practice with recommendations for process improvement. The staff of the State CIO recommends that 4% of the overall budget of a major IT project be reserved for independent QA, based on a standard QA statement of work.

The experience of the State of Oregon is that independent QA and external oversight can be very useful tools to assure project quality.



Requirements Traceability

- Business process model
  - Business needs
  - Compliance
- Business entity model
- User Interface model

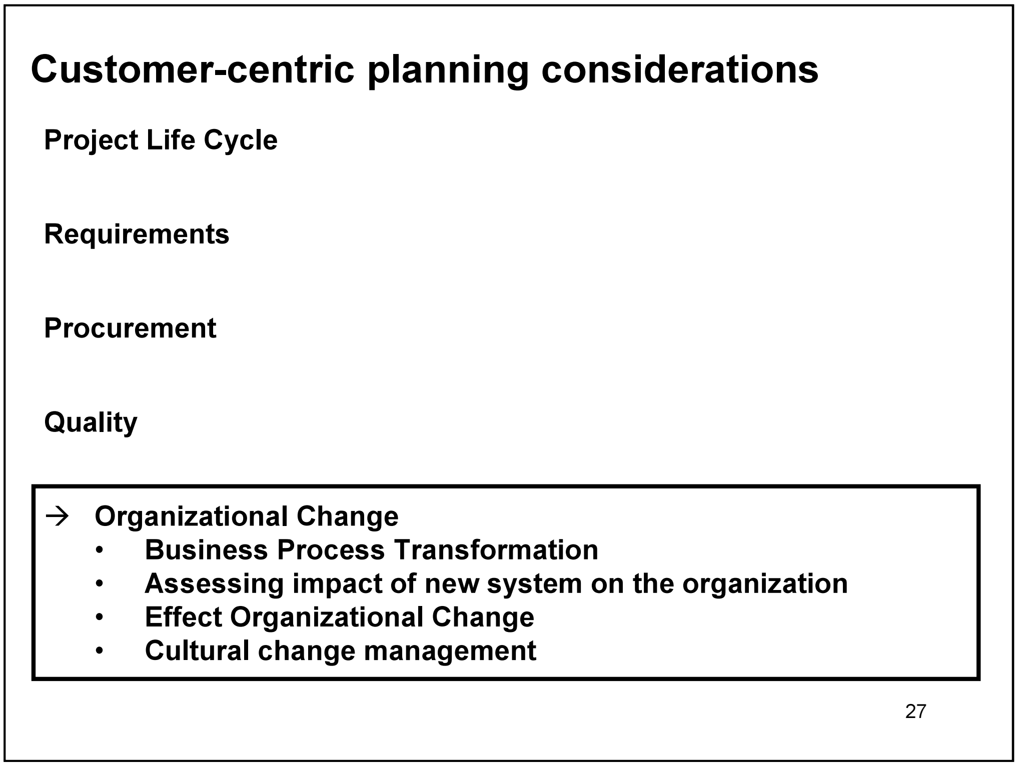
- Architecture
- Design
- Data Model
- System Integration

- System Tests
- User Acceptance Tests

- Project QA
- Project Reporting

We have already discussed a fair amount about requirements management and its role in quality management of an outsourced project. This slide emphasizes the importance of business requirements as the driver for business process modeling, system architecture and design, acceptance testing, and quality monitoring and reporting.

Historically, large enterprises exhibit a chasm between its IT and business departments. Many failures of major IT projects can be attributed to the disconnect between the IT view of the enterprise vs. the business view of the enterprise. In recent years, there are wider recognition and acceptance that major IT projects need to be driven by business requirements in order to accomplish desired operational improvement and strategic objectives. This is a healthy development that helps assure overall quality.



Customer-centric planning considerations

Project Life Cycle

Requirements

Procurement

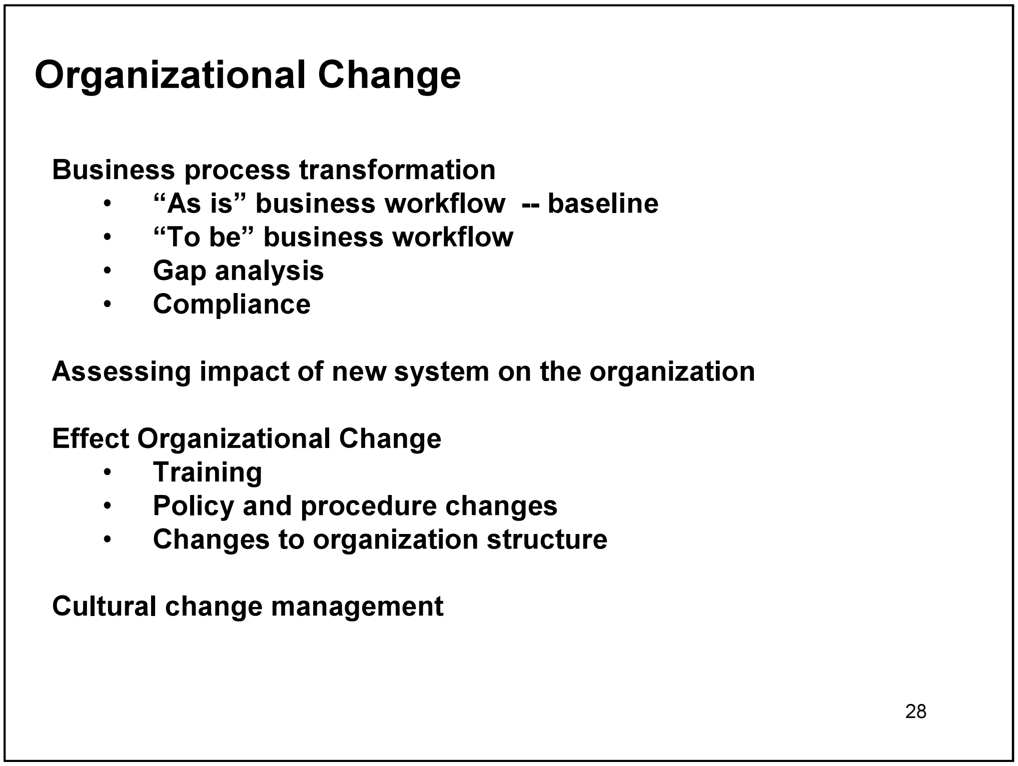
Quality

→ Organizational Change

- Business Process Transformation
- Assessing impact of new system on the organization
- Effect Organizational Change
- Cultural change management

27

The final topic of this presentation is Organizational Change.



Organizational Change

Business process transformation

- **“As is” business workflow -- baseline**
- **“To be” business workflow**
- **Gap analysis**
- **Compliance**

Assessing impact of new system on the organization

Effect Organizational Change

- **Training**
- **Policy and procedure changes**
- **Changes to organization structure**

Cultural change management

28

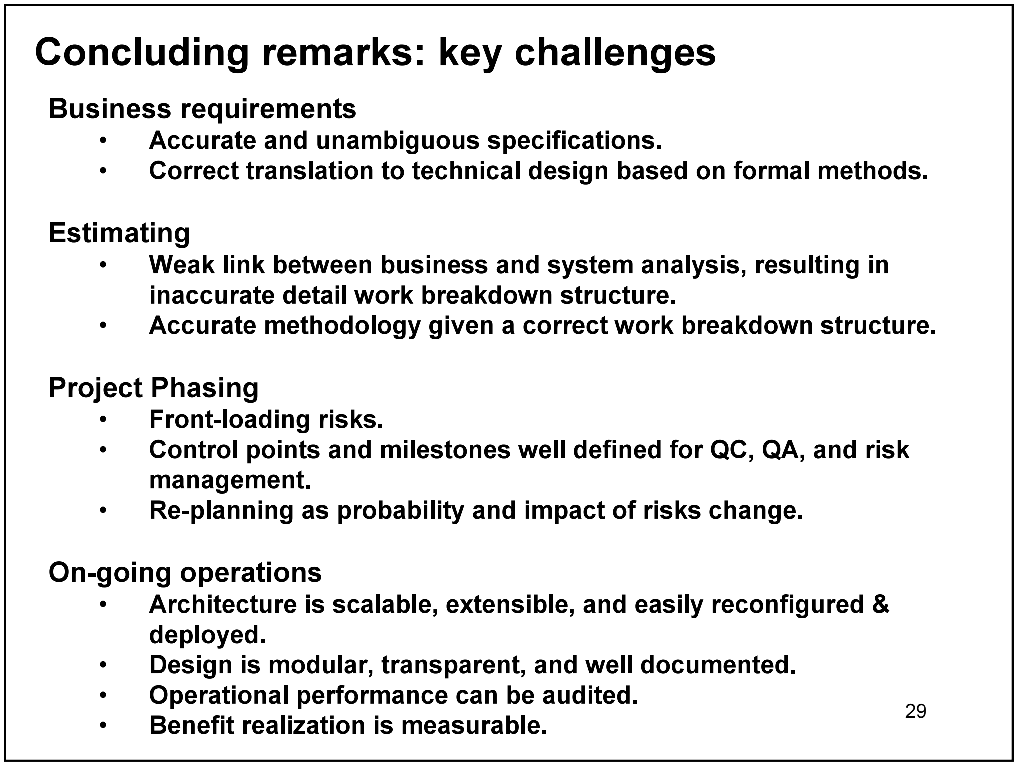
By nature, a major IT project has big impact on the operations of an enterprise. Frequently, such a projects is the enabler for major business process improvement or re-engineering and may entail significant change to staff responsibilities and even organizational structure.

Human nature is naturally resistant to change, and human and organizational factors impose challenges to all phases of a major IT project. A reasonable way to assure quality of the overall process is to emphasize business requirements and business process. In this regard, a good starting point of a major IT project is the documentation and analysis of the As Is business process. The To Be business process is then defined, ideally in a collaborative manner between contractor staff and customer staff.

The foundation of To Be business processes should be business requirements, regulatory requirements, and the capabilities (as well as constraints) of new technology. Optimization of proposed future processes should be done on the basis of balancing potential efficiency gain and return-on-investment with organizational change. Given two processes with similar effectiveness or efficiency, the process that results in smaller change, or gap, should be chosen.

Impact of the new system on the enterprise needs to be evaluated and analyzed across functional units. Training needs should be an integral part of the project plan and its budget, as should efforts required to update relevant policies and procedures. Organizational change and even cultural change may also be necessary in order to assure overall implementation quality of a major IT project.

Clearly, technical success *is necessary but not sufficient* for the overall success of a major IT project and related enterprise business process improvement initiatives.



Concluding remarks: key challenges

Business requirements

- **Accurate and unambiguous specifications.**
- **Correct translation to technical design based on formal methods.**

Estimating

- **Weak link between business and system analysis, resulting in inaccurate detail work breakdown structure.**
- **Accurate methodology given a correct work breakdown structure.**

Project Phasing

- **Front-loading risks.**
- **Control points and milestones well defined for QC, QA, and risk management.**
- **Re-planning as probability and impact of risks change.**

On-going operations

- **Architecture is scalable, extensible, and easily reconfigured & deployed.**
- **Design is modular, transparent, and well documented.**
- **Operational performance can be audited.**
- **Benefit realization is measurable.**

29

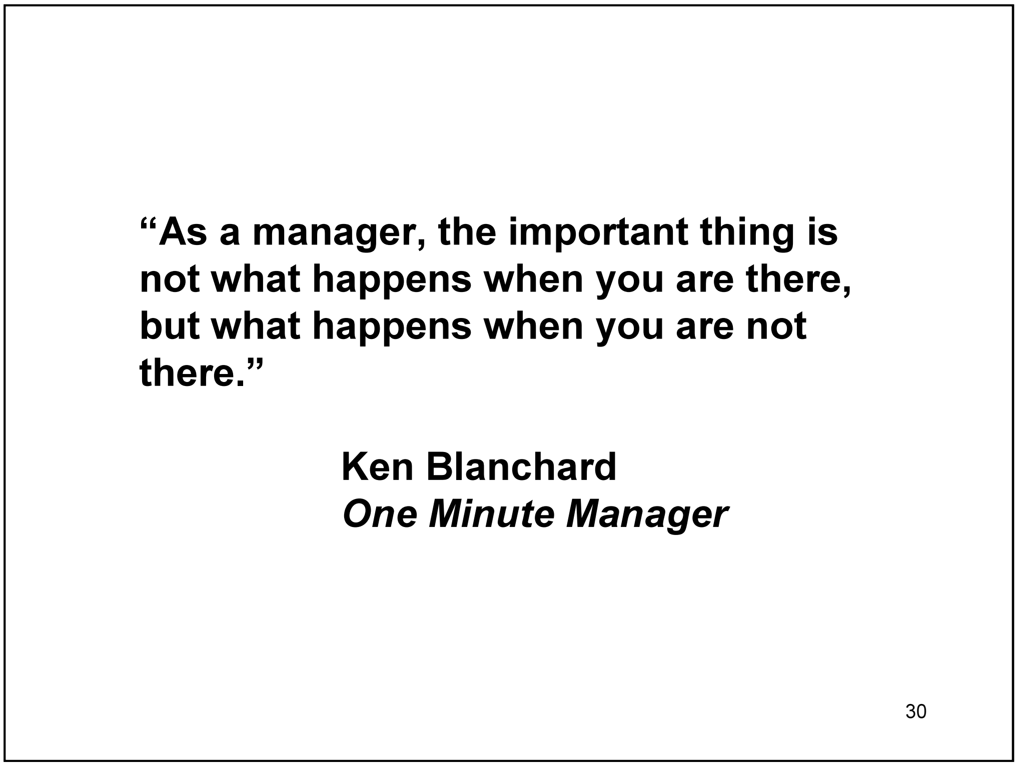
This presentation will conclude with the author’s impression of key challenges for quality management of major IT projects that are outsourced.

First, it is difficult to define good requirements. Software requirements are frequently not well understood even by the acquiring enterprise. One reason is that business conditions evolve rapidly. As such, it may be difficult to have stable, well documented requirements. A second challenge is the chasm that tends to exist between an enterprise’s IT department and its operational units. As a result, there is always a risk of miscommunication during requirements definition and their subsequent management.

Second, even when requirements are stable, the method for estimating resources and time can be a challenge. Many contractors’ work plans are based on estimates that may not be transparent to the customer. Frequently, there is inadequate allocation of resource and time for customer reviews, data conversion, user acceptance testing, and rework.

Third, project phases frequently delay high risk areas, instead of front-loading them (as recommended in SDLC like RUP). Projects that progresses well (or has high velocity in the sense of Agile SDLC) may slow down if high risks areas are deferred to later iterations. Management pressure to see early results sometime drives the tendency in delaying high risk portion of the project. This presentation advocates the inclusion of ample intermediate milestones to facilitate project status/performance assessment. These milestones are also potential management control points for quality and risk assessment. Project managers may consider Earned Value methodology of the PMBOK, but other ways to facilitate good communication between developer and customers are also beneficial.

Fourth, a project needs to take into account on-going support and maintenance needs of an enterprise application early, beginning with the architecture. Designs needs to be robust and supportable, with tools to support on-going monitoring of operational performance and key performance indicators, both in business and technical terms. Where possible, architecture and design need to conform to enterprise stand



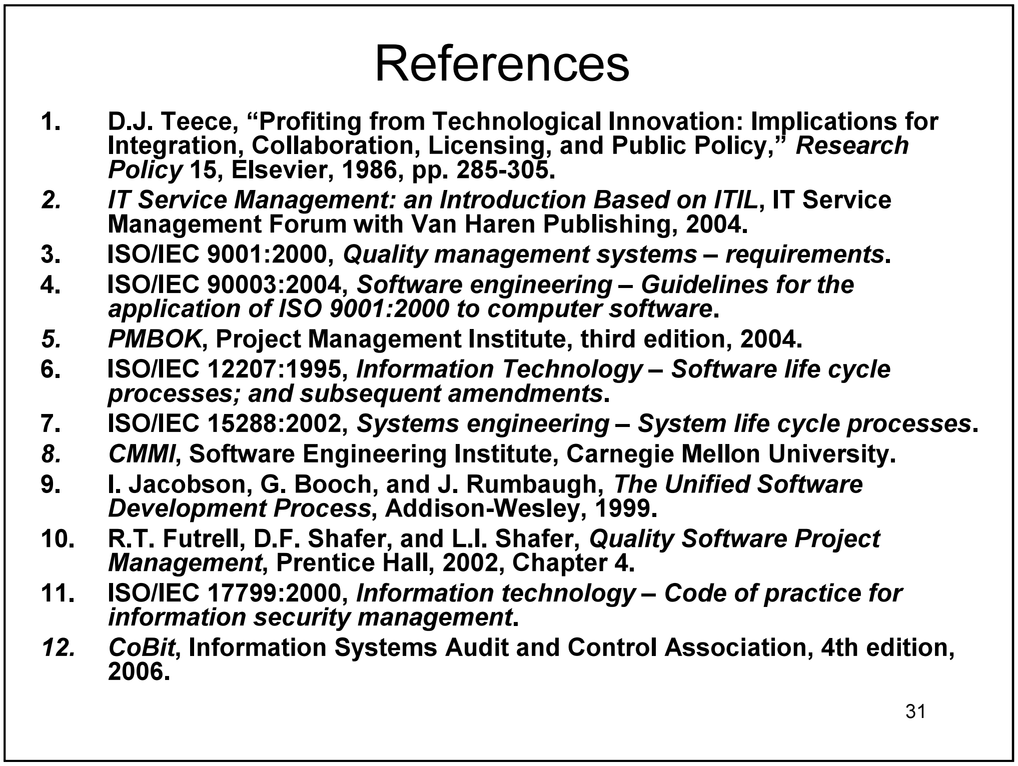
"As a manager, the important thing is not what happens when you are there, but what happens when you are not there."

Ken Blanchard
One Minute Manager

30

The author believes that the key to quality is process, and the management of quality is ultimately about management of process that "designs in" quality.

We end this presentation with this quote. Thank you.



References

1. D.J. Teece, “Profiting from Technological Innovation: Implications for Integration, Collaboration, Licensing, and Public Policy,” *Research Policy* 15, Elsevier, 1986, pp. 285-305.

2. *IT Service Management: an Introduction Based on ITIL*, IT Service Management Forum with Van Haren Publishing, 2004.

3. ISO/IEC 9001:2000, *Quality management systems – requirements*.

4. ISO/IEC 90003:2004, *Software engineering – Guidelines for the application of ISO 9001:2000 to computer software*.

5. *PMBOK*, Project Management Institute, third edition, 2004.

6. ISO/IEC 12207:1995, *Information Technology – Software life cycle processes; and subsequent amendments*.

7. ISO/IEC 15288:2002, *Systems engineering – System life cycle processes*.

8. *CMMI*, Software Engineering Institute, Carnegie Mellon University.

9. I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.

10. R.T. Futrell, D.F. Shafer, and L.I. Shafer, *Quality Software Project Management*, Prentice Hall, 2002, Chapter 4.

11. ISO/IEC 17799:2000, *Information technology – Code of practice for information security management*.

12. *CoBit*, Information Systems Audit and Control Association, 4th edition, 2006.

31

Building a Successful Multi-Site Team Using Chocolate

Doug Whitney dwhitney@mcafee.com

Doug Whitney manages two QA teams at McAfee. He has presented papers at PNSQC and Quality Week on various topics. He has 16 years of QA experience and has managed QA teams at both McAfee and Intel.

Srinidhi Krishnan srinidhi_krishnan@mcafee.com

Srinidhi Krishnan manages two QA teams in Bangalore India for McAfee. He is involved in several QA initiatives within McAfee's QA organization. He also managed the corporate applications group for McAfee in Bangalore, India.

Abstract

Working with remote teams, whether across town, or across the ocean, can have its set of challenges. Trying to determine how to interact, when to interact, and if to interact can make the difference between a successful end result and a failure. The teams in both the US and India for the McAfee product Total Protection Service have created an ongoing successful model that has resulted in on time delivery and high team moral. There seems to be a common thread that binds the team together – CHOCOLATE. This paper will describe the initial interactions, planning methods, travel tips, team building experiences and types of chocolate used to enable our successful team experience.

Introduction

The idea for this paper was an ongoing set of dialogs between me (Doug) and my counterpart in India (Srinidhi). We wanted to demonstrate that there are two points of view, one from the US perspective and one from the India perspective. You will see sections that encapsulate the abstract and are from both points of view. We do not boast to have all of the answers and our working methodology is a constant work in progress. We have found some success and that is what we are trying to describe.

Doug:

When I was a kid, I always referred to my father as “sir”. I just got in the habit of referring to other people, regardless of age, more formally. I was also very polite by saying thank you and holding the door for people. I had no idea that these actions would prove valuable to me later in life in an effort to build relationship with people in another country. It seemed to work out that way for me in my relationship with the QA team working with me in Bangalore, India and it wasn’t from the perspective that I thought it would be either. The VP of Quality told me prior to my first trip to pay for team events and lunches. When I arrived the Director of Quality in India and I had a one on one. In it she told me that the team considers it an honor to have a “visitor” and to allow them to pay and to set up events. I quickly learned what she meant.

Srinidhi:

Working successfully in a shared mode with a distributed team is a challenge. The common force that binds us together is the product vision, the cause, the drive to be the best amongst other teams across the company and off course “Chocolates”.

Initial interactions

Doug:

When I first learned that I would be managing a team that was located in India, I had no idea of what to expect. I had dealt with multi-site US teams before, while working at Intel, but the involvement of those teams was not the daily tasks that it is with this team. Initially, my dealings with the team in Bangalore started out on the phone. There was a weekly meeting that had been scheduled by the previous manager for 7:00am on Tuesdays. I wondered why so early? On the initial call I introduced myself and describe my working methodology. One of the items that I stressed was that I do not expect anyone on the team to do something that I am not willing to do myself. I really don’t think they understood this at first. They would. I also didn’t understand the relationship that an Indian employee has with a person of responsibility, especially from another country. The 7:00am meeting continued since they are twelve and a half hours difference from the Pacific Time zone and they were all staying late in their day to attend. We worked well together and since it was relatively late in the project cycle, the tasks they were already doing provided value in releasing the product. When we shipped, I thanked them for their efforts. It was the acknowledgement and the appreciation, the holding of the door if you will, that helped to solidify the working relationship.

Srinidhi:

Having a counterpart in the US really helped in getting the right Information and direction at the right time. When I was first assigned to the team, quality time was spent in understanding individuals and teams onsite. During this process we built mutual trust which happened to be the foundation of our mode of working. The beauty of such a cohesive team is that it has a mix of people from different cultures, different styles, different modes of communication and different ways of interpreting things. Our weekly telephone conferences did help to mitigate these factors. It was worth the time even if it meant staying late in the office. Such conversations also helped us in understanding technical problems with our product and in coming up with solutions. We did define clear roles and responsibilities across sites and planned tasks in a way that there is no redundancy across sites. By doing so, we never encountered a conflicting situation. We started to build a "can do" attitude within the team. Our collective decision making did bring about a level of belonging among the team members, who would then, open up and start sharing technical ideas. With the technical skills, the right attitude and effective use of the 24 hour cycle across sites, this team was on a mission to prove that we can work seamlessly even if the team is distributed. The first couple of successful releases were a testament to what the team could achieve. After every release we would go back to the drawing board and do an assessment of what went well and what did not go well. This helps us to improve and fine-tune not only our processes, but also our working model.

Travel Tips

Doug:

Shortly after becoming the QA manager for the ToPS team, I was told by the VP of Quality that I should travel to India and meet the team. He felt it was very important to be able to have a face to go along with the name. He also mentioned that the teams in India really enjoyed American chocolate. I had a passport, but needed to have a letter of introduction from the McAfee India office in order to obtain a visa for travel to India. Once we (including our very helpful Admin, Marlana) received the visa and ordered the tickets, I thought I was all ready. Not quite. Again, taking advice from others that had travelled to Bangalore; I was informed that I should go to my doctor for "shots". I was given a Hepatitis shot (series actually), a prescription for both a daily malaria (don't take the weekly) and Cipro. Cipro is an anti-biotic you take in the event you developed traveler's diarrhea. The primary way to get this is to drink the water in India.
I developed a standard rule of thumb while travelling to Bangalore. It is to eat food that is baked, boiled, or peeled, and to drink only bottled water. The team looks out for me while I am there by ordering bottled water at each restaurant we visit. There was one time when the restaurant served ice cream and it looked very good, but the scoop was sitting in a bucket of warm water. I disappointedly avoided the ice cream. I even use bottled water to brush my teeth.

The flight is long. My flight involves flying for 10 hours to Frankfurt, Germany and then another 9.5 hours to Bangalore, The flight arrives in Bangalore around midnight and after clearing customs and finding my ride to the hotel (we also utilize a corporate apartment) it is about 2:00am. After sleeping for a short while, it is off to the office by 9:00am. I usually begin to feel sleepy at 4:00pm my first day there. On the trip to the office on the first day, you are introduced to traffic in Bangalore. There are lane markings on some roads, but there are ignored. On a road with two lanes each direction, there are five to seven vehicles. Some are cars, some are motorcycles and some are three wheeled rickshaws that are taxis. The other thing is that everyone honks their horn. The rooms at the hotel (the Oberoi) have and outer door and an inner door to keep out the street noise. One more thing about travel, you are only allowed a certain weight for your checked bag (it varies for each airline), so plan carefully. I had too many one pound bags of chocolate and needed to move some to my carry on. ☺

Srinidhi:
Traveling to the US has been a pleasant experience for me especially after joining the ToPS QA Team. I have been well received by my counterparts in both QA and development and they have been very co-operative. We feel that face to face interaction is very much needed in a distributed team. Both teams try to make a point to have someone travel at least once in a year on crucial

assignments.

Planning Methods

Doug:

The way in which we plan projects requires a software quality plan (SQP) at the point we reach the plan complete milestone. Within the quality plan is a QA schedule. One of the methods we utilize in order to get the schedule data is to take the development components from the engineering plan and from that create a testing schedule spreadsheet. One of the first planning exercises that we did as a team was to have the test leads in both the US and India take this spreadsheet and compile their estimates as to who owns, and how long it will take to test (develop test cases, completed the functional testing, and to regress any defects) the individual components. They went to the teams and received feedback. Once this was completed, there was a pivot table to calculate which tester was testing what component and what was the total time required to complete the testing. This data was very valuable to the schedule portions of the SQP, and more importantly, it came from the team.

We have a set of tasks centering around the posting of the files on the server that is an exact process. We needed to do this in order to avoid an erroneous posting that would cause all of the customers on the service to receive incorrect files. Due to personnel changes, this posting process needed to be moved from the US to India. Our posting engineer in India would ask questions, but would take the stance that the US team would always have the answer. I actually gave him permission to challenge me; to ask questions if HE deems something incorrect about the files we were posting. The first few times, he would apologize after proposing a question. Once he felt comfortable and to this day, he will question our process as well as the correct implementation.

We now have calls in the evening US time. They are usually at 9:00pm, which is 9:30am in Bangalore. We did this since there are more people that attend in Bangalore. We have one meeting a week that is a team meeting where all attend from both the US and India. We have other meetings that are done as needed during the critical points in the project. There are usually no meetings on Friday nights (India Saturday). When we do not have a meeting scheduled, we utilize instant messenger to communicate ideas and quick thoughts. I find that when I am logged on in the evening, I will check to see who on the India team is logged in and send them a note. We use the instant messenger extensively and would recommend it for any remote team.

Srinidhi:

By adopting an Agile Mode of Operation, we have been able to reap a lot of benefits like having shorter development cycles which leads to lower risk, less changes, greater predictability and higher quality. By adhering to an Agile Resource Model, we have come up with a spreadsheet to estimate the target quarter based resources availability, leading to better effort sizing. We have also implemented user story based development with complete traceability to the MRD. By getting QA involved early enough and following a test driven approach that pairs QA and Dev engineers together, it has resulted in high quality releases. We also work with our development counterparts in attempting early defect prevention rather than defect detection at later stages.

We bring product specific functional teams together as a core team for efficient problem solving. This would comprise of QA, Dev, Tech Pubs and Support teams. During crunch time we implement a daily stand up meetings. We use this as a forum to bring up “what I did yesterday”, “what I’m doing today”, any blocking issues, and it also acts as a forum to get other issues resolved. This process takes less than 5 minutes of time. We have also come up with simple Agile reporting artifacts, which takes less time to fill, but gives a clear indication and status of the testing activities. We also have recurring telephone conferences, to keep people across sites on the same page. All project related documents are located and maintained in a single location (Twiki) for cross site reference.

We plan for ownership of components to individuals across sites. This helps with individual specialty, but it also helps to cross train with coverage issues that may happen. All engineers attend planning meetings and are assigned goals and deliverables. Attendance at these meetings is an active not a passive role as all are free to ask questions, get involved, and collect the information needed to design effective tests. For each user story document we help determine the risks, issues, concerns, complexities, assumptions, and dependencies. We also encourage cross site review of Quality Plan, the schedule, the user stories, and the results.

We believe that "Anything which cannot be measured cannot be improved". We generate metrics through all the stages of product development, not just the QA stabilization phases (Requirement, Planning, Design, Development, QA, Release and Support). We constantly refer to these metrics and set higher level of expectations and improvements for future releases.

Team Building

Doug:

During one of my first visits to India, I was asked if I wanted to interview one of the candidates for a test position. During the course of the interview, I asked the question, "Do you like chocolate?" The individual looked at me for a second, and then smiled. When she said yes, I told her she would get along well with the rest of the team. This technique has been repeated during other interviews as well. It makes the person relax and also helps them to understand that we are a team. I have been told this is very different that the "normal" questions that are asked, but one that is welcomed by the candidates.

During every visit I have made to Bangalore, I have brought chocolate. I will bring some in on the first day in the office and set it on one of the desks. That desk becomes a gathering place for discussions about the product as well as about the people. It works as a very pleasant stand up meeting to catch up with each member of the team (as well as people from other teams). In addition, whenever I am in India, we will go out as a team to lunch. The team finds this a good opportunity to share thoughts and ideas.

Just after my first trip to Bangalore, one of the QA managers from India came to the US for some training. On one evening I asked him if he would like to go to dinner. He brought one of the QA engineers that he worked with and we had dinner at my home. Now, whenever one of the team comes to the Beaverton office, they come to my house for dinner. Sometimes, for the vegetarian type eaters, we would have a toasted cheese sandwich and tomato soup and all enjoy it. For the non-vegetarian, it is a barbeque. ☺ I have also been invited to homes in Bangalore for dinner while I visit.

Srinidhi:

After every major release, we make sure that we go out on a day's event with the team. Past events have included team lunches, bowling, go-kart racing, and water games. This would rejuvenate the team. Very recently we went to watch a 20:20 cricket match. All members on the team will act as product and company ambassadors. We distribute promotional materials like caps, t-shirts, jackets, coffee mugs to show off the company or product specific logos. We try to ensure that there are "Team Building Events" on a time bound basis so as to bring the team much closer, which results in a better understanding of each other and ultimately results in higher efficiency of the team.

Types of Chocolates

Doug:
Hershey kisses
Ghirardelli's
Nestle value pack (Nestles Crunch, Nestles Chocolate with Almonds and Nestles Milk Chocolate)

Srinidhi:
Dove
Ritter Sport
Hershey Almond
Nestle-Crunch

Conclusion

Doug:

People all have basic needs and one of the most important is respect. A team is a group of individuals, who are working on a similar task that will provide some benefit. On a successful multi site team, there is also mutual respect. Maybe by engaging with your team, asking for their input, and providing an open environment for sharing both face to face and electronically, you can build that mutual respect. It might be as simple as a thank you sir, or a let me get that for you, but please, don't forget the chocolate.

Srinidhi:

We believe that for any individual to be at peak performance, they should be at ease. That is one of the reasons why we believe in a "Fun at Work" culture. The organizations role is very important in building a successful team. There has to be strong thought leadership in driving career development and training. In recognizing talent and rewarding the deserving ones, having employee friendly policies and work culture, we would help to ensure higher employee satisfaction levels.

Considerable amount of time should be spent on getting resources that have right mix of talent and attitude, the success lies in inducting and grooming them with team values. We need to align individual goals to that of the product and organization. This helps individuals in the team to achieve their aspirations and thereby adding value to the product and the organization. Enough opportunities should be given for innovation at work. The key lies in encouraging and motivating the team to achieve something new that would add value to the company. The theme of our teams is chocolates; we make the candidate feel at ease during the interview process by asking if they like chocolates? This would ensure that the person sheds away the fear or tension associated with a normal interview and be more involved in discussions, this is one of the binding factors of the team.

Distributed Agile

An Experience Report Joy Shafer

Abstract

This paper details our journey from chaos to concord while developing software as a service with a globally distributed team using Agile methodologies. It highlights the challenges and successes we experienced as we concurrently developed two small, first-version, online services using a development team that was located in Redmond and Moscow, and a test team that was located in Redmond and India. We encountered many interesting and difficult challenges, but were able to successfully overcome them and release high quality software on time, delighting our stakeholders. This experience report highlights our learnings, focusing on several critical success factors, such as well-defined processes, cultural awareness, continuous integration, open communication and relationship building.

Introduction

About eight months into our first Agile development projects, the Development Lead, the Program Manager Lead, and I, the Test Lead, attended a seminar titled "Agile meets Offshore." The presenter, Ole Jepsen, who has considerable experience in this arena, told the audience not to try Agile offshore on "… first releases of complex and high-technology-risk projects, if your onshore development process is not in place, [or] if you don't have any onshore Agile experience."[1] My colleagues and I looked at each other, and the Development Lead said, "No wonder this is so hard. We're doing all of those things!"

Six months later, we had fallen into a cadence of releasing both of our online services on time every two months. These were the easiest releases I've experienced in my fifteen-year career in software testing. We were even able to release a week early in one case. Our software quality and team morale was high, we worked productively in 'round-the-clock' shifts, and the offshore teams were truly an extension of the core team.

Some of us were enthusiastic about using an Agile software development methodology from the beginning. Our management was sold on Agile and sent us all to Scrum Master training. The basic tenets of Agile made sense to me. I had been in software development for many years, and shipped many different products. I had long been an advocate of involving test in the process as early as possible, and keeping the bug counts low. The concept of short, iterative sprints was new to me, but one I thought held promise.

Once the class was over and we went back to our office and tried to implement Agile, we struggled. There were many on our team who were rather vocal about how silly the methodology seemed ("Pigs and chickens? Please spare me!"). They did not want to track their hours. They did not want to spend the time needed in meetings to plan a sprint fully, and they did not want to be held accountable daily for the work they were supposed to be doing. There were also struggles with bringing the offshore team members into the process.

As a management team we patiently, and sometimes not so patiently, met every objection. We persevered and found solutions to most of the problems, eventually creating a very effective Agile process that the whole team embraced.

How We Used Agile

Wikipedia defines Agile Software Development as a conceptual framework for software development that promotes development iterations, open collaboration, and adaptability throughout the life cycle of the project.

Agile comes with its own set of vocabulary. We used a method called 'Agile with Scrum.' We did not use paired programming, test driven development or story points, although some of us were strong proponents of these methods, and we may eventually have adopted some of them if we were still together as a team.*

Agile is characterized by:

- Short iterative 'sprints' which generally last 2 to 4 weeks

  Each iteration includes a detailed planning session at the beginning of each sprint, and at the conclusion of the sprint there is a retrospective to pinpoint areas for improvement, and a demonstration to stakeholders of what was accomplished during the sprint. By the end of the sprint, all features under development should be completely finished. That means not only coded, but also tested, and all bugs fixed and verified.

- Agile has its own terminology

  ScrumMaster: the person who facilitates meetings, unblocks team members, and runs interference against those who would randomize team members
  Product Owner: the person who makes decisions about what features should be built
  Product backlog: a prioritized list of features or other desired or required work items, such as infrastructure improvements

- Intensive communications: daily 'Scrum'

  The Scrum meeting is a fifteen minutes daily meeting whereby the team members are held accountable for the work they committed to completing the day before. The scrum is also an opportunity for the 'Scrum Master' to discover and remove blocking issues.

*Our group was reorganized last summer into a larger non-Agile team, and many of the team members have moved on to different projects. One of our services was shut down even though it was successful with customers, because no one could figure out how to make money from it.

- Small teams and team focused control

  A typical Agile team is about eight people. Ideally the team is co-located and able to communicate easily about their work throughout the day. Because of the structure of Agile, the team holds each other accountable for the work, and management should be less involved in making sure day-to-day work is completed.

- Division of work by product functionality rather than by task

  As the Agile team works together to accomplish their objectives, and given the short timeframe of a sprint, traditional roles of development, testing, project management, technical writing, and so on tend to be shared across the team. Some sprints may be 'test heavy' in which case developers or doc writers or other non-test team members may lend a hand to the testing. In some sprints, testers or other technical team members may fix bugs or work on tools that would typically be done by a developer.

Agile works well when requirements are evolving, as new requirements can easily be accommodated at the beginning of every sprint.

Project Details

We were a newly formed group, charged with building two brand-new, version-one online services.

Windows Live VoiceMail is a service that forwards your telephone voicemail to your Windows Live (Hotmail) account. We were responsible for building the middleware piece that receives the voicemail from the Network Operator, translates it into a different format and forwards it on to Hotmail. It also kept the Hotmail inbox in sync with the Voicemail inbox.

This team consisted of Developers, Testers and Program Managers. We had six based in Redmond, five in India and three in Russia.

We worked with four internal Microsoft partners and two external partners.

Windows Live Call for Free was a service that could be accessed through a link from the Earthlink site (http://maps.live.com). It connected users with merchants by phone, either through a VOIP call or by connecting the user's phone line with the merchant's line through the PSTN (publically switched telephone network). Although the UI (user interface) was simple, extensive backend rules were implemented to make sure the service was secure and that there was no possibility for users or merchants to be spammed.

This team had twelve members based in Redmond, four in India and three in Russia. We interfaced with two internal partners and two external partners.

The offshore development team, which consisted of a lead and seven developers, worked for a small company in Russia. The vendor relationship with this company had been obtained along with the acquisition of a company called Teleo. All of the Microsoft developers on the two teams had joined through the Teleo acquisition as well. The Teleo development team had been successfully working in a geographically distributed model for several years.

The Indian test team, which initially consisted of twelve people across both projects, was brought on board because there were no other test resources. Qualified testers are scarce, and there were few internal headcounts available to hire against. By the time the offshore test team was in place, the rest of the development team had been working on the services for four months and the target date for the first beta was only two-months out. Test was already considerably behind.

Challenges of Implementing Agile in a Distributed Team

Certain facets of the Agile methodology make it challenging to implement successfully in a distributed team environment. Some of the challenges we faced included:

Information Flow

One of the tenets of Agile is lean documentation. Things are only documented when there is a clear need and purpose for that documentation. When we started doing Agile, we found that the lack of detailed development specifications caused problems for the offshore team. A lot of detailed information was shared during our four-hour long planning meetings. Because of time zone differences, it was impractical for the offshore teams to engage in these planning meetings. The onshore team was too busy to relay all the detailed information to the offshore teams, leaving the offshore teams confused and ineffective.

Technical Communication Difficulties

In addition to the logistical problems with communication, there were other more basic issues. Most of the interactions with the offshore team took place over the phone. Initially we had a very difficult time understanding what the offshore team was saying. The voice quality of phone calls was often very bad. The offshore teams had unfamiliar accents and speech patterns. There were cultural and expectation differences which sometimes made it difficult for us to understand each other.

Round-the-Clock Work?

The differences in time zones were a big problem at first. Our main method of communication was email, and it would sometimes take days to resolve blocking issues because twelve hours passed between each question and answer.

If the onshore team did not complete something or get some piece of information to the offshore team by the end of our day, we would delay progress by a full 24 hours.

No Shared Vision

Partly because the teams were offshore, and partly because they were vendor teams, we found it difficult to share the business reasons behind the project activities. The offshore team was often working in the dark without the knowledge they needed to make decisions independently. Some of the tasks we asked them to do made no sense to them.

No Opportunities for Socializing

Trusting relationships are the foundation of team productivity. One of the best ways to build these relationships is through casual interaction, sharing details about our lives that let people know who we are and why we do what we do. It was impossible to get together with the offshore team in a non-business setting, therefore these relationships did not get built.

Additional Challenges

In addition to the above challenges, which are characteristic of a distributed team, we had other difficulties to overcome.

Initial Lack of Support for Offshore Test Team

I was hired as the first Microsoft test person on the team. When I was brought on board, the twelve-member Indian test team had already been working for a month, but they had accomplished almost nothing. They had no access to the Microsoft environment and they had spent their energies trying to build their own lab to mimic our facilities, but the environment was complicated and difficult to setup. After a month of frustration, they still did not have a working test environment and we were only five weeks away from our first scheduled beta release. Not a single bug had been filed.

The Microsoft contact for the Indian test team was a very busy developer who did not have the disposition or desire to help them. He was frustrated that the offshore team was demanding so much of his time, and the offshore team was stymied by his terse answers and lack of follow-through.

Lack of Access to Resources:

Initially, neither the Indian testers nor the Russian developers had access to the Microsoft corporate network. This created a huge overhead for our team, as someone had to send them files of everything they needed to work on, and the onshore team was required to spend a lot of time merging the Russian development teams' code changes into our code base.

Ineffective Onsite Coordinator

The plan for the Indian test team from the beginning called for an 'onsite coordinator,' a member of the Indian vendor team who could act as the liaison between the offshore test team and the onshore team. However, visa issues prevented the chosen onsite coordinator from being able to come to Redmond. (He shared his name, Amit Kumar, with millions of other Indians, making the visa process very time consuming. Amit Kumar is the Indian equivalent of Mike Smith.)

The delay in having an onsite coordinator caused considerable frustration for both teams, so the vendor company sent an alternate. Unfortunately, the person they sent did not have the temperament to deal with the strong-willed people on the Microsoft team and he was lacking in basic troubleshooting skills and systems knowledge. He could code, but he didn't know anything about SQL administration or Web Services, and he lacked the ability to unblock himself when he became stuck.

Communication Channels Not Yet Established

When I started, there were two half-hour meetings each week with the offshore test team, one for each project. These meetings were ineffective because of the difficulties in understanding each other and the need for them to 'try out' our suggested solution before they could come back and ask more questions.

Initially, even after the offshore team was set up with Microsoft email accounts, they were not in the habit of using them and would miss important emails about the project.

Little Visibility into What Offshore Test Team was Doing

I had twelve unpronounceable names reporting to me on the organization chart, but there was little visible productivity from the offshore team. This is particularly a problem for test, which traditionally is difficult to measure.

Lack of Trust

Going along with the lack of visibility into what the offshore team was doing, came a lack of trust. I was under a fair amount of pressure to make sure the services were tested and ready for the rapidly approaching beta dates. The communication difficulties, cultural differences, and lack of a real relationship with the offshore team, led me to mistrust them. I wondered if they were even honest. The list of people working for them included 'Amit Kumar' and 'Kumar Amit.' I thought they were trying to pull the wool over my eyes, but I later found out these really were two different people. I also questioned whether they knew what they were doing. Most of the documents they sent were not what I was expecting. I've since learned that their documentation was done in a very typical Indian style, but my initial conclusion was that they did not understand test at all, and were trying to cover up their lack of expertise with volumes of fluff.

Best Practices for Distributed Agile

We found solutions to most of the problems we encountered. In this section, I'll highlight our learnings.

Build Offshore Team Incrementally

Don't staff up until you're ready to support an offshore team. When you have someone onboard who is willing and able to support an offshore effort, you can bring offshore workers on effectively. However, if there is work that needs to be completed before all team members can be productive, such as building a lab environment, bring only a few people onboard at first, and only add the rest of the team when you are able to support them and keep them productively busy.

Bring Offshore Onshore

The offshore test teams did not start to show results until we had good technical communicators from offshore come to Redmond to become dedicated conduits of information for the offshore team.

After about two months of working with the offshore test team, I was fed up. The onsite coordinator they sent was ineffective. The offshore team was finding few bugs and I had little confidence in their test results. The documentation they sent was subpar. I was ready to sever our relationship and start over with a new vendor team.

Due to earlier complaints, the vendor company with which we were working had already sent a second onsite coordinator to assist the first one they sent. She was a little better at communications, but she didn't know test. We sent the first onsite coordinator back to India, and the second followed shortly thereafter for personal reasons (family emergency).

I set up a face-to-face meeting with our vendor representatives in Seattle and we laid out our grievances, rather bluntly. They listened to us, thanked us for being candid and promised to try their best to set things right. We were skeptical, but willing to give them another try, especially since our beta date was fast approaching and we knew that starting the process of engaging a new vendor would take time. The two projects were different enough that I realized we needed a dedicated resource for each project to handle the communications with offshore, someone who could do a deep technical dive and really understand what we were doing,

At this point in time, Amit Kumar's visa finally came through and he arrived in Seattle, followed shortly by Amit Sharma. Both of these men were the test leads from the two offshore teams. Although the initial contract had only called for one person from offshore to be housed in Redmond, our vendor company did not charge us for the additional onsite resource. They wanted our business and were willing to eat the extra expenses to make things right for us. Both of the people they sent were bright, hard-working and technical. They set up a system whereby they called the offshore teams every night and spent a half hour or more going over the day's events and making sure the offshore teams knew what was expected of them.

The offshore teams summarized their days' work and sent it via email to the onsite coordinators, who acted as the offshore teams' proxies at the scrum meetings.

The most important thing we did to make the project successful was bringing dedicated technical resources onsite for each project. These people already had relationships with the offshore team members and were able to build relationships with the Microsoft teams. Both of these men were technical enough to make the offshore team understand the complex details of the software under development.

Build Relationships

Another thing that happened about the same time that helped with my acceptance of our vendor company was that their offshore project manager came to Seattle for a visit. I had spoken with her many times over the phone, but my impression of her changed considerably when I met her in person. Over the phone, long distance, with her unfamiliar accent and unusual verbiage, I assumed she was young and inexperienced. Once I met her in person, I realized she was a competent, accomplished professional with goals similar to mine. If fact, we have built the roots of a relationship that will probably outlast our professional association.

Throughout the lives of these projects, we maintained onsite representatives from offshore. These typically switched out every three months (we insisted on telephone interviewing the replacements to avoid the first couple of disasters). Because our relationship lasted for a number of years, most of the offshore team members had the opportunity work onsite. They came to understand our culture and 'how work gets done' at Microsoft. We learned that the Indian team overall was smart, honest and hard-working, and they brought their positive impressions of us back to India with them as well.

Unfortunately, due to budget constraints, the Microsoft team was unable to visit the team offshore, however, I think it would have given us a deeper understanding of and appreciation for our offshore workers, and it is something that I highly recommend.

Multiple Communication Channels

Establishing multiple communication channels was very helpful in fostering productivity. Much of our communication was done via email; however, many team members would send Instant Messages (IMs) to each other with questions if they found their counterparts online. We had regular telephone meetings, and occasionally scheduled video conferences as well. Sometimes we would take photos of whiteboard diagrams to send to our offshore teams.

Cultural and Time Zone Awareness

About three months into the projects, I hired three Microsoft testers for our teams. Although I did not specifically look for Indians, two of the three people I hired were Indian. By chance, one of our developers was Russian and we also hired a Russian Program Manager (PM). These folks were able to communicate effectively with the offshore teams, and to explain cultural nuances to the onsite team members.

The time-zone differences became less of a problem once everyone got into the habit of making sure their work was ready for hand-off by the end of the day. Simple awareness of the time zones made a big difference. The overlap in time zones between the Russian developers and the Indian testers was a bonus. They often instant-messaged back and forth with each other.

Common Code Base / Document Repository / Tools

Having a common code base and document repository was essential to making the offshore teams effective. It took a very long time to get Microsoft corporate network access for the Russian developers, but our new Russian PM made this happen shortly after he came onboard.

The problem of lack of visibility was solved by having the offshore team members enter their hours directly into our scrum tool. Every day we could see exactly who spent their time doing what, and along with tangible evidence of their productivity such as test cases, bug reports, test automation, and metrics reporting, my confidence in the offshore team increased dramatically.

Partners Must be Flexible

If our vendor company had not been so willing to go the extra mile for us, we would not have been successful. The vendor company must be willing to try new processes and procedures and not be too dedicated to one way of doing business.

Allow Time for Processes to Mature

Assume that it will take several months for problems to surface and solutions to be found. Plan time in the schedule for communication channels to be worked out and for working processes to be put in place. Don't assume instant productivity and a smooth transition for an offshore team.

Open, Honest Communication

We were blunt in our communications with the vendors. We told them our grievances without sugarcoating them. We encouraged feedback from them as well. Both vendor teams were receptive and responsive to our complaints.

Other Best Practices

In addition to the problems we encountered in working in a distributed Agile model, there were other problems we ran into that are likely to crop up even with co-located teams. I will not go into the details of the problems, but I would like to highlight a few additional best practices that we discovered.

Continuous Integration

Once we finally got both the offshore teams connected to the Microsoft corporate network and we invested in infrastructure so that we could automatically build, deploy to a test environment, and run build verification tests (BVTs), our velocity increased dramatically. Before releases, we built twice a day. The morning build would include all of the Russian developers' check-ins from the night before and would get tested by the team in Redmond. The evening build would include the check-ins done during the day by the Redmond team, and would get tested by the offshore team over night.

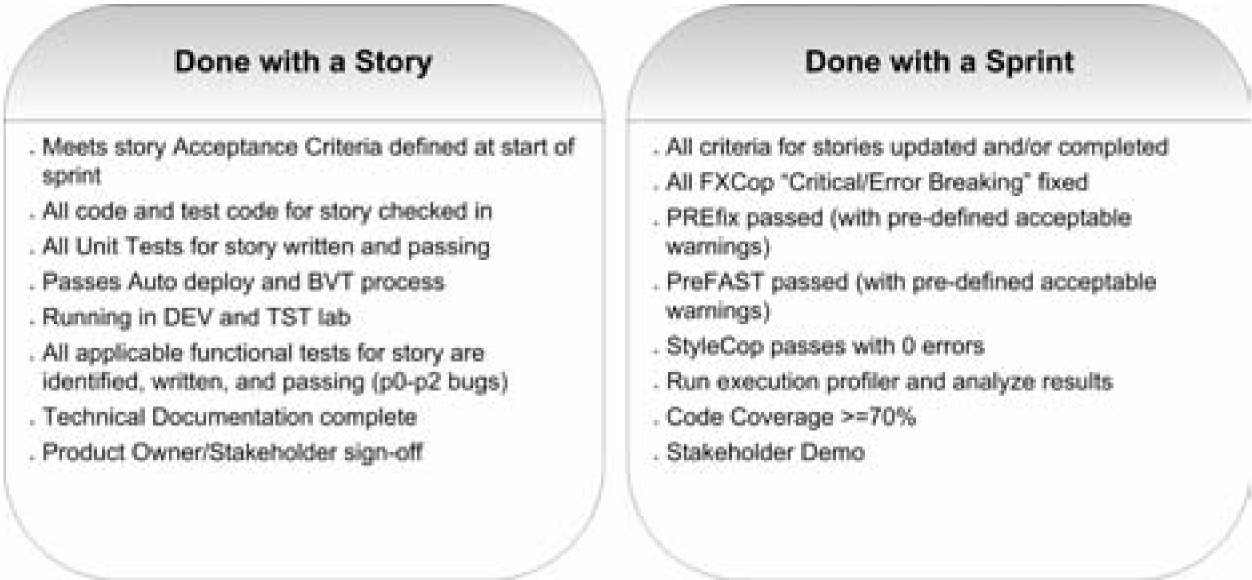
Continuous integration allowed us to create and maintain a very stable code base, making both new feature development and subsequent releases much easier.

Understanding 'Done'

Our team struggled with the meaning of 'Done.' Early on, if the developer declared the code finished, the story was deemed complete. However, we then found ourselves continuing to work on tasks related to the same story into the next couple of sprints. It seemed impossible to both code and test a feature completely within the same sprint.

Our velocity increased, which helped, but the main thing that brought us to the point where we could finish a complete story within a single sprint was getting really clear on the done criteria for the story, and making sure that all of the attendant tasks would fit within the sprint. We defined our done criteria (acceptance criteria) within our sprint tool and our PM Lead became very particular about making sure the criteria were complete and detailed for every story on the first day of every sprint. Of necessity, we had to break our features up into smaller pieces so that the entire development cycle could happen in a two-week timeframe.

Defined 'Done' Criteria



Done with a Story

- . Meets story Acceptance Criteria defined at start of sprint
- . All code and test code for story checked in
- . All Unit Tests for story written and passing
- . Passes Auto deploy and BVT process
- . Running in DEV and TST lab
- . All applicable functional tests for story are identified, written, and passing (p0-p2 bugs)
- . Technical Documentation complete
- . Product Owner/Stakeholder sign-off

Done with a Sprint

- . All criteria for stories updated and/or completed
- . All FXCop "Critical/Error Breaking" fixed
- . PREfix passed (with pre-defined acceptable warnings)
- . PreFAST passed (with pre-defined acceptable warnings)
- . StyleCop passes with 0 errors
- . Run execution profiler and analyze results
- . Code Coverage >=70%
- . Stakeholder Demo

Notes: P0-P2 refer to bug priority, P0 meaning 'fix immediately.' FXCop, PREfix, PreFAST and StyleCop are all internal Microsoft tools which check for code and security flaws.

Continuous Improvement

Retrospectives: Early on in our adoption of Agile, our retrospectives were hour-long meetings that yielded volumes of suggestions for improvements. Some items, such as 'there was not enough time to finish testing,' appeared sprint after sprint. As time went on and our processes matured, we had fewer and fewer things to discuss at the retrospective. Our retrospective shrank to fifteen minutes, and we usually did them at the very beginning of our sprint planning session rather than in a separate meeting. Because the team was committed to continuous improvement, we never abandoned the retrospectives, and even once things were going very well, there were sometimes great improvement suggestions that came out of the retrospectives.

Investing in infrastructure and process: Early on in the formation of our team, it was very difficult to make forward progress on software development. It seemed that for every hour spent productively, at least two hours needed to be spent on support tasks. We realized that we needed to invest in automation and process improvements in order to cut our overhead time. We took this back to our Product Owner and convinced her that we needed to invest some of our energies in infrastructure otherwise we'd never be able to reach the productivity levels that she wanted. Some of the investments we made were:

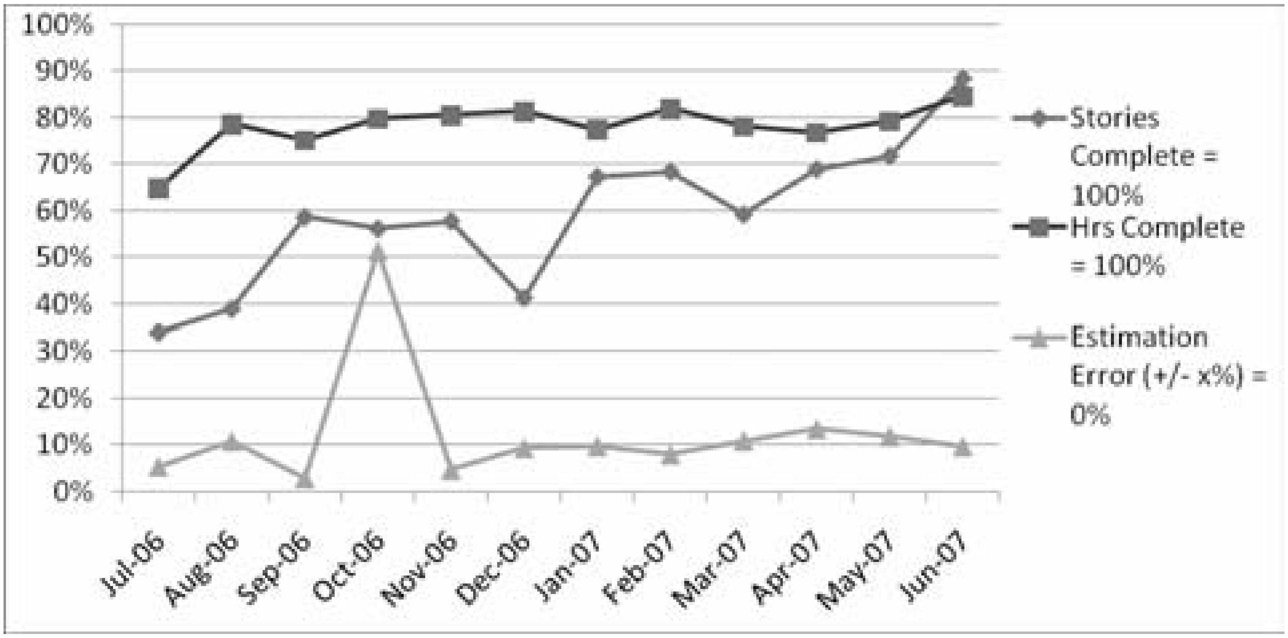
- Building an automatic build/deploy/BVT system
- Building emulators for all of our external dependencies
- Investing in lab infrastructure to increase our flexibility and minimize resource contentions
- Hours spent in meetings hammering out what 'Done' meant
- Hours spent in meetings discussing process improvements

Metrics for tracking progress: Our Product Owner believed strongly in tracking progress through metrics, and most of the team agreed with her. We defined a basic set of metrics to track and report. Some metrics were reported for every sprint, some for every month, some for every release, and some we only tracked quarterly. For each release metric, we set a minimum bar that we needed to meet before we could release, as well as a goal metric for which to strive. We began to see improvements in the metrics almost immediately and this lent us motivation to try even harder.

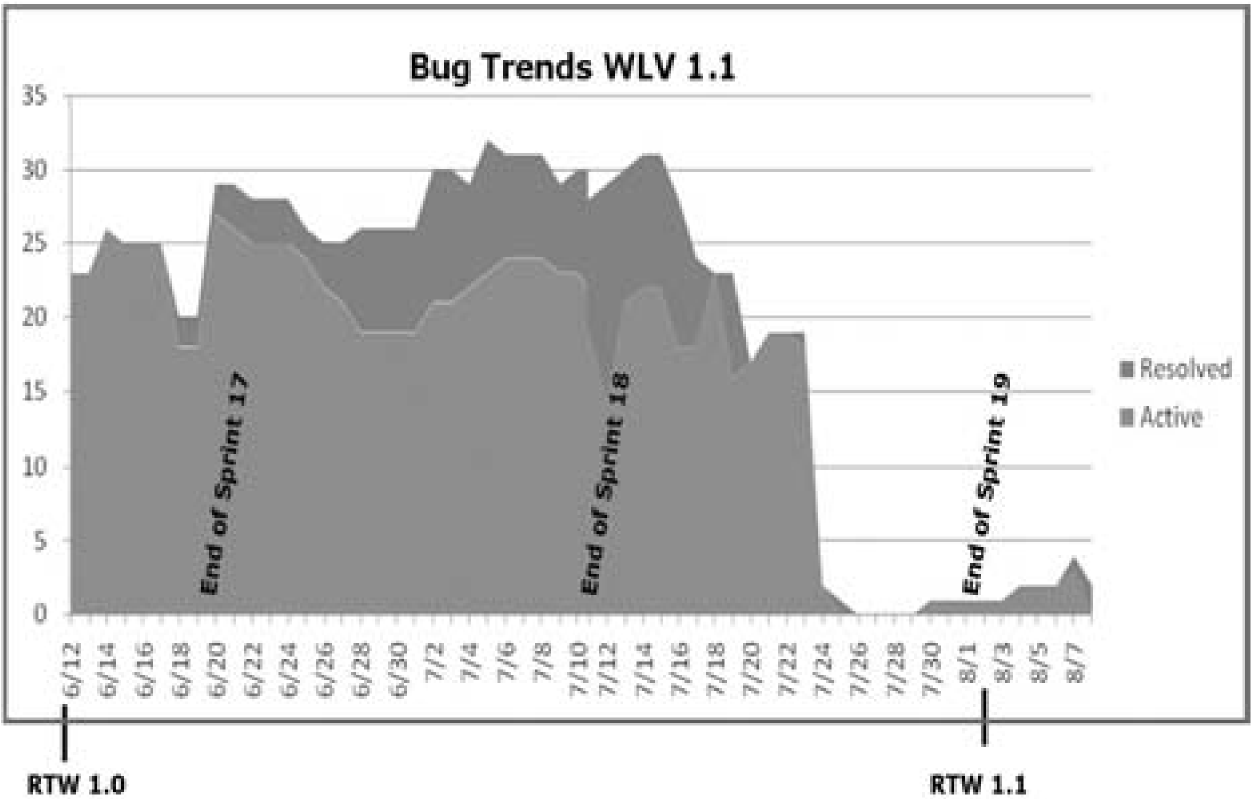
Some of the metrics we tracked included:

- Bug metrics: bugs found per release, resolution of bug, priority of bug, feature area of bug, etc.
- Code coverage
- Test automation percentages
- Test results
- Code complexity
- Unit test coverage
- Percent of stories completed in each sprint
- Percent of committed hours completed in each sprint
- Customer metrics: % up-time for the services, % positive feedback from customers, average latency of transactions, etc.

Sprint Estimation Metrics

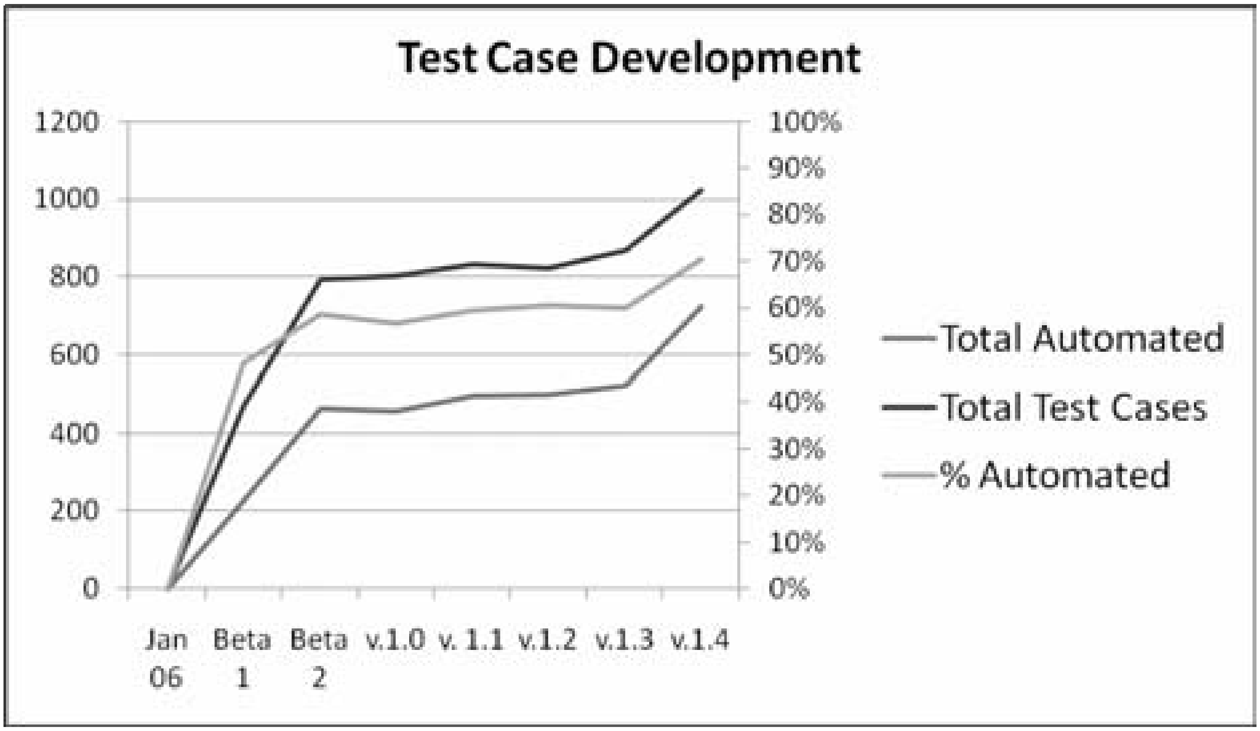


Bug Statistics



Note: RTW means ‘Release to Web’ (or, sometimes, ‘Release to World’)

Test Automation Statistics



Summary

Agile solves many of the problems associated with the Waterfall model. It incorporates requirements change as part of the model, and it leads to more dependable schedules and higher quality releases. There are many benefits to Agile in general.

But what about Agile offshore? Does it really make sense for a geographically distributed team to adopt Agile processes? We found that doing Agile in a distributed environment is difficult, but so is doing Waterfall in a distributed environment. Waterfall relies on detailed requirements and specs, but documentation is not generally a good method for communication. Once you solve the communication and relationship problems, working in an Agile fashion on a distributed team can be very successful.

I'll summarize what we learned using Agile with Scrum in a distributed development environment with the following four points:

- Have patience—expect several months of chaos before relationships are built and processes evolve to create a smoothly working team.
- Choose a flexible partner—your offshore counterpart should be willing to work the way you want to work.
- Plan for onsite rotation—set up a system whereby your team members work at the offshore facility and/or offshore team members work at your onsite location. Rotate team members out occasionally so everyone gets a chance to know the offshore team.

- Build relationships—the heart of a solid team is trust. Building a solid working relationship will provide a foundation for you to work through issues as they arise and maintain mutual respect for each other.

We made tremendous improvements in our processes and productivity over the course of a year-and-a-half. After our initial betas came in months late, we were finally able to reach a cadence of releasing updates to each service every two months, so effectively every month we had a release.

Releases became easier as we gained more experience doing them. Our partners came to believe us when we gave them a release date, and also to have confidence in the quality of our systems.

The offshore teams truly became an extension of core team. Due to a hiring freeze and turnover, our team ended up being 80% vendor/contractor resources, and only 20% full-time Microsoft resources. We were able to work very effectively with this model.

Our metrics continued to improve, and the VP of our group began pointing to our team as the 'poster child' for productivity.

Early on when we were struggling so hard to get a release out, we would work evenings and sometimes weekends. As we made improvements that increased our productivity, we found we could accomplish considerably more in less time. Once we reached the cadence of releasing every two months, we never needed to work extra hours. The team was happy and relaxed, and we all enjoyed working together in the Agile model. Best of all, we felt good about our contribution.

Endnotes

[1] Jepsen, Ole "Agile Meets Offshore," *Agile2006 Conference,* Minneapolis, MN, Handout, p.3.

Case Study: Fostering Meaningful Change with the Large Format Printer Division at HP

Carolina Altafulla (carolinaaltafulla@gmail.com)

Jim Brosseau (jim.brosseau@clarrus.com)

Abstract

Sustaining meaningful, strategic change in any organization can be difficult, particularly when there are strong personalities and established practices in place. Most initiatives are either too broad in their changes, or fail to address the needs of participants.

Within the Large Format Printer Division at Hewlett-Packard in Barcelona, despite some of the strongest front-end analysis practices in the industry, projects continued to face delivery challenges similar to those in many other companies.

Management decided to take action and revert the trend in declining product development efficiency. The authors collaborated with the specific intent to provide lasting and meaningful change for the group.

Through a series of interview sessions with all of the key people in the team, we collected a wide range of different perspectives: current practices, pain points, and suggestions for improvement.

As the stories were collected, a broader picture began to emerge. This was a strong, disciplined engineering culture, with strong elements of practice in place that needed to be reinforced and leveraged for the needed improvements.

Rather than making recommendations that could easily be dismissed, we focused on collaborating with the teams to help them find a solution they could embrace. The resulting solution required more of a change in perception than behavior.

In deployment, we worked with management, marketing and engineering to help them understand the value of change in their terms, and the energy of the group grew quickly. As the information sessions progressed, there was greater demand for participation.

We describe the initial situation within the group, and the approach to selecting and implementing appropriate changes. We will reveal the deceptively simple viewpoint that was the core of the changes, the effect on the culture and project results that have taken place since.

Biographies

Jim Brosseau

jim.brosseau@clarrus.com

Jim has been in the software industry since 1980, in a variety of roles and responsibilities. He has worked in the QA role, and has acted as team lead, project manager, and director. Jim has wide experience project management, software estimation, quality assurance and testing, peer reviews, and requirements management. He has provided training and mentoring in all these areas, both publicly and onsite, with clients on three continents.

He has published numerous technical articles, and has presented at major conferences and local professional associations. Addison-Wesley Professional published his first book, **Software Teamwork: Taking Ownership for Success**, in 2007. Jim lives with his wife and two children in Vancouver.

Carolina Altafulla

carolinaaltafulla@gmail.com

Natural from Barcelona, Spain, Carolina has been in the technology industry since 1992, first in medical devices and later in the printing industry. She has worked in new product development as development and as QA engineer; she has also acted as team lead, and initiative manager for international and multidisciplinary projects.

As a matter of fact Carolina speaks fluently 6 languages.

Carolina has recently relocated with HP to the USA to broader her experience in Program management for new product development in the printing industry.

Carolina lives with her husband and 4 children in San Diego, California.

Case Study: Fostering Meaningful Change with the Large Format Printer Division at HP

Carolina Altafulla

Jim Brosseau

Sustaining meaningful, strategic change in any organization can be difficult, particularly when there are strong personalities and established practices in place. Most initiatives are either too broad in their changes, or fail to address the needs of participants.

Within the Large Format Printer Division at Hewlett-Packard in Barcelona, despite some of the strongest front-end analysis practices in the industry, projects continued to face delivery challenges similar to those in many other companies.

This case study is presented from two perspectives: inside the business (Carolina) and outside, from an external consultant's viewpoint (Jim). We go back and forth between the inside and outside perspective, chronologically through the engagement. Both perspectives help us understand the complete picture, and we will see that only when the perspectives are brought together in a team environment do we see the complete value of the engagement.

Outside Perspective: Overall Industry Context

Change is tough on anyone. We'll fight tooth and nail to avoid change, even if our current situation is untenable. Dealing with this human barrier is the key to driving effective change.

As consultants, there is the old story that we can identify the top three issues that clients face before we even talk to them. First, they will have challenges around how they define and manage the scope of what they want to build. Secondly, because of this first challenge, they will have deficiencies in estimation and planning, and many schedules are driven primarily by time constraints than any credible understanding of how long it will take. Thirdly, because of the first two, the teams will not have the infrastructure for effective change management.

That's the easy part. The tough part is to understand how each of these three issues manifests themselves for each client. While many organizations are finding value in adopting agile approaches, they are acknowledging these challenges and addressing them by embracing a dynamic environment.

For some organizations and some products, though, there is an opportunity to perform a deep analysis of the customer problem up front, and use this as a basis for developing a product. This turned out to be one of those places.

Inside Perspective: The Existing Situation

The balance is easy to break and difficult to keep. The balance in a mature organization is essential to most of its members. Break it and you will be in trouble.

Changing an organization and not impacting that balance was something that worried both the management and myself (Carolina).

Before this initiative, engineers would attend meetings to try to catch up with overall project information and more specifically with the customer requirements that they had to deliver. That was easy as long as they only reported to a small or medium project, as the engineers were able to manage and filter all information about the whole project.

When the projects grew in complexity and number, for efficiency sake engineers became horizontal (delivering the same piece to different programs). They also felt more pressure for multiple and more rhythmic deliveries.

Program synchronization among the members became unmanageable as there were too many people needed in a room. Many of them had the sense of wasting their time.

Engineering teams began to be more complex, with more layers of hierarchy. We worked in a matrix model, assigning each engineer one program for primary responsibility, but also being the owner of deliverables for different programs.

What began to happen is that many requirements were discussed without enough criteria because the owner was not present.

Extra management layers were added and then engineers were diluted in the organization. Their access to reliable information was heavily reduced and the "theoretical" information channel was not working well.

The communication skills did not grow at the same pace as the organization and what happened is that there was missing information all over the place.

Nothing gets engineers more nervous than being out in the dark. And that began to happen a lot.

Outside Perspective: History With The Team, and Why Go Back?

I (Jim) originally worked with this Division in Barcelona several years earlier, with what can be called drive-by training. Back then, two jam-packed days of bullets of information hit the group, but nothing really came of the engagement. There was no opportunity to facilitate effective change of any kind. Combine this with the long flights over to Europe and back, and the time away from the family and I quickly decided that this sort of engagement wasn’t all that rewarding for me.

The same group called over a year later. They were now really interested in change. They had found an internal champion and a budget, and asked if I was interested in pitching in. Despite the past experience where I had sworn it just wasn’t worth the effort, I now jumped in with both feet.

One of the first things I look for with clients is the team dynamics. If we consciously understand the rules that govern our actions as well as those of our peers, we have harnessed a very powerful tool.

As with many large groups in this industry, this group demonstrated a wide variety of situations and responses. A diverse group will always bring a number of different players to the table, and it can be fascinating to step back and watch the moves the players make.

This particular group was dominated by what one could call a pack of alpha males: very bright and competent, and also very adept at working the system to help them achieve their goals. They had advanced within this system through leveraging influence, exhibiting behaviors that demonstrated their superiority, and at times maneuvering craftily through the political landscape. They leveraged their deep technical knowledge, but this was not their strong differentiating trait. They understood the game being played, played it well, and thrived in their environment. They played to win.

There were others that saw the game for what it was and chose not to get caught up in the machinations. They had found their niche, their comfort zone where they could gain satisfaction from a job well done despite the game going on about them. They were not as deeply impacted by their environment, and could perform well in a wide variety of different games – while they may not have come out as the dominant winner, they were also less likely to suffer deep losses. Consciously or not, they behaved in a manner that supported both themselves and others.

Finally, there were those that were frustrated by the games. They were often overwhelmed by the strong players, sometimes to the point of having to leave the game altogether, through choice or through exhaustion.

We clearly had a wide range of viewpoints that we needed to address with this engagement.

Inside Perspective: Why Did We Follow This Approach?

We heard the consequences; engineers were verbal about the issues. We had internal reviews where much of the feedback was complaints.

Identifying the causes could be a difficult work for the insiders as we are part of the problem and it is difficult to know what "we do not know".

Doing a good job about solving the problem in a balanced organization without breaking it looked huge. It was important to keep the "status quo" as it had to be part of the solution. It was also difficult to provide a different way of doing things to a group of colleagues that have worked together for a long time.

We attempted to look for the solution a couple of times without a real plan and a real effort towards it. Complaints kept on coming, so a group of senior managers decided to give change another chance. I (Carolina) was appointed, and I actually did get some resources.

Being supported by management, I decided to give this a different approach. I went for a deep change.

Right from the beginning I had in my mind to do extensive research on what was going on, with a detailed design of what should be happening instead. I knew from the first minute that I needed external help that no insider could give me.

So I looked for the best available help in the industry and that led me to Jim, who guided me through the research, which went from informal to more formal. We got data to compare to and he told me where to look.

Outside Perspective: What We Found in Discussions

The group certainly had what would appear to be all the classic symptoms of trouble in the area of requirements. Projects were being significantly delayed and resources were cannibalized from other projects to pick up the pieces (hence delaying those projects, sometimes before they got started). Key resources on the team spent a great deal of their time answering the same questions from different people, and felt bogged down with their efforts. E-mail was the primary communication source for changes that occurred during the project (and we all know how effective E-mail can be for maintaining a clean audit trail).

With a little bit of poking around, though, they had some real strengths to build on. They had a great handle on the competitive landscape and set objectives for their projects based on specific quantified factors relative to their competition. They took the time to identify composite personas for their different user classes to ensure that they had a common understanding of their client and could reasonably gather the breadth of critical use cases. They even started to manage all their projects in terms of a broad strategic portfolio rather than unrelated projects with no leverage between them.

They spent a great deal of time up front defining the business requirements, and their practices were among the best I (Jim) have ever seen in the industry, yet they had all these symptoms of requirements problems.

Where was the challenge?

The challenge was in the minds and attitudes of the team members themselves. Most of them approached requirements as something to do on a project before you get started to actually build the product. They had two modes of operation. The engineers would start out chilling their heels with requirements work in the early stages when the pressure was low. At some point in the project, an imaginary switch would be turned on, and suddenly it became time to get busy.

At that point, all that requirements work would be cast aside; they were under pressure to get what they perceived to be the real work done. Build some prototypes, evolve them into the final system, and then switch over to fixing bugs as a means of getting the project completed. Changes were made on the fly in a scramble to get things done, and the overruns began. Eventually a product was shipped, and the whole cycle started again after management rewarded the heroes that saved the day on this challenged project.

The team needed to appreciate the true value that their early requirements work could provide them.

Here is some context for our decisions in Barcelona. We ran a diagnostic of current requirements practices with all participants to use as a baseline prior to our efforts (this was used, with permission, from Software Requirements, 2nd Edition (1), Appendix A). As with most diagnostics we have run, we found a broad range of responses, and the results were a great way to get everyone thinking about the breadth of issues to address.

We began to see some interesting trends in the data when we broke the twenty questions into four categories: deriving requirements from predecessor information; developing the requirements in a disciplined fashion; using them as a basis for successor products on the project; and effective change control. For all of the groups that we have worked with so far (this diagnosis has been performed with 25 different groups thus far, in a wide range of companies), they tend to score higher in the first two categories than in the last two.

It appeared that this group was not much different. Many companies put significant effort into developing their requirements, then fail to leverage their full value, either by not referring to them or failing to reasonably manage them throughout the lifecycle. Often, this leads to the perception that the whole requirements effort is a waste of time - is overhead work. The cycle can spiral out of control, and teams fall into the code-and-fix approach for getting their work done.

On the positive side of this challenge, we had the opportunity to frame potential changes in the context of work that is currently being done. All the effort in place provided powerful insight into the problem space and structure into defining an effective solution, so change in this context did not involve a great deal of additional work, which is usually the greatest barrier to change.

Indeed, in this situation, we needed to weigh the cost of everyone discovering and solving important issues independently (the status quo) against the cost of managing an infrastructure that allows people to actually find the information that has been collected up front (the better future). The arguments for change become much more compelling as we get people to recognize that the information exists, and we provide improved mechanisms for finding and maintaining this information.

The requirements gathering effort was fine here; it was the knowledge management that needed to be addressed, as well as the habits and perceptions surrounding the initial efforts on the project. We refined the problem from trying to put a reasonable requirements development process in place to helping people appreciate the value in the work they are already doing. This was still a challenge to resolve, but a much more precise area to focus on.

That work they were doing up front was the real effort to drive project success.

Inside Perspective: What People were Thinking

People had different expectations about the outcomes of our work, especially in the beginning when no communication was being done. These expectations could be classified in different ways.

There were people that thought that someone would become the middle man, who would have all the information from marketing. These people wanted someone to tell them what to do so they could focus on the “important” work, which is engineering delivery. They did not look at requirements work as part of the solution. This was a simplistic way to see things.

There was another group of people that wanted to have a say in the solution, in a confrontational way. They felt they had a lot of experience in different projects and knew what “others needed to do”. They had the need to know more and the need to feel they were part of the success, because they felt like heroes in many projects. They were used to being treated as key people. In the end, these were the easiest people to deal with.

There were the skeptics. Nothing we could do would help in any way. Our destiny is to keep on dealing with the same issues project after project, and fatalism was the key in their discourse. Those people had an engagement problem.

And of course, there were also other people with other opinions and attitudes as well.

Everyone thought there was a problem with requirements, but there were many descriptions of the problem and many suggestions for how to solve it. This is a very positive thing, as that eased our work a lot. We benefitted from other people’s experiences, and as everyone felt there was a problem, change was easier to implement.

During the process, those mindsets were evolving. At some point, we could feel a lot of people "thinking about the solution" and contributing by coming back to us and explaining "my solution" as if this was their idea. That part was fun and it was great to be part of such a change in people’s opinions.

Outside Perspective: The Changes We Introduced

We listened to a lot of people: how they worked, the pain they were feeling, the suggestions they had for making things better. It was clear that everyone experienced significant pain, but everyone had a very different sensation for this pain. With different perspectives, the perceived ramifications of the challenges were different as well. As we correlated our information, we started to see a picture developing. We had uncovered a root cause, which was the disconnect between the initial business requirements gathering and the subsequent implementation. We could finally start attacking the real problem.

We went back to the people in face-to-face meetings to discuss the findings, both the strengths and opportunities within the team. We started to help them paint a picture of what a better world might look like. To do this effectively with the different groups, we ended up painting a gallery of different pictures. One where there was predictable closure on projects, another where there was less disruption and chaos. We painted a picture where projects could happily co-exist without stealing resources and the group as a whole could leap ahead against the competition.

We walked them down the path of how to get to these places. We had carefully selected the path of least resistance. There was little proposed disruption to the way they did work in the past. Indeed what we suggested was an infrastructure that would allow them to do the things they wanted and needed to do in an unfettered manner.

We then ran a significant amount of...well, not quite training, more like acclimatization. We reinforced the concerns that the pain was universal, walked them through the root causes we were trying to excise and the changes that would allow that to happen. Everyone had the opportunity to express their challenges, and in seeing the broad range of perspectives, different groups started to bond. We worked with almost as many managers as we did technical staff to reinforce that they had the critical role of removing the roadblocks that could get in the way (including some of their disruptive behaviors from the past). As the changes were incorporated, we worked with their partners that were across the globe, creating more managed communication across all groups.

There was only one critical adjustment that we made, which was one of perception. As the group was quite comfortable driving their development efforts based on the burn-down of defects, we proposed the idea that *an unimplemented business requirement can be treated essentially the same as a defect*. Rather than cast aside that wonderful business analysis that was done, we suggested they take that information and seed their defect tracking system (enter these unfulfilled business requirements into the system as defects) before implementation started. This gave them a way of working in the same environment they were used to (low culture shock), but allowed them to trace back to the original requirements.

This was the only change; no additional “best-practices” were added to their workload.

There was universal support. Beyond engagement in the classroom, there was a buzz around the coffee machines that went beyond caffeine, and as the training progressed there was an increased demand. Lunchtime discussions were centered on looking forward to getting started, rather than 'anything but that training'. Criticisms were almost exclusively suggestions to go further with the implementation, but we were careful to ensure we could walk before we ran.

Compared to the reaction to the stock training eighteen months earlier, it was a different group. Except that it was actually the same group. In the earlier training, there was relatively little investment, and virtually no value gained. We were merely helping them spend their training budget. Now, there was significantly more investment, but there is also an overwhelming upside. This turned out to be a strong business decision.

The difference is that we didn’t lecture or ramble on about what they should be doing. We listened, we coached, we solicited input, we facilitated. We helped the team find their own better place, which was not that far from where they were in the first place.

Inside Perspective: Current Practices

When we first proposed our solution, we wanted to implement it in a controlled way for only one project, then extending process improvement later to other projects in their beginning stages.

That got in the way of some people who wanted to implement our designed solution more aggressively. They wanted the solution for everything now, without understanding that change needs to be controlled in a consistent way.

A quiet period was negotiated. After a few months of success in the pilot project where these changes were working, I (Carolina) had to incorporate the changes into six more projects within the organization. I also had to help build databases for horizontal components deliveries (project components that were very similar from project to project but differences between them needed to be well controlled).

As I could not control all details in all those projects, I needed help. As the projects had essentially different needs, different approaches were redesigned for each one. Those designs were not my responsibility anymore.

The requirements project left my hands before I was ready for it: as when you are raising children, they leave before you think they are ready.

Looking at it now with the distance and time, I think the project was ready. Everyone went through the training and was aware of what to do and what not to do. The proof is that complaints about requirements engineering dropped significantly in the list of engineering complaints.

All the requirements engineering activities that happened afterwards were improved, with more understanding and more suited for the organization.

Outside Perspective: The Challenge of Sustaining Change

Many companies that change the way they do business often find themselves falling back into their familiar old habits.

It's not a software team syndrome; it is part of the human condition to quickly lose what has not been conditioned and reinforced in favor of what is familiar, regardless of the consequences.

In Leading Strategic Change (2), Black and Gregersen suggest that change is a cycle as follows:

> *"Stage 1: Do the right thing and do it well.*
>
> *Stage 2: Discover that the right thing is now the wrong thing.*
>
> *Stage 3: Do the new right thing, but do it poorly at first.*
>
> *Stage 4: Eventually do the new right thing well."*

We start in a state where we are doing the right thing, and doing it well – the proverbial status quo.

Something happens (market conditions change, for example) and we find that while we are still doing the same thing, and doing it well, it is no longer the right thing to do. In software, I would modify this to suggest that while in the status quo state, some disastrous event or astute introspection highlights that we actually weren't doing the right thing in the first place, but we were oblivious to the problems.

We change our behavior, and initially, the new practices, although they are the right things to do in the new situation, are not being done well. In software, this is where we learn to see the value of appropriate application of "best-practices". For the situation in this study, this crucial step was minimized: we did not introduce any "new" practices, thus minimizing the learning curve.

It is at this point where we need to ensure that there is strategic continuity in what we do. We need to constantly reinforce our belief that the new behaviors are the right ones, through demonstration of our new successes, however small initially. Until we start to see those new successes, we need to focus even harder to ensure our efforts don't fall off the rails.

We need to strive to truly institutionalize our changed behaviors, while recognizing that they may be subject to that same disruptive change cycle in the future. More than demonstrating simply that we can do it, more than even showing the benefit, we need to get to a point where it becomes rote. When it becomes rote, we need to continue to reinforce the practice, to make this reinforcement part of the practice itself, part of our culture. We need to constantly recognize the positive efforts in the group, share the positive experiences as a standard part of doing business, or those improved practices are at risk of being neglected into oblivion.

It is an ongoing effort to avoid the tendency to backslide into our old habits.

Inside Perspective: What has Happened Since

Since I (Carolina) left, the ownership of the project has changed a few times. Very different people with very different opinions have had this ownership. Their ideas and design are quite different.

One thing that remains is the alignment in the organization that we created with our training. Another is the awareness of what is a good practice and what is not and what are the consequences of these.

It now looks like no one owns the initiative, but it is alive and embodied in the engineering mindset. The Project managers continue to grow the requirements database. There are more uses for our design and more and more projects are controlled.

For my current work, I need to contact the people I used to work with. I have access to their projects, and also get some information periodically to design new project requirements structures. I am amazed by the maturity and detail of some of the new projects Barcelona is creating.

I could say Barcelona is now a leading edge organization in Requirements engineering, without being aware of it.

Lessons Learned

Overall, this initiative was a success for three key reasons:

1. Everyone (management, marketing and engineering) was involved in the initiative, from the solicitation of their diverse perspectives through the understanding of how potential changes would affect each of them differently.
2. The proposed changes were extremely simple to implement (to interpret unimplemented business requirements as defects), leveraging off of the

group's existing strengths and carefully tying these strengths to existing behaviors that were already part of the culture.

3. The engineering team took ownership of requirements management, and the effort was carefully managed and sustained over time to ensure that these new behaviors truly took hold.

References

(1) Software Requirements, 2nd Edition, Karl Wiegers (Microsoft Press, 2003)

(2) Leading Strategic Change, J. Stewart Black and Hal B. Gregersen (Prentice Hall, 2003)

Collaboration Among Software Components

Dick Hamlet
Department of Computer Science
Portland State University
hamlet@cs.pdx.edu

Biographical Sketch

Dick Hamlet is Professor (emeritus) in the Department of Computer Science at Portland State University. He has worked as an operating-systems programmer and systems-programming manager for a commercial service bureau and for a university data-processing center. He was a member of the software engineering research group at the University of Maryland for 12 years, a visiting lecturer at University of Melbourne in 1982, and a Science Foundation Ireland fellow at National University of Ireland, Galway in 2003-4. He has been actively involved in theoretical program-testing research and in building testing tools for almost 40 years. He is the author of three textbooks and more than 50 refereed conference and journal publications. Dick has been involved with PNSQC since 1985, when he gave the keynote speech at the 3rd Conference. He has worked on program committees for many of the Conferences since then and he helped to invent the present system for soliciting and selecting technical papers.

Keywords: Software components, design decomposition, testability, synthesis of properties, CAD tools

Abstract

Software components are executable programs created without knowledge of how they may later be assembled into systems. They are therefore an ideal setting for analysis of unit- vs. system testing. Using a suite of prototype CAD tools for component-based analysis and synthesis, the behavior of components and systems can be measured and predicted. Examples are presented to illustrate the pitfalls of testing component-based designs. Two questions are investigated: (1) To what extent can the results of component testing predict the results of system testing? (2) What component- and system designs minimize surprises that emerge when systems are assembled and tested? Several design rules are presented that will improve the accuracy of the tool predictions. When accurate predictions can be made, a new method of system development is suggested.

1 Introduction

Software components are independent executable programs developed in isolation, but intended to be combined into a variety of systems. Blackbox components communicate entirely through well defined interfaces, which describe their stand-alone inputs and outputs. A component may have local persistent state, but does not share state with any other component, nor may there be any global system state. These strong constraints make it possible to implement and test components without knowing their eventual application, and to design and test systems without access to the component source code. However, within the constraints some components and some systems are much better than others, as judged by how easy the designs are to understand and to test. It is not at all obvious how different component structures and allocation of state among them influence quality of a system made from those components.

The intuition behind good quality components and good quality systems design is that component testing in isolation should allow prediction of system properties. In good designs, component tests capture the component behavior; then those tests can be combined to predict system behavior with enough accuracy to be useful in understanding the system and seeing if it meets its specification.

In this paper, component-synthesis tools (described in Section 2) are used to experiment with testing of components and systems. Simple case studies suggest some guidelines (Section 5) for defining components and designing systems whose quality can be more easily tested. Cases will be exhibited in which:

1. Series composition of components leads to unpredictable system behavior because of the pattern of discontinuities in each. (Section 3)

2. Apparently meaningful component and system states are in actuality impossible to enter. (Section 4)

Sometimes an improved design results from combining several components into one. This moves the burden of testing the composite from the system level where there are always too many test scenarios to the component level where testing is easier to control. The lesson for the component designer is not to go too far in dividing functionality. Sometimes it is systematic testing of states that misleads the tester; random testing using input sequences may be better.

When the tools described make accurate system predictions, their calculations can replace execution of a system assembled from components. The tool predictions are very fast, allowing the system designer to try more components in different configurations, 'on paper' instead of by physical trial and error.

2 Component Analysis and Synthesis Tools

A collection of software tools was written over the course of the last six years to experiment with component testing and synthesis. Analysis tools for the component level are straightforward. Given executable code for a component, the tools allow it to be executed in several systematic ways (that is, tested) and the test results displayed. The analysis is based on a division of the component input domain into subdomains. These subdomains are provided along with the executable code, and represent the programmer's best guess at input subsets on which it is useful to try the code. On each subdomain, the component is executed a number of times, either by systematically sampling the subdomain, or by collecting samples from domain-wide random sampling that happen to fall in the subdomain. The execution results are averaged across the subdomain, and this yields an approximation to the component's execution behavior. The tools compute the r-m-s deviation of these subdomain averages from the actual behavior samples, which is a measure of how well the subdomains capture all possible component executions. Both positive and negative deviations accumulate without cancellation in the r-m-s error measure.

Subdomain testing from a specification (which is usually called 'functional testing' or 'specification-based testing') is the primary method used in the testing phase of software development. If the averages measured in subdomains are a close approximation to what the actual code does (because the subdomains are well chosen to capture different aspects of its execution), then intuitively the subdomain test is a good one. However, our component tools use subdomain testing for a quite different reason. Dave Mason and Denise Woit recognized in the late 1990s that subdomain-defined approximations to the behavior of code can be the basis for a composition theory of components [7]. By subdividing a component's behavior into subdomain fragments, it is possible to calculate how the fragments of one component will feed into the fragments of another, and thus make an approximate prediction of what the two components would do if placed in series. In particular, subdomain analysis partially solves the difficult technical problem of mismatch between tests of two components: in haphazard testing the outputs obtained from a component-A test may not match inputs tested independently for component B, so that predictions for the $A-B$ series system cannot be made from isolated tests of the two components. But if both A and B have tests that cover all subdomains, then any A-input subdomain must produce outputs that fall in some B-input subdomain, so the tests always match up.

Although subdomain decomposition of testing always matches A test outputs to B test inputs, there is more to component composition. For independent test results to combine accurately, the distribution of A's outputs over B's input domain must match the way test points were selected in testing B. These distributions are called *profiles*, which are probability density functions that describe how likely it is for any particular point to appear. Accurate composition of test results requires that the output profile of A (from testing A) match the input profile used to test B. In terms of subdomains, if inputs in a B subdomain are equally likely to be tested (a *uniform input profile* by subdomain), then A's test output profile must also be uniform. That means not only that the spread across each B subdomain must be uniform, but that each B subdomain must receive the same number of values from A, just as it was when B was tested. Unfortunately, the profiles usually fail to match. In a sense, each subdomain now acts as did the original input domain, scattering values in a non-uniform way. We call this fundamental difficulty the *profile-matching problem.* Without using subdomains, tests do not match, and there can be no predictions of combined behavior. Using subdomains, predictions are always possible, but for badly matching profiles they are inaccurate.

Subdomain-based composition theory also applies to non-functional software properties like performance (run time) and to reliability. For example, to predict the run time of an $A - B$ series system on one of its subdomains D (which is also a subdomain of A), it is only necessary to add A's run time on D to B's run time on its subdomain to which D maps. The match-up of subdomains is again crucial, since B is not running on an arbitrary input, but only on one that A produces as output. Again, the profile-matching problem may affect the accuracy of predictions for non-functional properties.

In 2001, Mason, Woit, and Hamlet developed this theory to predict system reliability from measured reliabilities of stateless components that form a system [5]. They used the system synthesis operations of sequence (series), conditional, and iteration. Shortly after, the first experiments were conducted with real code and the stateless versions of synthesis tools developed. The tools were extended to components with local state starting in 2005. Currently, versions of the tools that process a change in subroutine definition incrementally, without re-measuring or re-calculating for subdomains that have not changed, are under development. The tool collection has a number of features not of immediate interest here, notably the ability to include non-functional behavior in its measurements and predictions. Complete documentation, tutorial examples, and the tools themselves are freely available on the Internet [3].

Implementation of novel and sophisticated software tools is a process of carefully choosing what programming languages and systems will be allowed in the software to which those tools apply. This implementation effort was done by a small team with ever-changing student members, many of them high school students working under a summer internship program. It was therefore decided to impose severe restrictions as follows:

- The tools run on Linux systems, making heavy use of the GNU utilities, particularly shells and file manipulation. The tools are written in Perl. It would not be difficult to port to another UNIX system, but moving to a different operating system not intended for code development (like Microsoft Windows) was never imagined.
- Components analyzed and synthesized may be written in any language, and are presented to the system as compiled executable code files.
- Each component must observe the following restrictions on its input-output behavior:
  - There must be a single floating-point input value read by the component during its execution.
  - There must be a single floating-point output value written on each execution.
  - A single non-functional parameter with a floating-point value (e.g., run time) is measured and output on each execution.

  - Persistent local state is kept in a specially named permanent disk file, and consists of a single floating-point value. Initialization occurs when this file is absent at the beginning of execution and the component creates it; thereafter an input state is read from the file at the beginning of execution and a result state written at the end of execution.

- Systems are formed using only a pipe-and-filter architecture with sequence, conditional, and iteration.

These restrictions are severe limitations that would disqualify most software that is intuitively a 'component.' In particular, code is seldom written with a tidy 'read input, read state, compute, write output, write state' pattern. The model does not allow for concurrency, with its many ways of communication and synchronization among components. The restrictions are justified primarily because they enabled a complete tool set to be written and debugged by a small team in a few years, a tool set that is useful in experimenting with components. The choices can be partially justified on other grounds. Numerical values allow experimentation with random testing, an important research topic. Single values do not seem a fundamental restriction, since in principle it is possible to code several values into one. For the given component restrictions, the pipe-and-filter architecture covers all possibilities.

3 Emergent System Properties

A good way to introduce the tools described in Section 2 is to consider two identical stateless software components CS connected to form a series system. In Fig. 1, each copy of CS has its black-box input-output behavior (at the left), as does the system combination (at the right). On a UNIX platform, the series system could be executed using a pipe: `CS | CS`.

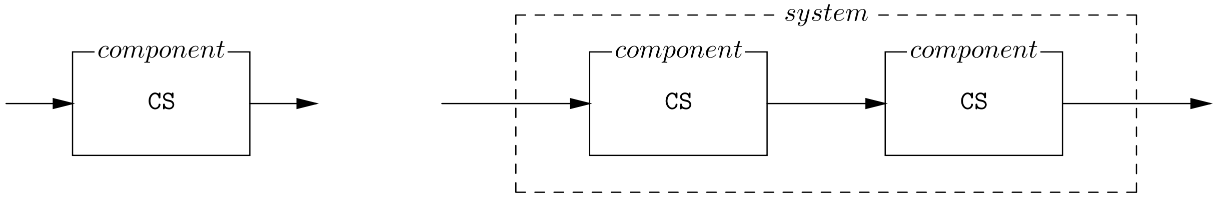


Figure 1: A component CS in series with itself

Fig. 2 shows behavior graphs of a particular CS measured by the prototype tools; this example component computes a saw-tooth function with an envelope that is an inverted parabola. The tools are used to test CS on the interval [0,10). The left graph in Fig. 2 shows this function measured by executing the component

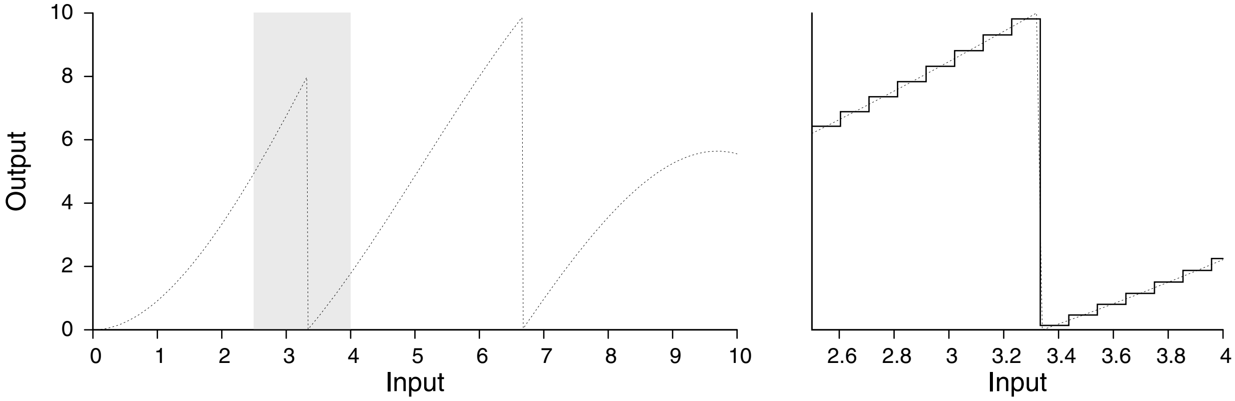


Figure 2: Component CS behavior measured and approximated by subdomain testing

over [0,10). The right graph in Fig. 2 reproduces the shaded region of the measurement (smooth curve) and superimposes (step function) values obtained from averaging over samples in 96 equi-spaced test intervals (subdomains). These intervals are the width of the 'steps.' The test averages capture the behavior of `CS` to within an average r-m-s error of about 1.9%. Testing is better in some subdomains than others; the r-m-s errors range from about 0.1% to 3.1%.

From these component test results, the tools can calculate a prediction for testing the series system. Fig. 3 shows the actual system measurements on [0,10) at the left, and at the right the superimposed prediction on the shaded region. The average r-m-s error in the predictions is about 9.8%. The r-m-s prediction error is below 10% in most subintervals, but is over 100% in five subintervals, e.g., 125% error in [2.92,3.02), the left 'valley' at the right of Fig. 3. These errors are considerably larger than those for the component measurements, and the reasons are apparent by comparing the right sides of Figs. 2 and 3: Firstly, the functional behavior of the system changes more rapidly than the behavior of `CS`, so in the same test intervals there is larger r-m-s deviation. More importantly, the test interval boundaries match the discontinuities in `CS`, but they do not match all the system discontinuities. In component testing the code is available to aid in choosing test intervals, and a subdomain boundary at (for example) 3.3333333333 was matched to the discontinuity in `CS` behavior there (right of Fig. 2). In contrast, the location of discontinuities in system behavior is an emergent property of the system, and cannot be predicted from component-testing data. This system also has a discontinuity at 3.3333333333 that is accurately captured, but the emergent discontinuity at about 2.96 is not (right of Fig. 3). The emergent discontinuity arises only from the system design, and although component testing could be adjusted after the fact to capture it, that would violate the essential purpose of components: to be reused without retesting.

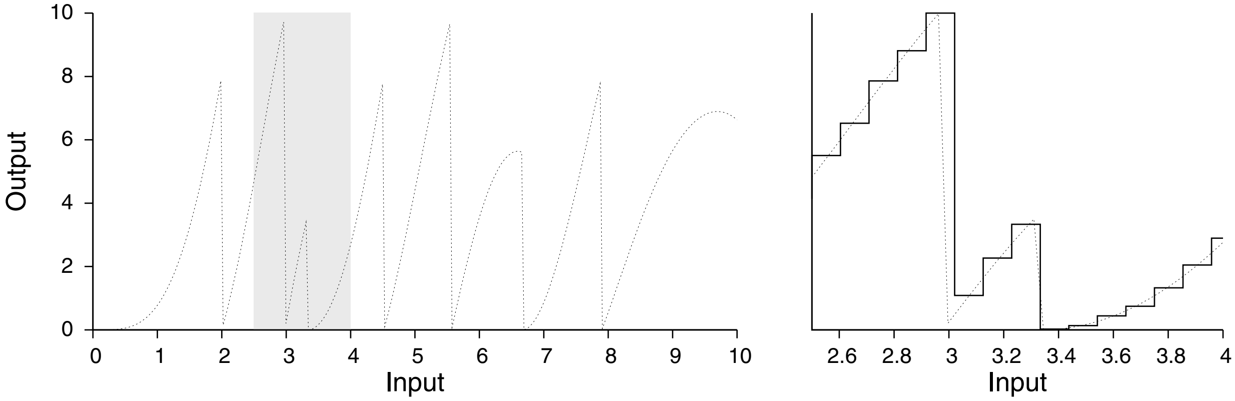


Figure 3: System `CS | CS` behavior measured and predicted from component test results for `CS`

Error in component measurement can be decreased by a better approximation. Using smaller subdomain intervals is one way, but the tools also provide fitting the best line to measurements in each subdomain [4]. Using this option, the `CS` component r-m-s error is reduced to an average of 0.8% using only 24 subdomains (vs. 96 above), while the r-m-s system prediction error for `CS | CS` is reduced to an average of 0.4% , using 82 subdomains[1].

The ultimate source of all prediction errors is the profile-mismatch problem. When `CS` is tested, its uniform input profile produces a non-uniform output profile. The tools predict what this profile will be by subdomain for the given system structure, as shown in Fig. 4, which also shows the observed profile obtained from conventional instrumentation of the executing system[2].

[1] One of the good features of the linear approximation is that it automatically refines system subdomains when making a system prediction, and to some extent the refinement tracks the system behavior. The problem of emergent discontinuities is thereby reduced, but not eliminated.

[2] The piece-wise linear approximation was used with 24 subdomains of `CS`; accurate profile predictions require small approximation errors.

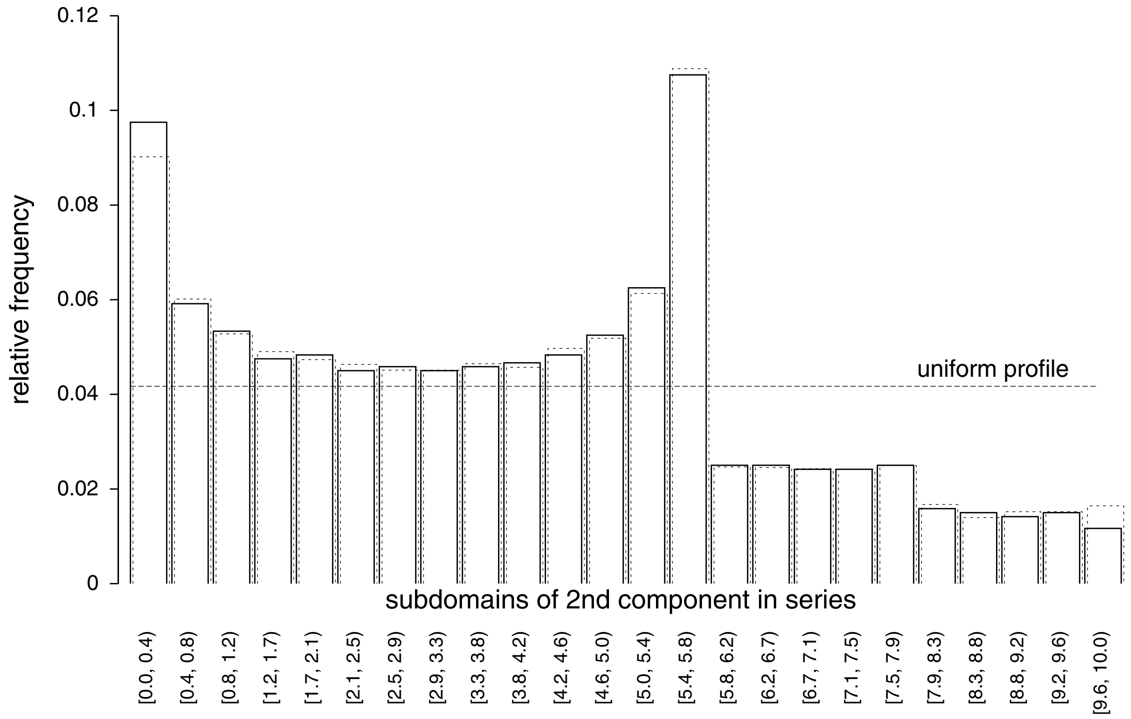


Figure 4: Output profile from the first component of `CS | CS`, which is the input profile seen by the second component. The solid-outline bars are measurements and the dashed bars are predictions

The inaccuracies displayed in this section are inherent limitations of component testing for quality as a means to guarantee system quality. Further discussion is postponed until Section 5.

4 Component and System State

A fundamental principle of software component design was recognized by Parnas more than 35 years ago: persistent software state should be encapsulated in what he called "information-holder modules" [6]. Local state is also a cornerstone of object-oriented design. These programming concerns stem from modularity and software maintenance: if state is allowed to pervade the code for a system, no code can be changed without examining and potentially changing all the code. But software components developed for deployment in a variety of systems add an important new reason to forbid shared state. If a component's behavior depends on state and state can be changed by other components, no isolated component properties can be trusted when that component is placed in a system: in the system, component interactions will make each component's behavior depend on them all. For example, testing a component in isolation ('unit testing' as usually understood) has little significance if the integrated system has shared state.

When there is only local state, a component's stand-alone behavior can be accurately described and there is hope for predicting the behavior of any system using that component. It is usual to say that the component's behavior, functional and non-functional, depends on both its input and its state. Indeed, in software reliability engineering (SRE), state parameters are simply added to the input parameters in sampling program behavior. It is common in testing object-oriented classes to set some state value externally, then try different inputs; this amounts to taking test points for the class as (input, state) pairs.

Unfortunately, the 'state is just another input' view is wrong in principle. Input is an *independent* test variable. The tester controls it and the component does not. Indeed, it is basic testing wisdom that every input may have to be considered, since in use the component may receive anything. Programs routinely contain conditional statements for unusual or 'error' inputs and issue error messages for them. State is quite different; it is *dependent* (on input): the component completely controls its state, setting state values in response to inputs, then subsequently using the values it has previously set. It is very unusual for a program to contain code that checks state values, because it is expected that they could not be in error. (A misguided view—erroneous state values are often created and it is good defensive coding to check for them. The point is that programmers know that states do not arise independently, and hence do not need to be verified as inputs must be.) Because state is not externally controlled, a tester must not choose state values arbitrarily—to do so risks fabricating behavior that can never occur. Trying impossible states is a mistake because:

- It wastes scarce test time;
- It can give a distorted picture of what a component actually does;
- It hides real but unexpected state values that could lead to failure.

The last two points often interact in a pernicious way, when the states tried come from a specification: If the actual program states do not follow the specification, by forcing the program into expected (but actually impossible) states it may appear to be correct, while in actuality its states are nothing like those of the specification and have not been tried.

The correct way to sample (test) in the presence of state uses *sequences* of inputs alone. Starting from an initialization state ('reset'), each input in a sequence sees the current state (starting with the reset value), and the program maps this (input, state) pair to an output value and a result state, the latter next paired with the following input in the sequence. Thus the output and result-state results depend on (input, state) pairs, but only with state values that can actually arise.

Another characteristic of real component states is that their values are very sparsely distributed over the possible data-type values. For example, a database is a particular kind of file, but of all possible files, even those observing strong syntactic restrictions, almost none are actually databases. It would be silly in testing a database component to attempt to cover the huge file space of non-databases. A database component creates databases, according to specification if it is correct, according to its own crazy logic if not. It must be tested with what it creates, and the tester's main task is to recognize when its logic goes wrong and creates states that are not according to specification.

To illustrate these points about state, consider the following specification of component `DIG` that requires state:

> On each execution, `DIG` takes a floating-point input in (-10.5,9.5). Let X be the input rounded to the nearest integer. After initialization, `DIG` uses local state to remember the length of longest sequences of the same digit (X non-negative input 0..9) over a series of executions, returning 0 on each execution. For negative rounded inputs, $-10 \leq X \leq -1$, the component returns the longest sequence of digit $|X| - 1$ that has been previously seen. A negative input is not considered to interrupt the count of a sequence of positive inputs. Out of range inputs are ignored, returning -1. Should a sequence of more than 9 successive same digits occur, -1 is returned, and the count stops at 9.
>
> For example, the following table shows a sequence of inputs in the first row, with the specified outputs in the second row.
>
> | | | | | | | | | | | | | | | |
> |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
> | (init) | -1 | 1 | 0 | -1 | 13 | 0 | 0 | 5 | -1 | -6 | 5 | 1 | -2 | -6 |
> | (0) | 0 | 0 | 0 | 1 | -1 | 0 | 0 | 0 | 3 | 1 | 0 | 0 | 1 | 2 |

A Perl program was written to implement this specification. The state that a component keeps to meet the specification is not prescribed, but this implementation keeps a counter for each digit and remembers the previous digit and its current sequence count. In keeping with the tool restrictions described in Section 2, its state is a single floating-point value, the integer part recording the previous digit seen d, and the mantissa keeping the digit counters, in order after the decimal point: (tenths) count of d seen so far in the current repetition; (hundredths) maximum count of digit 0 seen previously; (thousandths) maximum count of digit 1 seen previously; and so on for the other digit counters. For example, in state 3.14159265358, digit 3 was last seen and once so far; 0 had a maximum previous count of 4, 1 of 1, 2 of 5, 3 of 9 (or perhaps the 3-count exceeded 9), ..., 9 of 8. At the end of the sample sequence in the specification, the state should be 1.13100020000.

The subdomain description files for DIG have 46 input subdomains and 36 state subdomains, chosen as a compromise between accurate test measurements and the need to display graphs that are not too crowded. The tools were used to test DIG with systematic state sampling (wrong!) and with sequences of input (right!). The tests show component output values and also resulting state values, presented as a pair of 3-D graphs in 2-D projection. Figures 5 and 6 show test results when each of the 46 × 36 subdomains is sampled six times in a systematic (equi-spaced) manner for a total of 9936 test points. Figures 7 and 8 instead use 130 sequences of random inputs, of random lengths up to 130, keeping track of the 9114 (input, state) pairs that result to plot the graphs.

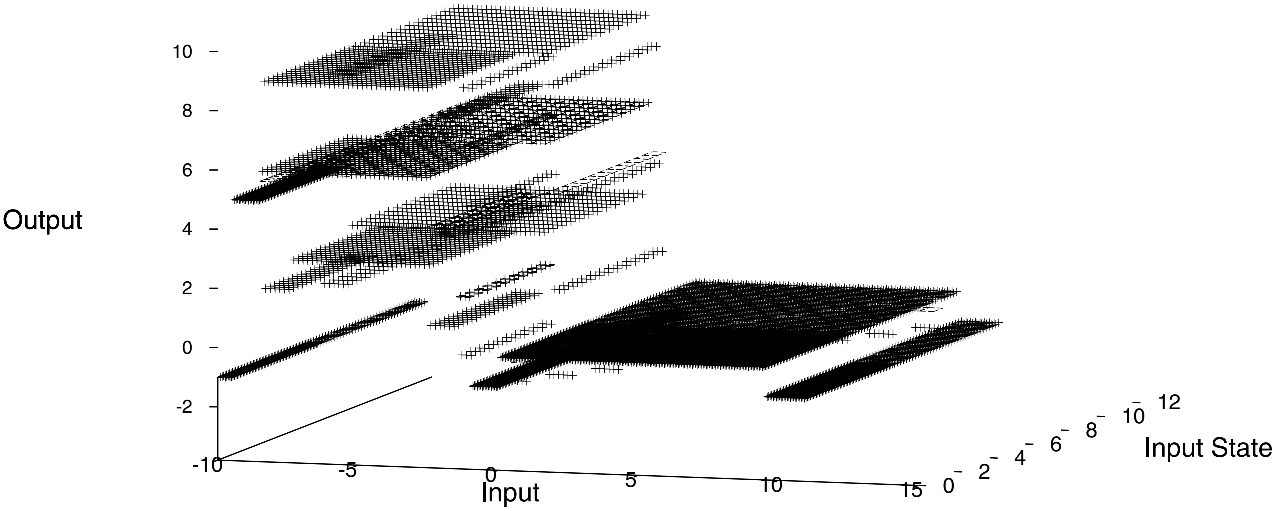


Figure 5: Equi-spaced sampling of DIG output as a function of (input, state). Crosses are measured data points; rectangles are average test results over input×state subdomains.

The best way to see the effect of erroneous systematic state sampling is to compare Figs. 6 and 8. Figures 5 – 8 illustrate testing a component with state and the tradeoff between systematic subdomain sampling and testing with random sequences. The following are a few observations:

Functional vs. state behavior. Programmers and testers are accustomed to thinking of output as the essence of software behavior. It is less usual to consider the functional form taken by state, since states are hidden. Furthermore, since state is both an input- and output variable, if the program does nothing with it, the result is an identity function as at the right of Fig. 6 for the error case of inputs at 10 and above. To get similar linear input-output behavior, the programmer would need to assign the input variable to the output variable. Thus better understanding is obtained from the output graphs like Fig. 5. The flat area with output 0 for non-negative inputs (right of the figure) is cases where digit counts are stored. And in the state graphs like Fig. 6 the result-state values are increasing roughly linearly with the input (the 'staircase' right from input 0). For negative inputs, the behavior at the

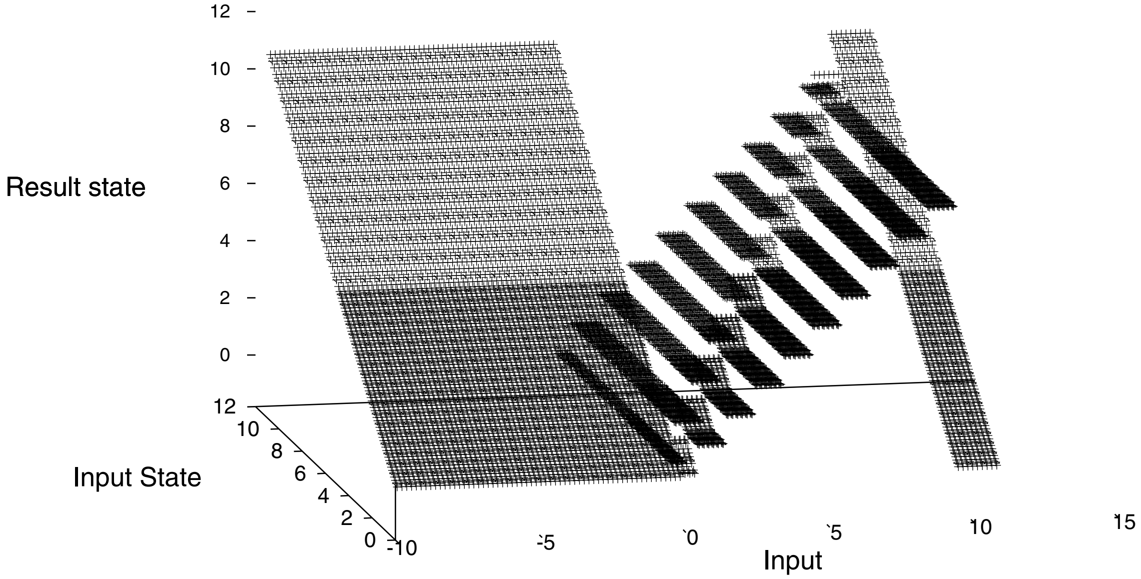


Figure 6: Equi-spaced sampling of DIG result-state as a function of (input, state).

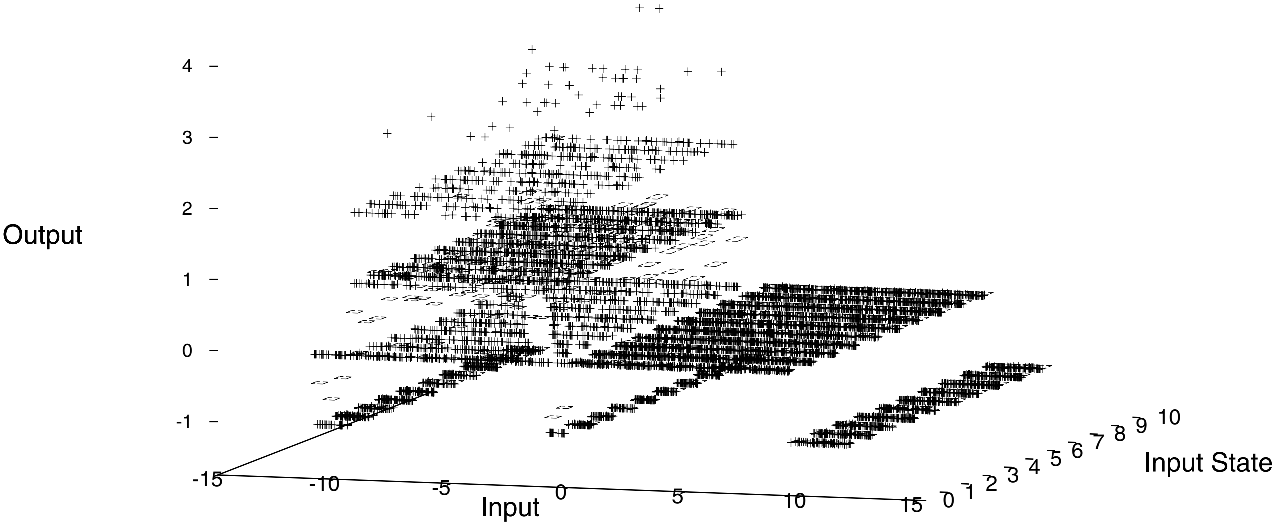


Figure 7: Random-sequence sampling of DIG output as a function of (input, state).

left of Fig. 5 is layers at output 0, 1, 2, ..., 9; these are the return of the stored digit counts. The subdomain averages are not accurate here, since at a given negative input all the values between 0 and 9 might be returned. The reader who thinks that state is easy to understand should try to account for short ramps in the state dimension along the 'staircase' for positive inputs in Fig. 6, for example in

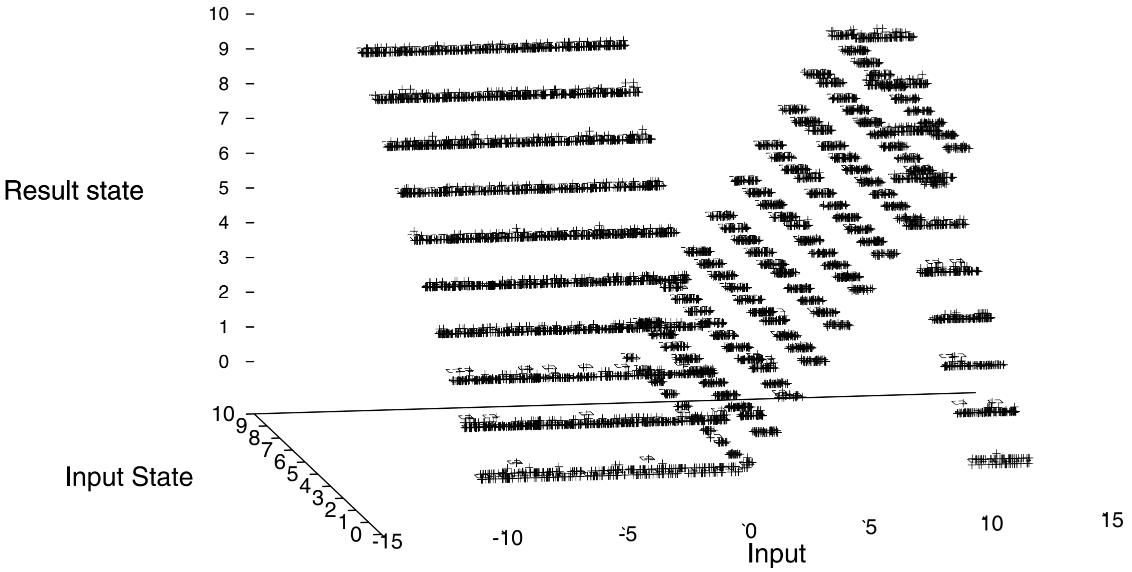


Figure 8: Random-sequence sampling of DIG result-state as a function of (input, state).

the four (input×state) subdomains within [0.5,1.5) × [1.25,1.75). Figure 9 shows this region magnified from Fig. 6 at the left and from Fig. 8 at the right.

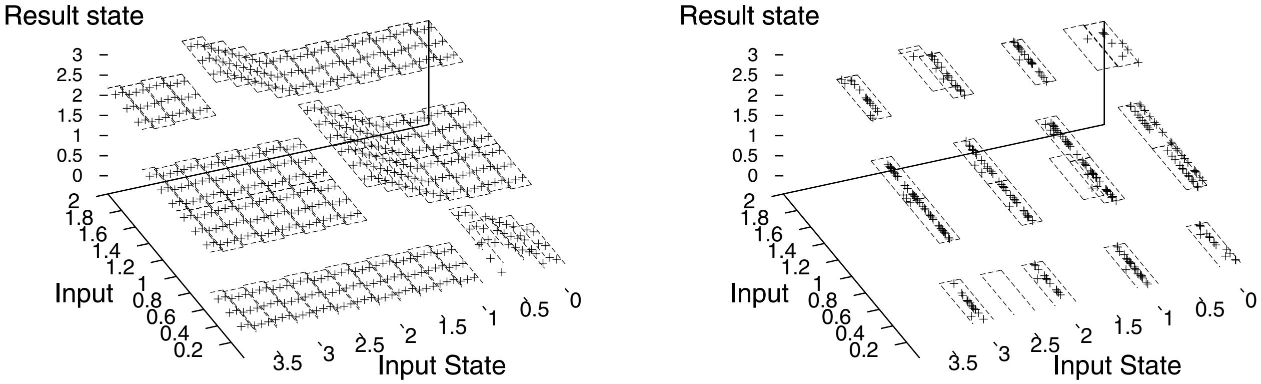


Figure 9: Detail of result-state behavior of DIG (left: systematic sampling; right: input-sequence sampling

Infeasible states. The most obvious difference between the equi-spaced sampling figures and those using sequences is missing states in the latter. Systematic sampling gives all state values equal weight, where in fact state values cluster closer to integers and state never reaches value 10 or above. What Fig. 5 doesn't show is a number of exceptions reported by the running code when it tries to compute with an 'impossible' state, for example, using 'digits' like 11. Chasing such 'bugs' is a waste of time, since they only occur in executions that are artifacts of systematic state sampling.

Spurious or missing behavior. There is a subtle difference between systematic-state sampling and input-sequence sampling shown at the left of Figs. 5 and 7. Each systematic test is started with state initialized to the value to be tested. Therefore, systematic tests are really only looking at sequences of length one, and the particular state chosen is critical. If that state is infeasible, the behavior is spurious. But for DIG it is also likely that the systematic sampling will fail to select interesting states. Thus while Fig. 7 shows the pattern of layers of output at 0, 1, 2, ..., Fig. 5 has strange gaps in this pattern. An empty diagonal 'track' is visible in the random-sampled figure, where output 0 does not occur for (input,state) pairs around (-1,0), (-2,1), (-3,2), etc. This is a consequence of the peculiar meaning given to states: for example, a state with integer part 3 means that digit 3 is currently being seen, hence must have occurred at least once, never 0. With systematic sampling, there is no reason not to select (say) state 3.0000000000, so this diagonal track is missing in Fig. 5. In 'compensation' there is a spurious diagonal track at level -1 for positive inputs. A close study of Fig. 5 reveals many such 'features' of the test, but close study is just what the figure won't bear, since much of it is erroneous.

Uncovered states. There is a cost for the use of test sequences, however. Large repeat counts are unlikely to occur, and in Fig. 7 only a few go as high as 4. If the tester were concerned about what happens for larger counts, the input sequences required to enter these states would be very long, and no sequence can settle the question of whether states at 10 and above are possible (because they are not). Similarly, more sequences would be required to answer the question about ramps in the state behavior; not enough samples were taken for the behavior magnified on the left of Fig. 9 to show clearly on the right of the figure. Is the ramp behavior feasible but hard to reach, or is it an infeasible artifact of systematic sampling?

Improving the tests. There seems no alternative to using sequences of inputs, since only in this way will tests show real behavior. Random sequences are used because the idea of a 'equi-spaced' or 'systematic coverage' sequence is hard to define. If there were such a sequence, how would the parameters of sequence-length, position in the sequence, and input values be 'systematically' varied without going too far in exploring one parameter at the expense of the others? However, when testing to a specification, it is sometimes possible and necessary to bias random factors. In the testing of DIG, a bias could be introduced that favors selecting the same input in successive positions in the sequence, which would more quickly reach states with larger counts. The need for bias was noticed by Antoy in testing a class that could store or retrieve values in a set. Unless he artificially forced more store operations than random selection would have used, the tested states were almost never complex enough to be interesting [1]. The difficulty in biasing random input sequences is in choosing a bias, i.e., what states to explore, and in finding input sequences that will explore them.

When a system is built from components with local state, the system behavior depends on a system state that is a cross product of the components' local states. Each local state of course can influence only the behavior of the component that owns it, but since the system can invoke all components it intuitively combines all their states[3]. Thinking about what the system does or should do requires reasoning about this composite. The issues of state infeasibility raised for components come up again for systems. To give a simple example, suppose that the component DIG were assembled with a conditional component that allows it to count only repeats of digits 0..3 and 7..9; the digits 4..6 are considered errors in this system, and are sent to a third component that returns -1 instead of being stored and counted. However, negative values are not filtered by the conditional, so an input in -7..-5 should always return a count of 0. The output behavior of this system is shown in Figs. 10 and 11 using random-sequence sampling. A band of states has become infeasible, and new plateaus at -1 for the 'error' digits, and at 0 for reporting their 'counts' have been created. Otherwise, Figs. 10 and 11 are copies of Figs. 7 and 8. They display both the strengths and weaknesses of random-sequence sampling: Only 322 of the 1656 subdomains were calculated to be feasible and were so; but 73 of the 1334 infeasible predictions were wrong because execution did fall in those subdomains.

[3] The composite is not a 'global' state, because components cannot use it to communicate.

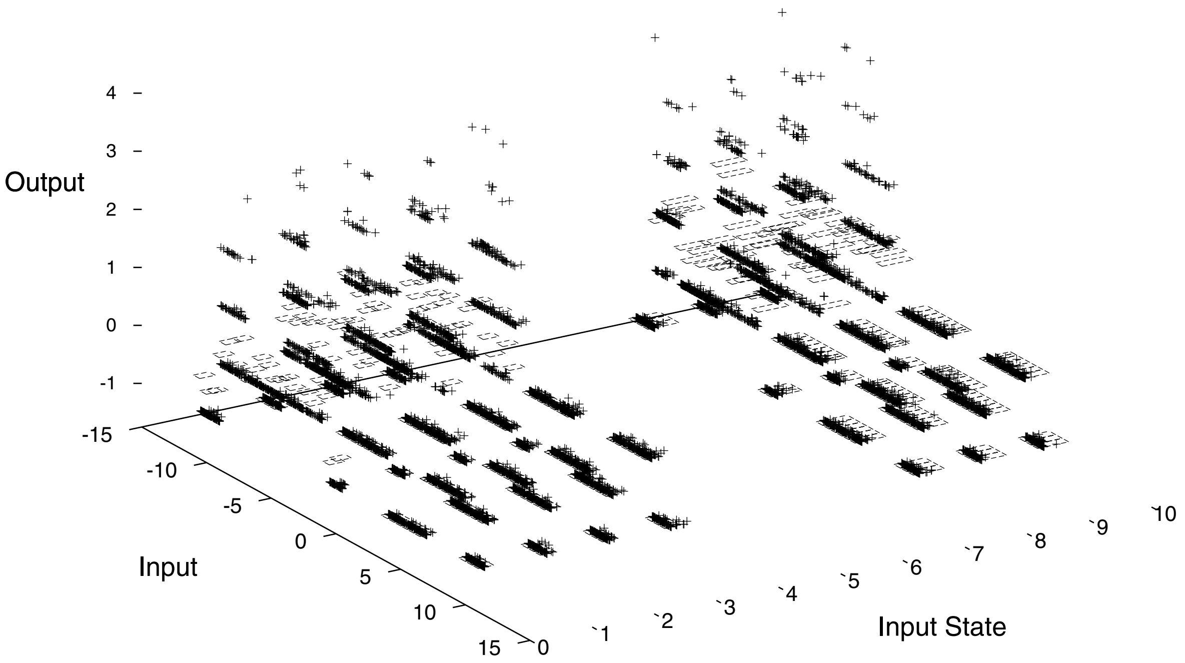


Figure 10: Predictions (rectangles) and measurements (crosses) of system output behavior.

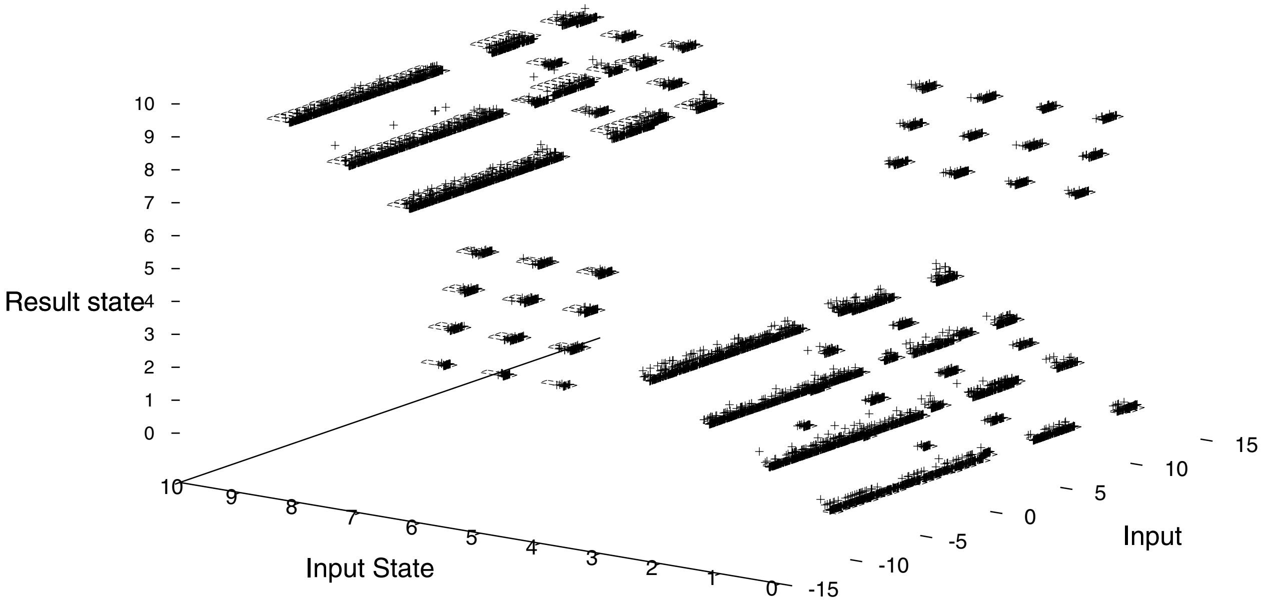


Figure 11: Predictions (rectangles) and measurements (crosses) of system result-state behavior.

Predictions of system behavior in Fig. 10 are not good for negative inputs (the cases where DIG is reporting previously-stored counts). The primary reason is that for DIG measurements (Fig. 7) cannot be accurate, as described above. However, there is a sense in which the measurements and predictions are useful even though they are averages of many scattered values. When DIG is reporting counts, all of its values lie between 0 and 9, with an average of 5 for systematic sampling. For the number of random trials shown in the figures, the values don't often reach 4 and average $1-2$. While the averages are not correct for any subdomains smaller than the whole negative input space, they *are* the correct averages, which for some purposes is enough. For example, the system run time is roughly the same no matter what count value is reported, so for predicting run time the calculations are OK.

If only one component of a system has state (as in the previous example), the component synthesis tools can display the system prediction as in Figs. 10 and 11. But we run out of dimensions if the behavior depends on two or more local states. For example, composing the digit-counting program `DIG` with itself (however silly that might be), the output would be a function of input and a pair of states. The system subdomains would be rectangular solids and no comprehensible graphs could be drawn for the system. As the number of components with state increases, the ability to visualize system behavior falls, and tools are less helpful. Graphs aid understanding of what is being tested and how well; for example, Fig. 8 illuminates the issue of testing infeasible states.

There is one important case with composite state in which the tools can be helpful, however. Often state takes the form of 'modes.' The state values are limited to a small set, used by the program to tailor its behavior without requiring user choices on each execution. (Modes are sometimes called 'preferences.') A command-driven text editor is a good example: it has an 'input' mode in which it stores text, and a 'command' mode in which text is taken as directives to act on what has been stored. (Of course, such an editor's primary state value is the stored text itself.) When several components assembled into a system each have only this kind of 'mode' state, then the system cross-product state consists of a finite number of state tuples, and these can be graphed in the input-state dimension as discrete points. An idealization of the text-editor example, used with a conditional component that itself has states to restrict access to the editor commands, is presented in [2].

A system whose composite state is mode-tuples is particularly difficult to test, because many tuple values are infeasible, the specification may be less than clear about what unusual tuples are supposed to mean, and the tester does not know which ones to try, or how to excite them. Yet catastrophic system failures arise precisely from somehow reaching in execution some strange state, not tested because it was not thought of or it was deemed impossible. In such cases it is especially dangerous to confuse specified states with real program states, since unexpected real states are 'meaningless' in the specification, which in no way prevents the program from falling into them and crashing.

5 Discussion: Component vs. System Testing; Design Rules

It is common wisdom that testing components in isolation is easier than testing a system built from them, but not useful in predicting what the system will do. The tools and experiments reported here support the first half of this position but refute the second half. Testing components *is* easier, if only because the results can always be graphed—a component has exactly one collection of local states. Furthermore, test subdomains for components can be adjusted to match significant input points like those where discontinuities occur. By using input test sequences, infeasible state values are avoided. Experimenting with these tools shows that system predictions *can* be made from component test results. The predictions are not always accurate, but by studying the sources of their inaccuracy, we may hope to come up with some rules for 'good' component and system design, that is, design rules that will make the prediction algorithms work.

As currently practiced, unit testing is an activity that is considered 'good for you,' but is quantitatively unrelated to the quality of the component, much less the quality of a system to be developed later. The surrogate quality measures for unit test (like structural coverage, number of points tried, or number of failures found) have only a weak connection with quality—everyone knows examples where the measures were good but the component later failed badly. More important, unit measures are entirely disconnected from the rest of development. No one says, "since I achieved 85% unit statement test coverage, system testing can use 36% fewer cases," for example. In other engineering disciplines, things are quite different. Parts are tested to verify the parameters on which system design is based. For an example in structural engineering, because the rivets pass hardness tests, beams fastened with them are expected to resist shearing. In software, unit testing is done and then forgotten, with no quantitative contribution to the rest of development. Our tools do better. A component test is accompanied by an error analysis, giving the r-m-s deviation of each subdomain approximation from measured behavior. When that error is small, the subdomains are capturing behavior well, and being well tested. Large errors indicate that the subdomain breakdown could be improved. The tools provide graphical feedback that helps the programmer/tester to adjust the subdomain boundaries[4]. The subdomain error analysis is thus a quantitative measure of component-test quality, and it can play a crucial role in component-based system development, described below.

The primary theoretical difficulty in accurately composing component test results into system predictions is the profile-matching problem. In an extreme case of mismatch, the first component might send the second only one value y_0; then no test of the second other than at y_0 has any significance. Our tools that predict system behavior try to get around the profile-matching problem by testing in subdomains. Component behavior is thereby split into a collection of behaviors, one on each of many subdomain pieces of the input space. But unless the subdomains are fine-grained enough to capture behavior perfectly (in the worst case, they would have to be singleton points, which is clearly impractical), they can be fooled. In the extreme case above, y_0 falls in some input subdomain which will have been tested in the second component, but the behavior across that subdomain may vary so much that what happens on input y_0 is poorly represented by the subdomain average value.

It is, however, possible to quantify the quality of a system prediction made by our tools without testing that system. Our tools can calculate the profile each component sees in the system (e.g., in Fig. 4). Insofar as this profile approximately matches the profile used to test that component in isolation, the predictions should be good. Where there is strong mismatch, it calls for replacing or retesting the component.

Design Rule 1 *Check calculated system internal profiles against component test profiles; retest or replace components when there is a mismatch.*

There is a simple special case of Design Rule 1: Don't use a general-purpose component with wide functionality for a few special values. It will have been tested with a wide spread of inputs, and your special values will likely have been neglected. Retest at those values or use a different component with restricted capability.

Design Rule 1 does not apply to the component tester, who is working before any system application is known. Rather it tells the system designer not to trust a component in a particular system if a mismatch is observed. For example, Fig. 4 shows that subdomains below about input 6.0 (particularly two of them) in the second copy of `CS` are more frequently used in the system `CS | CS` than they were tested.

Profile mismatch matters most when there is a discontinuity in component behavior that falls across some subdomain, as illustrated in Fig. 3. Since discontinuities always arise from conditional statements, the system designer can try to avoid the problem.

Design Rule 2 *Avoid series combination of components in which the first has major discontinuous behavior.*

One way to observe Design Rule 2 is to amalgamate two series components into one. This moves an offending discontinuity from the system- to the component level, where it is easier to capture with tests and subdomains. Another way is to move conditional statements out of the series code and into a new conditional component, which selects one of two new components, formed from the alternatives that were in the original code. The conditional component can be given subdomains that exactly capture the break point. Of course, Design Rule 2 should be used sparingly, because too much amalgamation defeats the purpose of component-based design.

It is tempting to formulate a simple design rule to control problems with persistent state: don't use it! But that's hardly practical. There has been very little study of testing in the presence of state, so the following are based on just a few experiments.

[4] In the tool versions now under development, changes in subdomains are handled incrementally, so that measurements and calculations are not repeated for subdomains that have not changed.

Design Rule 3 *Don't distribute 'modes' across states of several components; group all modes into one control component for a system.*

Design Rule 3 suggests that it is more comprehensible and easier to test many modes as state values in a single component than to deal with them at system level as cross-product states from several components. The number of cases is the same, but collecting them makes it harder to miss some composite case.

Design Rule 4 *When state has a large set of possible values, make one component responsible for all possibilities rather than distributing values among several components.*

Design Rule 4 can be viewed as a restatement of Parnas's advice to use information-holder modules (but not many). It is supported by systems that have a single database component. The reasoning behind the rule is similar to that for Design Rule 3: having many information holders creates problematic cross-product state values whose meaning is unclear, which must be dealt with at system level. Testing a database component is difficult enough by itself.

By observing these design rules (and no doubt many others yet to be formulated), the predictions made by our tools are likely to be accurate. Then unit (component) testing can assume a new importance: system tests are never needed. That does not mean that an arbitrary system can be assembled from tested components and it will work 'out of the box.' It is obvious that there might be arbitrary mistakes made in the system design that are unrelated to the quality of the components used. What it does mean is that when the predictions of tools like ours are accurate, predictions can be substituted for actual testing of the assembled system. Instead of trying system executions, one can examine the predictions of what the system will do and match them against the system specification. The advantage of using calculated values in place of real system executions is that they are much cheaper to obtain, and therefore the designer can afford to try several designs and to use a variety of alternative components in them. Our tools can calculate and graph the behavior of a system hundreds of times faster than a real system can be executed, and that does not count the time required to integrate the actual components and ready a system test.

The suggested paradigm for system development using components is then the following:

1. Develop components in isolation and test them using subdomains that capture their behavior, to the limit of testing resources. Or, the development and testing work may have been already done if the component was selected from a repository. In any case, a subdomain breakdown and quantitative test accuracy, not just the component code, are required.

2. Design the system using the components of step 1.

3. Calculate properties of the system of step 2, using CAD synthesis tools like those described in this paper. In particular, calculate the internal profiles seen by each component in place.

4. Check the validity of the system calculations by comparing the internal profiles calculated in step 3 with the test profiles in step 1. If there is a bad mismatch, repeat steps 1 and/or 2, changing components, tests, or system design to minimize it.

5. Using the system specification and the system properties calculated in step 3, verify that the system should meet its requirements. If not, change components and/or design and repeat from step 3.

This procedure corresponds quite closely with the way system design and implementation is done in structural, mechanical, and electrical engineering, particularly in relying on CAD tools to synthesize and verify a system design. No one (other than Rube Goldberg!) builds a mechanical system by buying components and trying to put them together by trial and error, then extensively testing a system where they seem to fit. The success of the older engineering disciplines in component-based design is one that we have yet to approach in software engineering.

References

[1] Sergio Antoy and Richard G. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Trans. on Soft. Eng.*, 26:55–69, 2000.

[2] Dick Hamlet. Subdomain testing of units and systems with state. In *Proceedings ISSTA 2006*, pages 85–96, Portland, ME, July 2006.

[3] Dick Hamlet. `www.cs.pdx.edu/∼hamlet/components.html`, 2007.

[4] Dick Hamlet. Software component composition: subdomain-based testing-theory foundation. *J. Software Testing, Verification and Reliability*, 17:243–269, December 2007.

[5] Dick Hamlet, Dave Mason, and Denise Woit. Theory of software reliability based on components. In *Proceedings ICSE '01*, pages 361–370, Toronto, Canada, 2001.

[6] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. of the ACM*, December 1972.

[7] Denise Woit and Dave Mason. Software component independence. In *Proc. 3rd IEEE High-Assurance Systems Engineering Symposium (HASE'98)*, Washington, DC, November 1998.

Selecting Software Estimating Techniques that Fit the Software Process

Kal Toth

Portland State University

ktoth@cs.pdx.edu

Abstract

When embarking on a new project, the software engineering manager will need to decide early on whether to follow a Waterfall, Agile, Prototyping, Incremental or some hybrid or variant of these software processes. To assess project feasibility, to secure budget, and to properly plan resources and schedules, responsible managers should also decide about their software estimating process - whether to us expert judgment with estimating rules-of-thumb; parametric techniques like COCOMO and Function-Points; or estimating databases populated with analogies and proxies from prior projects. The characterizing attributes of a given new project greatly influence software process and estimating choices. This paper provides context and guidance that will help the software practitioner understand the influences of project attributes on the selection of suitable software processes and on software estimating techniques when embarking on the next software project. This paper will also assist companies decide how to best apply their resources to maintain a suitable software estimating infrastructure to support project planning and execution.

About the Author

Kal Toth is the Director of the Oregon Master of Software Engineering Program (OMSE) and Associate Professor in the Maseeh College of Engineering and Computer Science, Portland State University (PSU). A Professional Engineer (P. Eng) with a software engineering designation registered in British Columbia, he has a Ph.D. from Carleton University in electrical and computer systems engineering. Kal has over 25 years of management, technical, and consulting experience leading and working for a range of technology companies and organizations including Hughes Aircraft, Datalink Systems Corp., BC Software Productivity Centre, the CGI Group Inc., Intellitech Canada Ltd., National Defence (Canada), Communications Canada, and External Affairs (Canada). He has managed and participated in a technical capacity addressing project management, software quality, and information security aspects of air traffic control, e-commerce, distributed information, and packet-switching systems. At PSU he facilitates software engineering courses including software project management, software engineering principles and processes, software quality, and practicum projects.

Introduction

Software engineers and teams are constantly being asked questions like: “How long will this take?” and “How many developers are needed to get it done? Far too often, estimates are produced in an ad hoc manner and yield unreliable results, or results that are misunderstood and misused. The prudent software engineer will study the attributes of the software problem at hand and systematically apply their experience on similar projects to come up with useful estimates. They will also revise their estimates on an ongoing basis as project uncertainties firm up. The most experienced software professionals will apply software estimating methods that have worked consistently for them in the past. Mature software companies will support their projects by sustaining software estimating culture and infrastructure including proven estimating processes, tools and historical project data.

When practitioners work on similar projects, in similar domains, using similar software processes, and their company’s standard estimating process, they tend to produce fairly useful and consistent estimates. But what does a team or company do when they shift from Waterfall to Agile development? What happens if they transition from an estimating culture without systematic estimating to one where estimates are integral to the software process? What about the developer making a career move from the aerospace industry with a formal estimating culture to a company or industry sector in the world of commercial software products, e-commerce applications, enterprise information systems, embedded computing, or open source software? Such radical shifts may render a proven estimating technique to be inappropriate in the new domain. At the very least, several adjustments will be needed to re-tune and re-target the estimating technique being used to adapt it to the new development culture.

To make the necessary adjustments, software engineers first need to understand the ramifications of the estimating techniques they used in the prior context. Next they need to differentiate the new types of projects they expect to tackle from their previous ones (What’s different?). Finally they need to learn which aspects of their estimating technique(s) can be re-used, and which ones will need to be adjusted or re-worked to support the new domain. Armed with a suitable mapping of estimating techniques and development processes, the software professional should be able select the most appropriate estimating technique for the new domain, identify the gaps, and transition to the new or revised estimating approach.

One approach to tackle these challenges is to consider key project attributes that distinguish projects from each other. If we can use these attributes to characterize both software process selection and software estimating choices, it should be possible to relate software processes and estimating techniques to achieve “best fit” (see Figure 1). This should help companies and their software engineers focus on the most suitable range of estimating techniques for their context thereby improving estimating effectiveness.

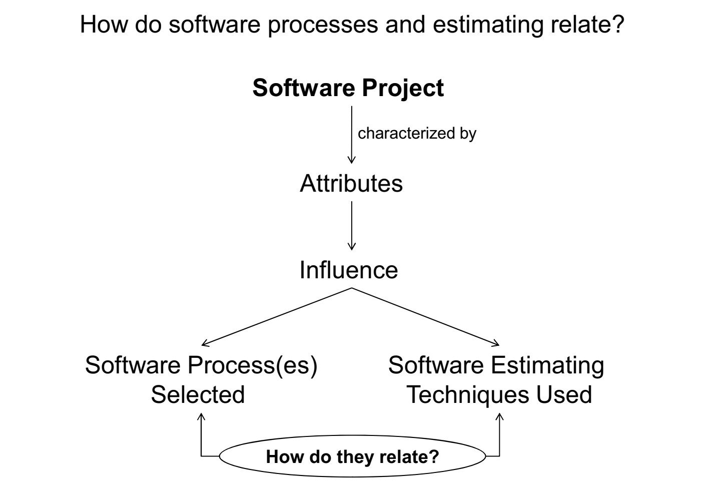

How do software processes and estimating relate?
Software Project
characterized by
Attributes
Influence
Software Process(es) Selected
Software Estimating Techniques Used
How do they relate?

Figure 1

Central Questions Posed by this Paper

Development teams naturally tend to select the software process they are most familiar with. This may often be the correct choice, especially if the company tends to take on similar projects in a given application domain. However, many companies, especially larger ones, have projects with widely ranging process needs and they often have multiple software process cultures and competencies across their various software development groups.

As suggested in [1], the unique attributes of a new software project should be examined and related to the company's capabilities when deciding on the software process to be adopted. Once this choice is made, project planning can begin in earnest including the application of suitable processes to estimate effort and schedules. Given that software estimates are derived, one way or another, from work activities, it follows that the chosen software estimating technique(s) should also account for the unique attributes of the project.

Such relationships between software process and estimating for a given project are not widely acknowledged across the software engineering community. This paper explores this gap by addressing the following questions:

❑ What are the key attributes characterizing an arbitrary project?
❑ How do such factors influence process selection? [1]
❑ How do such factors influence software estimating choices?
❑ Which SW estimating techniques are most suitable at a given stage of software development?
❑ Which SW estimating techniques are most suitable for a given software process?

Key Assumptions Underlying this Paper

The primary goals of software estimating are to: confirm project feasibility, rationalize budget for the project, and use estimates to control the project (determine when a new or better estimate is required).

Ideally, projects should start with stable objectives and scope:

❑ Scope should not change very much, otherwise you are really tackling a different project; [2]
❑ Properly defined scope covers the breadth of project requirements and constraints;
❑ When required functions and features emerge, they should be consistent with objectives and scope;
❑ Requirements should not expand scope without stakeholder agreement (those controlling budgets).

Estimates are uncertain, but they should improve with time: as the project progresses, new knowledge yields better estimates which reduces estimating uncertainty; estimating should be iterative - new estimates are opportunities for re-shaping the project, re-targeting objectives and scope as required.

The Customer and Contractor must mutually embrace success:

Both parties must recognize that requirements are rarely certain or static and that resources (budget and schedule) are finite; they must agree to consider and accept trade-offs between requirements, budget and schedule; also agree to explore and understand impacts of key project influencing factors while meeting

[1] Project influencing factors can be used as "adjustment factors" or "drivers" when producing estimates

[2] If scope is pushed out significantly by requirements changes, the customer and contractor should be prepared to revise estimates, assess project impacts on budgets and schedules, and renegotiate scope or trade-off requirements to fit within previously agreed upon budgets and schedules.

mutual commitments; The Customer and Contractor are thereby mutually involved and committed to achieve success (a.k.a. “win-win”).

Software Estimating in a Nutshell

Software Estimating Basics

Most software estimating techniques start by estimating the “size” of the software product deriving effort and schedule estimates from software size. Historical and industry data are used to arrive at standard allocations of effort to such size measures; and schedules are derived using other guidance that distributes effort over the timeline or calendar. Several (but not all) techniques apply project adjustment or influence factors to account for various attributes of the project.

Software size represents the magnitude of the problem being tackled. The underpinning rationale is that effort and schedule should naturally correlate with the relative difficulty of the software problem being tackled. Software “size” is typically measured in terms of the number of features or “points” representing the amount of software to be constructed – for example, the number of functions, stories, use cases, objects, components, files, user interface widgets, lines-of-code (LOC), etc. Such points are typically categorized by relative complexity and other product attributes with effort estimates being associated with each type of point. Nominal effort estimates per point come from industry data or past company projects. The software estimator’s task, then, boils down to assessing the number of points of each applicable category and calculating the total estimated product size, effort and schedules.

Of course, many practitioners directly estimate software construction effort without explicitly relating their estimates to product size – deriving loading profiles and schedules from these effort estimates.

A critical aspect of software estimating is to determine where to start. Figure 2 illustrates this challenge. Should we first estimate size and then estimate effort and schedule? Or should we start with a fixed schedule and then determine how much software (size) and effort can be fit into that schedule. Or is it really the level of effort that should drive the estimate? In other words, should we estimate how much software can be built within allocated effort and then find a schedule that will best accommodate this effort and the amount of software to be built?

What do software projects estimate?

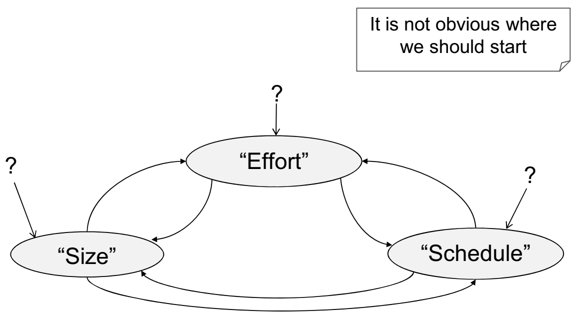


Figure 2

Using Historical Project Data

Estimates can be derived from historical industry and company data. Parametric models with estimating tools exist to support companies lacking their own historical project data. Tool vendors have collected, categorized and analyzed data from numerous projects across various application and technology domains. Their tools use empirical formulas to calculate estimates according to inputs that characterize the properties (influencing factors) of the software project to be estimated. The estimator’s main task is to identify and evaluate the new project’s influencing factors and provide these as inputs to the tool.

Historical databases of past projects can improve the quality of estimates and reduce estimating overheads, that is, the effort and time expended to produce good estimates. Such databases represent the collective software project knowledge of the company, thereby providing consistent and more certain estimates, than the other techniques mentioned above. Historical estimating biases (over and under estimating) can also be derived and used to adjust estimates. Furthermore, useful rules-of-thumb can be extracted from such databases and used to produce rough-cut estimates that match the company's project profiles. Estimating databases are populated with software components from prior projects. Referred to as software "analogies" and "proxies" they are "sized" and categorized by their project attributes. Statistical techniques (e.g. regression analysis) are applied to estimate and adjust size, effort and schedule from previous analogies and proxies.

Using Influencing Factors to Drive Estimates

It is fairly clear that the nature of the software product itself and the company's capabilities are the main factors that drive a given project estimate. But are there other useful factors? Luckily, we can draw upon parametric estimating techniques that employ detailed attributes of a software product, the chosen development environment, project team competencies, and other project aspects.

Take note that these parametric estimating techniques do not use the same or equivalent influencing factors (though they somewhat overlap), and many estimating techniques use them only superficially, or even disregard them altogether. For example, experienced software engineers often produce "fuzzy" estimates from their experience with similar prior projects together with prerequisite knowledge of the current project's objectives and scope - project influencing factors are considered implicitly - rarely explicitly. In other cases, rough estimates are derived from industry or company rules-of-thumb without addressing project attributes systematically.

If project influencing factors are not available or not used, the prudent estimator should understand and communicate to management and the customer that such estimates are very rough (uncertain) and that they should be accordingly used with caution.

Estimating Uncertainty

Software estimating should not be thought of as a one-time effort for a project. Estimates and plans should be updated iteratively during or at the completion of each significant development stage. Early on, estimates tend to be highly uncertain; as the project moves forward, additional knowledge and experience accumulate hence yielding tighter estimates (see Figure 3). Barry Boehm called this the "Cone of Uncertainty" [2].

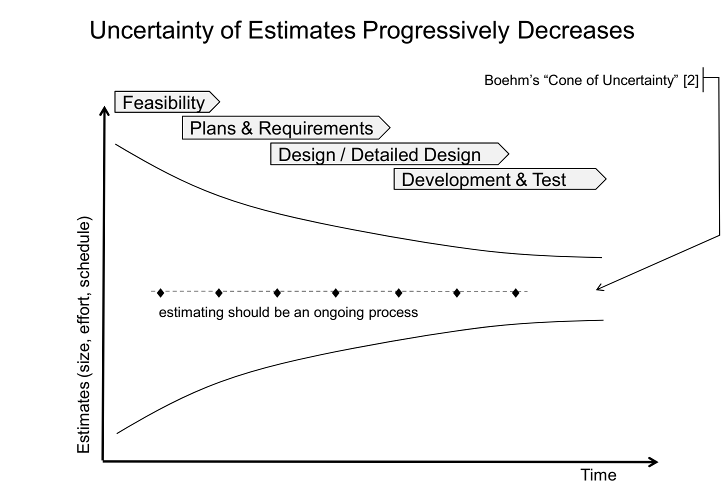

Uncertainty of Estimates Progressively Decreases
Boehm's "Cone of Uncertainty" [2]
Feasibility
Plans & Requirements
Design / Detailed Design
Development & Test
estimating should be an ongoing process
Estimates (size, effort, schedule)
Time

Figure 3

Recommended Readings on Estimating: Barry Boehm [2], Steve McConnell [3], and Mike Cohn [4].

Software Process Selection

What Influences Software Process Selection?

To understand how project factors influence software process selection, let's review the rationale that motivate the software practitioner to select one software process over another:

- ❑ **Waterfall development** is best suited for projects with high complexity, criticality, and maintainability requirements. Waterfall processes are fairly sequential allocating detailed analysis, specification, and design decisions in the initial stages of development to address performance, security, reliability, safety, maintainability, and other non-functional requirements including design constraints. This approach implies higher process ceremony than most of the alternative software processes.
- ❑ **Agile development** focuses on working code over detailed specification, analysis and documentation. Short iterations (often called "sprints") of fairly systematic development are advocated with close customer involvement, test-driven development, continuous integration, pair-programming, and ongoing refactoring. Agile processes are characterized by lower process ceremony than Waterfall.
- ❑ **Prototyping**, sometimes referred to as evolutionary development, is exploratory in nature tackling unknown or poorly understood aspects of the project - for example, user interface and technology uncertainties. Typically, software prototypes are developed using low process ceremony to build knowledge before transitioning the project into more systematic development following Waterfall and Agile development processes.
- ❑ **Incremental development** aims at increasing project scalability by applying divide-and-conquer strategies to fashion independent development activities that may be executed concurrently or serially. Such independent development is achieved by using thorough analysis and design techniques to partition the requirements and/or the architectural design into loosely coupled components that may be independently addressed. Development increments may be governed by Prototyping, Waterfall or Agile approaches.

Incidentally, none of these processes is "pure" – they don't adhere dogmatically to a given process. For example, Waterfall processes do not necessarily dictate strict sequencing, high ceremony processes, or massive documentation. Similarly, Agile does not mean abandoning proven design principles, coding standards, or test coverage analysis.

Aside: You might wonder why I left iterative and spiral software development off the above list of processes. My reasoning is that Prototyping, Agile, Waterfall and Incremental processes possess very distinctive properties. And both iterative and spiral development processes are really strategies that can be applied to fine-tune them. For example, an entire project could be broken down into iterations along the time-line where each iteration delineates several concurrent increments of development. Iteration is also a way of organizing the work within a given development increment to progressively explore software work products as they are being prototyped, or augment functional capabilities as new stories are formulated during Agile development, or execute mini-waterfall processes to progressively refine and mature a branch of the software development effort. Meanwhile, spiral development, which is very closely related to iterative development, allocates risk management activities to separate spirals where each could follow Prototyping, Agile or Waterfall processes to address specific risk mitigation objectives, albeit using different styles of development.

Software Process Ceremony is Driven by Project Influencing Factors

Figure 4 depicts how software processes relate process ceremony to the difficulty of the problem being tackled. Prototyping, Agile, Waterfall, and Incremental development are shown as software processes that cover a scale of increasing process ceremony addressing increasingly difficult (harder) software problems. For example, a project to develop a large, complex system to control a nuclear reactor, an aircraft navigation system, or an embedded medical device, may be best organized as a number of concurrent increments of development that incorporate Prototyping, Agile, and Waterfall processes across an iterative lifecycle. In other words, a hybrid high-ceremony process is planned out to address the needs of a very difficult software problem.

Process Ceremony Versus Problem Difficulty

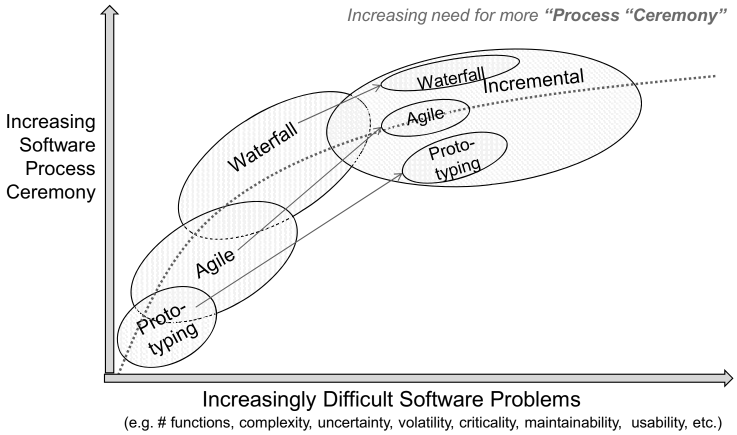

Increasing need for more *"Process "Ceremony"*
Increasing Software Process Ceremony
Waterfall
Incremental
Agile
Proto- typing
Waterfall
Agile
Proto- typing
Increasingly Difficult Software Problems
(e.g. # functions, complexity, uncertainty, volatility, criticality, maintainability, usability, etc.)

Figure 4

Note: See Toth [1] for insights into process selection, and Brook and Toth [5] on process ceremony.

Customer-Supplier "Flexibility"

Customer-supplier "flexibility" represents the degree to which the customer and supplier have agreed to constrain requirements, budget and schedule. For example, the budget might be agreed to be fixed while the requirements are flexible (e.g. allowed to vary within project scope), and the schedule is constrained (e.g. the schedule may be pushed out by some agreed period of time). Futrell et. al. in [6] recommend explicitly addressing such flexibility in the project plan using a matrix. Table 1 illustrates customer-supplier flexibility for 4 projects.

Customer-Supplier "Flexibility" Biases the SW Process

	Project A	Project B	Project C	Project D
Requirements	"Fixed"	"Fixed"	"Flexible"	"Flexible"
Budget	"Constrained"	"Flexible"	"Constrained"	"Fixed"
Schedule	"Flexible"	"Constrained"	"Fixed"	"Constrained"

Table 1

Project A: Consider Waterfall to ensure fixed requirements are addressed within the constrained budget. Estimating effort should focus on assessing whether an acceptable schedule is feasible;
Project B: Consider Waterfall to ensure fixed requirements are addressed within the constrained schedule. Estimating effort should assess whether an acceptable budget is feasible;
Project C: Consider Agile since time-to-market / product release is the top priority. Estimating should be "design-to-schedule" within an acceptable budget;
Project D: Consider Agile since budget is fixed. Estimate should be "design-to-budget" within an acceptable schedule.

How does flexibility affect estimating?

As discussed earlier, the goal of estimating is to assess the expected product “size”, the effort to build the product, and the schedule within which to build the product. However, it is not always clear where to start, and it is not always possible to arrive at a reasonable size estimate. One interesting observation, is that a flexibility matrix focuses our attention on a natural starting point. Consider the following project flexibility possibilities:

Requirements are “fixed”: This implies that the customer has a fairly fixed idea of functions, features and non-functional requirements that have been well-specified in a software requirements specification (SRS) or similar document. The customer has likely agreed to a fairly systematic change process as well. The main goal, therefore, is to estimate acceptable budget and schedule to implement the requirements. Some flexibility with respect to budget and schedule may be allowed. Such projects are considered “requirements driven”.

Schedule is “fixed”: This implies that the customer or marketing department is highly focused on delivery milestones and shipping dates to meet market urgency or operational needs. The main goal is to estimate which requirements can be implemented within the allocated schedule. They are also likely to allow changes in functions and features mid-stream to meet requirements. And some budget flexibility may be permitted. Such projects are referred to as “schedule driven” or “design-to-schedule”.

Budget is “fixed”: This implies that the customer expects the project to operate within a budget envelop that will directly or indirectly fix the total allowable effort. Exceeding the budget will likely lead to serious customer dissatisfaction and possibly even payment penalties. Such projects will endeavor to assess which functions and features can be feasibly implemented within the fixed budget. Depending on the customer’s preferences, trade-offs between software size and schedule may be made before settling on an acceptable project posture. Such projects are considered “budget driven” or “design-to-cost”.

Figure 5 illustrates these three project flexibility scenarios and reinforces the iterative aspect of estimating. For example, a design-to-cost (budget driven) project starts with budget and effort considerations but is likely to trade off various requirements and schedule options before settling on a combination of effort, requirements and schedule that will satisfy the budget constraint.

“Flexibility” also Influences Software Estimating

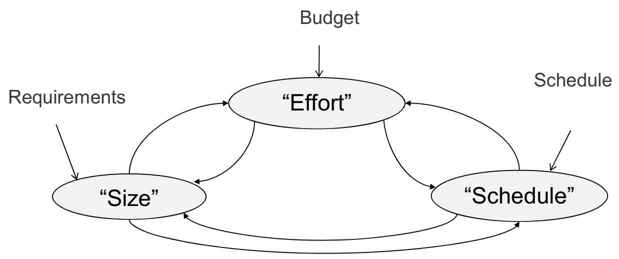


Figure 5

Common Estimating Techniques and Strategies

To explore the implications of project influencing factors, let us delve a bit further into the common estimating techniques (illustrated in Figure 6) focusing on key differentiating aspects:

Expert Judgment: Assumes an experienced software practitioner is familiar enough with the application domain to make a sound estimate. The expert is really calling on their personal recall of past software projects and project attributes. Such estimating is typically referred to as rough-cut or “fuzzy” estimating”. When the base data is simply memory recall, we refer to it as “Guestimating”.

Function-Point Estimating: Function and feature points and categories are derived from industry data. The estimator assesses the number and complexity of functions and features by category to arrive at a size estimate and uses estimating adjustment factors that characterize the requirements and the design.

Analogy-Based Estimating: Past project analogies are characterized by their attributes, actual software size, effort, and project schedules. The estimator searches for comparable analogies in the historical database to arrive at representative estimates.

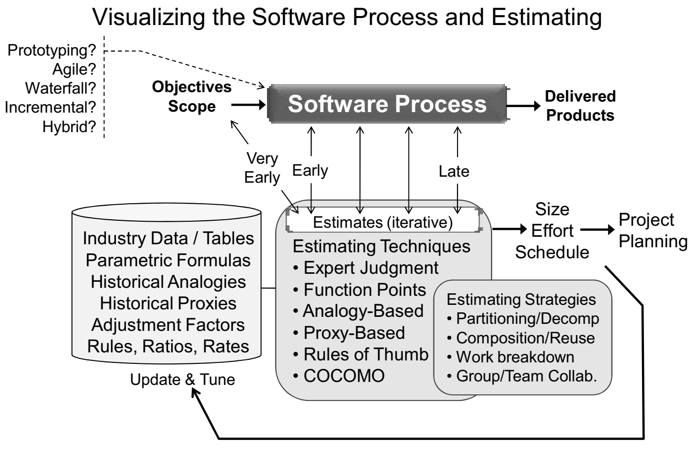


Figure 6

Proxy-Based Estimating: Software features (e.g. stories, objects, components, widgets, etc.) of past projects are counted, characterized, categorized, and maintained in the estimating database. The estimator assesses the number and size of each feature point in the new product, searches for comparable proxies, and extrapolates size, effort and schedules from this base data.

Estimating Using Rules-of-Thumb: Rules-of-Thumb include: software productivity rates (LOC/unit of time), possibly categorized; re-use factors such as the relative cost of reusing software components as a percent (%) of the original development effort; and effort distribution factors used to distribute rough-cut estimates over project activities and phases. Rules-of-Thumb may be derived from industry data or may be drawn from a company's software project history.

COCOMO Estimating: A fairly widely used parametric technique, COCOMO used empirical formulas based on regression analysis of industry data categorized by relative project complexity ("organic", "semi-detached", or embedded"). COCOMO uses estimating adjustment factors that characterize product, computing environment, personnel and project attributes. COCOMO uses an input size estimate in lines-of-code (LOC) to yield effort and schedule estimates. Several COCOMO variants and products are available.

Estimating Strategies for Refining Estimates

Design to Cost/Schedule: This is really a cost or schedule driven estimating technique. Through trial and error, size estimates yielding the desired cost or schedule are located. Software size may be derived from analogies, proxies or function-points.

Partitioning/Decomposition: Requirements may be partitioned into relatively independent parts that can be separately estimated. Once a stable design is developed it can be partitioned into loosely coupled parts and separately estimated.

Composition: This is actually a design technique that can be used to produce bottom-up solutions from components that can be estimated and aggregated to yield rolled-up estimates.

Group Estimating: Delphi estimating involves bringing together several software practitioners who use their expert judgment to converge on a collective estimate through consensus. Team members may also be called upon to support bottom-up estimating, a recommended strategy to secure commitment of team members to achieve results that meet their own estimates.

Project Influencing Factors

It should be clear by now that project attributes influence both the software process and the software estimating technique(s) we may elect to use. Top-level project flexibility attributes have been identified (requirements, budget and schedule) that influence how to begin strategizing about estimating. These factors also help us decide whether to use Waterfall or Agile development – but they tell us less about how to choose when to use Prototyping and Incremental development. The discussion about process ceremony should encourage us to consider incremental development strategies to scale-up by organizing hybrid development processes.

Meanwhile, we have examples of estimating techniques that leverage project attributes to account for project differences. COCOMO, for example, uses various product, personnel competency and other project attributes. And Function-Points uses product-specific attributes almost exclusively. Of course, many estimating techniques almost totally ignore such project attributes assuming the estimating expert will somehow bring them into play as required.

Figure 7 captures and synthesizes the fundamental observations made earlier about project flexibility, COCOMO, Function-Points, and personal experience. This figure presents what I believe to be a fairly complete "taxonomy" of project influencing factors. The hierarchy is first broken down by project flexibility drivers, and then further broken down by product, environment, personnel and project pressure / involvement factors. Several of the project influencing factors map to Function-Points and COCOMO adjustment factors.

Important Note: Although Function-Points and COCOMO provide guidance on assigning weights to their adjustment factors, they use different value ranges and aggregation methods. I have made no attempt herein to rationalize these scales. For the purposes of this paper, the reader should simply assume that each of the influencing factors may take on a range of values (e.g. small to large; few to many; low to high; etc) and that such factors should be used to adjust initial estimates upwards or downwards as appropriate.

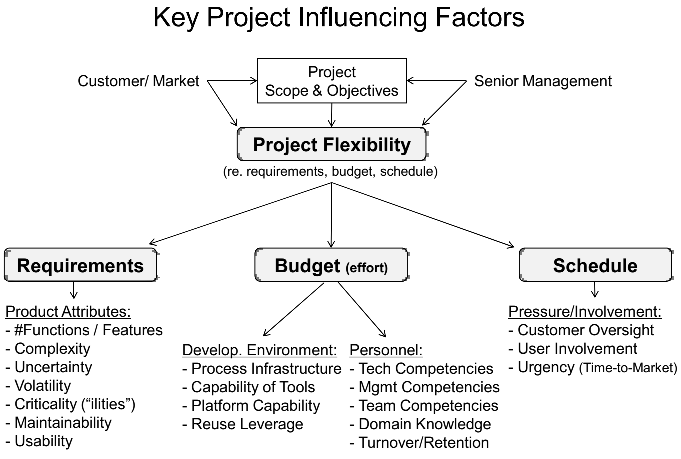


Figure 7

Project Flexibility

As discussed earlier, this represents customer-supplier flexibility regarding the project's requirements, budget and schedule within agreed objectives and scope. For example, the customer and supplier might agree to be flexible with respect to requirements, but fixed with respect to project schedule (completion date) while constraining total budget.

Product Attributes

Functions / Features (#): the relative number of functional requirements and features; from a few key functions to a very large number of functions, use cases, scenarios, user stories, etc.
Complexity: represents the relative complexity of the product in terms of the number of dependencies among functional requirements, features, external interfaces, and technologies.

Requirements Uncertainty: unambiguous, precise, and logically complete specifications denote low uncertainty; high-level, vague and incomplete requirements imply high uncertainty.
Requirements Volatility: the rate at which and the extent to which the customer and users are expected to make changes to required functions and features throughout the project.
Criticality: characterizes the security, reliability, safety, availability and performance requirements (a.k.a. "ilities") of the project – for example, web-sites, games, electronic toys are examples of low criticality applications; aerospace, military, and embedded applications are typically high criticality projects.
Maintainability: relates to the quality of the documentation and the modularity of the design – ad hoc, experimental prototypes do not need much documentation; highly maintainable systems have extensive and high quality requirements specifications, design documentation, and installation manuals.
Usability: represents the relative ease of use of the product's user interfaces and navigational controls.

Personnel / Team Factors

Technical Competency: analysis, requirements, design, coding, integration, testing, etc.
Management Competency: planning, estimating, directing, monitoring, reporting, negotiation, etc.
Team Competency: communications, collaboration, coordination, interpersonal, etc.
Application Domain Knowledge: knowledge and experience with domain functions and features.
Retention / Turnover: ability of the company and team to retain quality and proven personnel.

Development Environment Characteristics

Process Infrastructure: technical (eg coding) standards, spec templates, checklists, guidelines, etc.
Capability of Tools: extent and quality of requisite tools supporting reqts, design, coding, testing, etc
Platform Capability / Volatility: capability and maturity of O.S., DBMS, commercial apps, etc.
Reuse Leverage: extent and quality of reusable components and source code from prior projects.

Pressure and Involvement

Customer Oversight: the degree to which progress is made visible to customer representatives through reporting, documentation, and demonstrations.
User Involvement: the degree to which users get involved in developing and reviewing work products from requirements, through development, to product acceptance.
Schedule Pressure / Market Urgency: relative urgency expressed by the customer to deliver and release final products.

Key Project Factors Influence Software Process Selection

With reference to Table 2, the following summarizes the "key drivers" (only) that influence the selection of each software process (note: lesser project influencing factors have been set aside):

Prototype Development is driven by product complexity, unknown platform capability (technology risk). Also usability, requirements uncertainty and unfamiliar domain knowledge (requirements risks).

Agile Development is driven by schedule pressure (urgency to deliver to market). Volatility and uncertainty of requirements are also key Agile drivers. Agile is particularly dependent on technical competency and user involvement.

Waterfall Development is driven by criticality and maintainability requirements. Waterfall is also heavily dependent on process infrastructure, management skills, and customer oversight. Typically, the customer requires significant project visibility through documentation, reporting and in-progress demonstrations.

Incremental Development is driven by large-scale, that is, large numbers of functions, features, and complexity. An incremental process implies strong process infrastructure, customer oversight and project management competencies.

Key Factors Influence Software Process Selection

	Prototyping	Agile	Waterfall	Incremental
Product Requirements:				
• **Functions / Components**				♦
• **Complexity**	♦			♦
• **Volatility**		♦		
• **Uncertainty**	♦	♦		
• **Criticality**			♦	
• **Maintainability**			♦	
• **Usability**	♦			
Team / Personnel:				
• **Technical Competency**		♦		
• **Management Competency**			♦	♦
• **Team Competency**				
• **Application Domain Knowledge**	♦			
• **Retention / Turnover**				
Development Environment:				
• **Process Infrastructure**		◊	♦	♦
• **Capability of Tools**				
• **Platform Capability / Volatility**	♦			
• **Reuse Leverage**				
Pressure / Involvement:				
• **Customer Oversight**			♦	♦
• **User Involvement**		♦		
• **Schedule Pressure / Market Urgency**		♦		

Table 2

A Closer Look of Estimating and Influence Factors

Suppose we had a few highly expert estimators who are trained and experienced in all of the estimating techniques identified herein. These estimating gurus are also well-versed in facilitating estimates with developers. Suppose also that their company supports a "Super Estimating Model" (see Figure 8) with commercial Function-Points and COCOMO tools as well as industry tables and a corporate repository of historical analogies and proxies. Estimating has been facilitated by having the proxies and their nominal counts categorized and regressions of these proxies automated. Various rules-of-thumb have been derived and tuned from previous projects.

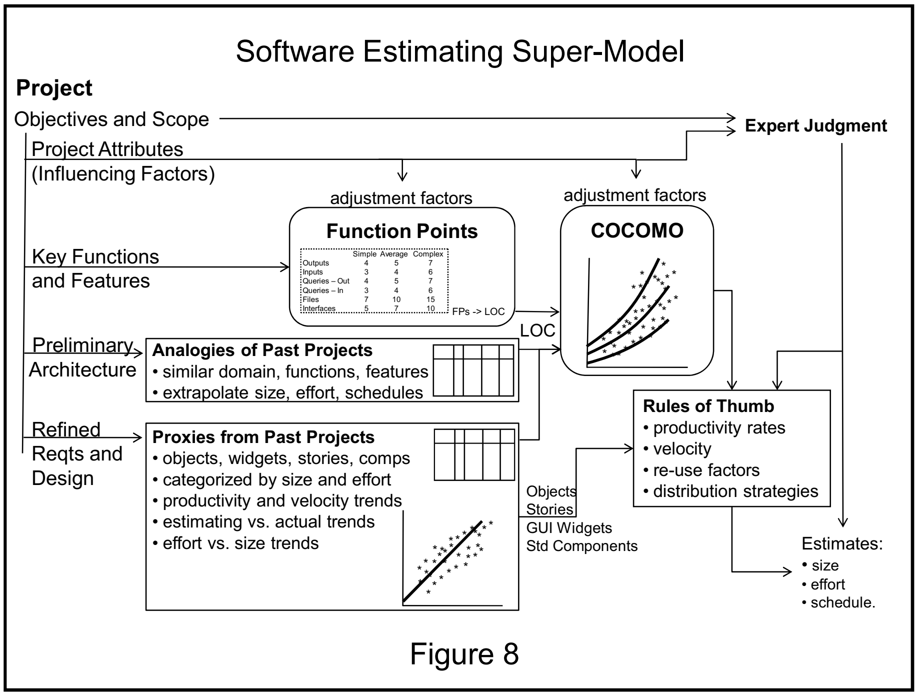


Figure 8

On top of this, the estimating gurus are quite familiar with the range of application domains and technologies within which the company specializes, namely, healthcare and medical informatics. These estimating specialists also recognize that this project, like most, will encounter some technology risks, application domain unknowns, and uncertain requirements. Also most of their projects need to address some critical security, reliability and availability requirements. And they have a good supply of skilled developers at their disposal, some of whom have experience building security and reliability systems, and others who have good experience with Agile development.

Now suppose they are responsible for leading the software estimating effort on a new project that is similar to various elements of prior projects covered by the company's historical estimating database. Their estimating process would be characterized as follows:

A. They would apply the following estimating steps iteratively throughout the lifecycle of a given project from early feasibility, through requirements and design stages of development, to core software development stages
B. As they progress through the project from feasibility through to final delivery, they would:
    1. Validate project objectives and scope;
    2. Identify project influencing factors / attributes (hopefully most of them, but perhaps not all);
    3. Assign appropriate values for each project influencing factor;
    4. Obtain key functions and features that comprise the initial requirements baseline;
    5. Obtain preliminary architectural design plus additional functions and features as they become better understood;
    6. Obtain refined requirements and design details as they become better understood;
    7. Repeat step 6 as required until project completion;
    8. If major changes in scope are encountered and (hopefully) agreed to, many of the steps above will need to be re-visited to assess impacts and produce new estimates.

Software Estimating by Stage of Development

Given the above scenario, the estimating specialist would develop the software estimates iteratively along the timeline as indicated in Figure 9. Judgment with rules-of-thumb is useful during early project stages. Function-Points estimating is useful during initial project planning and requirements engineering; Analogy-based estimating is most useful in support of initial planning, requirements and preliminary design stages. Proxy-based techniques should be used to develop more accurate estimates in later project stages from detailed design through software development (construction) stages. Here are some additional pertinent observations:

Choosing SW Estimating Technique by Development Stage

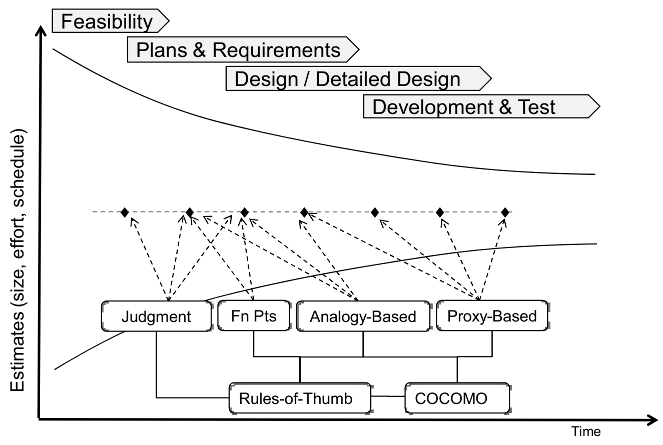


Figure 9

□ Judgment and Rules-of-Thumb are commonly used together; but Judgment accuracy fades as the project moves forward; Rules-of-Thumb also fade with time;
□ Function-Points needs special training;
□ Analogy-based estimating is easier to use and support than Proxy-based estimating;

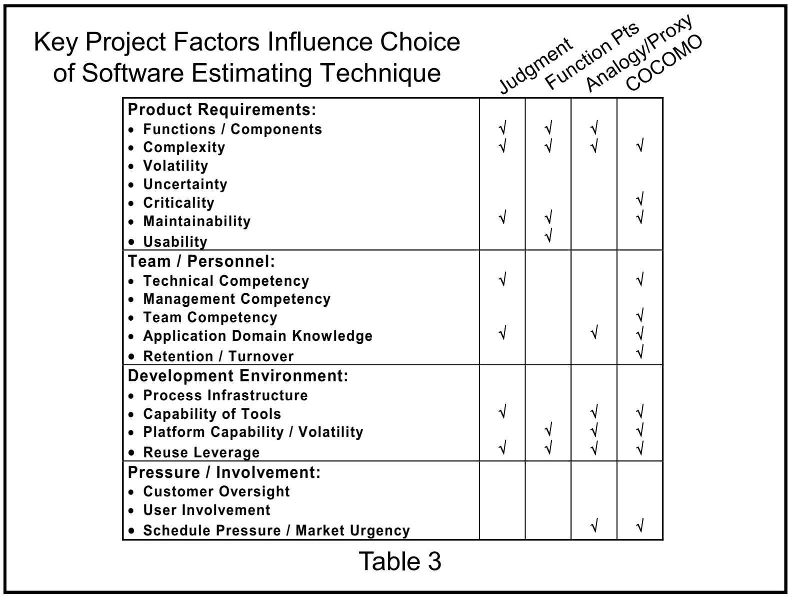
- □ COCOMO is useful throughout lifecycle; but COCOMO may over-estimate small projects;
- □ Analogy-Based and Proxy-Based require significant investments in maintaining a historical database;
- □ Rules-of-Thumb and Proxies need to be constantly tuned to realize good results.

Key Project Factors Influence Choice of Estimating Technique

Important Observations:

1. Table 3 identifies key project factors for comparing estimating techniques; unchecked factors have second-order influences and have been set aside (are unchecked);

2. Volatility, uncertainty, management competency, process infrastructure, customer oversight, and user involvement are not explicitly addressed by any of the estimating techniques. However, volatility, uncertainty and user involvement may influence the selected software process (Agile versus Waterfall for example). The estimator should consider these factors prior to completing an estimate.



Key Project Factors Influence Choice of Software Estimating Technique

	Judgment	Function Pts	Analogy/Proxy	COCOMO
Product Requirements:				
• Functions / Components	√	√	√	
• Complexity	√	√	√	√
• Volatility				
• Uncertainty				
• Criticality				√
• Maintainability	√	√		√
• Usability		√		
Team / Personnel:				
• Technical Competency	√			√
• Management Competency				
• Team Competency				√
• Application Domain Knowledge	√		√	√
• Retention / Turnover				√
Development Environment:				
• Process Infrastructure				
• Capability of Tools	√		√	√
• Platform Capability / Volatility		√	√	√
• Reuse Leverage	√	√	√	√
Pressure / Involvement:				
• Customer Oversight				
• User Involvement				
• Schedule Pressure / Market Urgency			√	√

Table 3

With reference to Table 3, estimating technique selections are driven as described below:

Judgment:

- □ Judgment can be used to yield effort estimates and/or size (LOC, stories, widgets, objects, …);
- □ Use Rules-of-Thumb to derive effort and schedule from size estimates;
- □ First consider number of functions and features, complexity, and technical competencies;
- □ Then consider applying the remaining maintainability, application domain, tools and reuse factors.

Function-Points:

- □ Works best when high level functions and their relative complexities are fairly well understood;
- □ Consider only if you have requisite training and practice so that you apply this technique correctly;
- □ Function-Points adjustment factors cover maintainability, usability, platform capability and reuse;
- □ If used together with COCOMO, most project influencing factors will be covered.

Analogy-Based and Proxy-Based:

- □ First consider prior projects with similar application domains, functionalities and complexities;
- □ Second consider past projects that used similar tools, were implemented on similar platforms, and were produced under similar schedule pressure;
- □ Proxy-Based estimating will need to choose the appropriate type of proxy and should incorporate reuse into estimates by applying rules-of-thumb;
- □ If COCOMO is used to derive effort and schedule estimates, many of the other influencing factors will be accounted for in the estimates.

COCOMO:

- ❑ COCOMO is capable of addressing a range of projects, including large scale and complex projects often with significant criticality and maintainability requirements;
- ❑ Use COCOMO to derive effort and schedule from LOC; or when other size measures like function-points can be converted into LOC;
- ❑ COCOMO, when carefully applied, should yield better estimates than Rules-of-Thumb estimates because this technique encourages the estimator to consider most of the project factors while Rules-of-Thumb do not cover such influencing factors;
- ❑ The estimator should consider adjusting the estimates for process infrastructure, customer oversight, and management competency which are not explicitly covered by COCOMO adjustment factors.

Recommendations / Guidance

The following top recommendations are first offered:

- ❑ When tackling a new project, the software practitioner should start by reviewing and annotating the project influencing factors presented in Figure 7. By creating and annotating a checklist of these factors, analysis will be focused on selecting the most appropriate software processes and estimating techniques thereby facilitating and rationalizing subsequent budgeting and planning decisions;
- ❑ Consider investing in a COCOMO tool – a relatively inexpensive but useful initiative. Although, COCOMO estimates can be rough-cut – especially when influencing factors are not well-understood, a COCOMO estimate using nominal values for these factors will provide a solid initial basis for new project planning and launch.
- ❑ Companies should also seriously consider establishing and maintaining an estimating database of historical software project data to support analogy-based, proxy-based and rules-of-thumb estimating. This would leverage several of the practical recommendations that follow.

The choice of software development process should be made as early as possible in a project. This can be done once the project's flexibility posture is decided and once key project drivers are understood.

With reference to Table 4, Prototyping is mainly driven by technical and requirements uncertainty and lack of knowledge; Agile processes mainly by requirements flexibility and schedule pressure; Waterfall development mainly by criticality and maintainability factors; and Incremental development by total project scale and complexity.

Expert Judgment combined with Rules-of-Thumb is a particularly useful and economic approach when little more than project objectives and scope are known, that is, during feasibility and early requirements formulation stages. However, the quality of estimates is highly dependent upon the domain experience and estimating skills of the software practitioner. Such estimates should be used mainly to assess project feasibility and rationalize budgets and schedules for new projects. They may be safely used to control small projects or those with constrained or fixed schedules or budgets and a flexible requirements posture (i.e. design-to-cost and design-to-schedule projects). However, when it comes to large, critical, complex and/or constrained projects, Expert Judgment with Rules-of-Thumb (only) estimating will not be good-enough, in most cases, to monitor and control budgets and schedules for fixed or highly constrained projects.

Because prototyping aims to explore technology, requirements and other risks and unknowns, it is not feasible to apply a systematic estimating technique to such activities. Prototyping, therefore, will require careful Expert Judgment to assess each project risk that could impact negatively on project feasibility, budget and schedule. Rules-of-Thumb will not apply. Such prototyping tasks will need to be broken-down, time-boxed, resourced, and addressed early enough in the project schedule to avoid budget and critical path (schedule) problems. Clearly, adequate risk identification, risk evaluation and risk mitigation actions are essential project aspects to coordinate with estimating efforts.

Function-Points estimating is particularly useful during the requirements phase. However, the technique requires intensive training and is not widely practiced in non-MIS (management information system) domains.

Process and Estimating Choices are Driven by Key Project Influencing Factors

	SW Processes					Estimating Techniques			
	Prototyping	Agile	Waterfall	Incremental		Judgment	Function Pts	Analogy/Proxy	COCOMO
Product Requirements:									
• Functions / Components				♦		√	√	√	
• Complexity	♦			♦		√	√	√	√
• Volatility		♦							
• Uncertainty	♦	♦							
• Criticality			♦						√
• Maintainability			♦			√	√		√
• Usability	♦						√		
Team / Personnel:									
• Technical Competency		♦				√			√
• Management Competency			♦	♦					
• Team Competency									√
• Application Domain Knowledge	♦					√		√	√
• Retention / Turnover									√
Development Environment:									
• Process Infrastructure		◊	♦	♦					
• Capability of Tools						√		√	√
• Platform Capability / Volatility	♦						√	√	√
• Reuse Leverage						√	√	√	√
Pressure / Involvement:									
• Customer Oversight			♦	♦					
• User Involvement		♦							
• Schedule Pressure / Market Urgency		♦						√	√

Table 4

Analogy-Based estimating and Proxy-Based estimating require the company to create, maintain and tune a historical database of analogies and proxies. One recommended strategy is to augment the company's version control system with tools to categorize and count proxies. Mechanisms for collecting and associating development effort would also be required. Similarly, Rules-of-Thumb should be maintained in such a database and tuned as new project data is collected. This implies a corporate investment in requisite tools and support.

Analogy-Based estimating is applicable during early stages of a project while Proxy-Based techniques begin to be practical during the core construction phases of a project. Agile development will use story-points and track "velocity" to estimate the next iteration (or two) of development. Waterfall development, meanwhile, can exploit standard components as proxies during design stages, and GUI widgets and use-case-points during construction stages. For smaller and less critical projects, Rules-of-Thumb (like productivity factors and velocity) are likely good-enough to derive effort estimates from LOC, Function Point and story-point estimates.

COCOMO can be used successfully to produce effort and schedule estimates for both Agile and Waterfall projects given LOC size estimates have been provided as input, the correct COCOMO estimating equations are used, and suitable values for the adjustment (influencing) factors have been made. COCOMO is well-suited for medium to large sized projects with high complexity, criticality, and maintainability requirements using Waterfall or Incremental development.

Incremental development can benefit from all of these estimating techniques. For example, Analogy-Based estimates can be used to secure early budgets and other resources; a prototyping increment would use Expert Judgment; an Agile increment might use Proxy-Based story-points; and Waterfall increments could apply Analogy-Based estimating with COCOMO. A pre-requisite for such an approach would be to partition requirements, and possibly the preliminary design as early as possibly in the project.

Summary / Conclusion

The software practitioner, whether developing project plans, leading the estimating effort, or contributing bottom-up estimates, has a vital role to play in the preparation and ongoing control of the project. As professionals, they are accountable to senior management and customers for the software estimates that shape the project. They should therefore be intimately involved in the selection of software processes that directly affect the estimates.

The selection of software process and software estimating technique are driven by key project attributes, namely, project influencing factors. The relationships between software process and estimating technique are far from deterministic, however, they do appear to be coupled in practical ways that should inform the software practitioner and should be exploited in the conduct of their work.

To enhance software estimating effectiveness, the company should first decide whether to invest in building and maintaining a software estimating database which represents a significant investment in tools and effort. However, without such a repository in place, the company will not be able leverage Analogy-Based or Proxy-Based estimating techniques and the increased confidence in estimates they offer.

As an alternative, the company may consider acquiring Function-Points training and tools. An investment in COCOMO estimating tools, meanwhile, would be useful as an adjunct to Analogy-Based, Proxy-Based and Function-Points tools and infrastructure.

The fall-back, of course, is to rely on Expert Judgment and Rules-of-Thumb which yield good-enough early estimates to secure budgets, and to support design-to-budget or design-to-schedule projects. However, Expert Judgment and Rules-of-Thumb will not yield better estimates-to-complete for projects with relatively fixed requirements. On the other hand, story-point estimating may be a reasonable alternative for companies practicing Agile projects where it is only necessary to estimate the next iteration (or so) of development.

A given project's influencing factors will inform the software professional, especially with respect to project planning and estimating. The project influencing factors identified in this paper can be used as a checklist to help decide on the software process, choose an appropriate software estimating technique, and use project influencing factors to adjust software estimates. Such a checklist should be annotated and appended to the project's objectives and scope during the feasibility phase and updated as required throughout the project lifecycle.

A project's influencing factors should to be tracked and "tuned" from project to project. This implies conducting in-progress and post-project reviews comparing actual results to original estimates. Such reviews should assess the relative impacts of project influencing factors on the obtained results and fine-tune the criteria used to select values for each of these factors.

In summary, software practitioners and their companies are strongly encouraged to create a sound software process and estimating infrastructure. This paper has recommended, among other things, to systematically apply project influencing factors, implement a historical database for analogy and proxy-based estimating, and employ a COCOMO estimating tool to anchor estimating activities.

Possible Further Study

Among the key influencing factors, Process Infrastructure, Customer Oversight, and Management Competency did not yield explicit intersections with software process selection, or software estimating techniques. However, COCOMO's equations for "embedded" projects may very well (partially) compensate for these omissions. It would appear that these attributes would be particularly relevant to large mission-critical projects and would therefore be used to necessarily inflate software estimates for such projects. These perspectives are left as an open area for further study. Nevertheless, these omitted factors should be systematically addressed by prudent software practitioners to adjust their estimates.

References

[1] Kal Toth, "Which is the Right Software Process for Your Problem?" The Cursor, Software Association of Oregon (SAO), April 2005

[2] Barry Boehm, "Software Engineering Economics", Prentice Hall, Englewood Cliff, N.J., 1981

[3] Steve McConnell, "Software Estimation: Demystifying the Black Art", Microsoft Press, 2006

[4] Mike Cohn, "Agile Estimating and Planning", Prentice Hall, 2005

[5] Dan Brook, Kal Toth, "Levels of Ceremony for Software Configuration Management", PNSQC, October 2007

[6] Futrell, Shafer, Shafer, "Quality Software Project Management", Prentice Hall, 2002

Collaborative Techniques for the Determination of a Best Alternative in a Software Quality Environment

Dr. James McCaffrey
Volt Information Sciences, Inc. / Microsoft Corp.

presented at the Pacific Northwest Software Quality Conference
Portland, Oregon
October 13-15, 2008

== Introduction

Consider the general problem of determining one best option from a list of alternatives, where the decision making process is collaborative (performed by two or more people) rather than by a single person or by some purely quantitative technique. Examples of this type of activity in a software quality environment include a group of beta users choosing the best user interface from a set of prototypes, and the members of a company or team voting for some policy alternative. Situations where exactly one of a set of alternatives must be selected by a group of people occur in a wide range of problem domains, and different problem domains tend to use different terminology. In sociology, the study of this type of problem is usually called social choice theory, and typically uses terms like options and evaluators. In political science, analysis of these problems is often called voting theory and often uses the terms candidates and voters. In mathematics, group determination of a best alternative is frequently considered a sub-branch of decision theory. In a software development and testing environment, a mixture of terminologies is typically used.

Based on my experience, practical techniques for performing a group determination of a best alternative are not widely known in the software quality and development communities. Dozens of group analysis techniques have been studied. This paper presents five of the most common methods used for group determination of a best alternative in a software testing environment:

- The pure plurality technique
- The majority runoff technique
- The Borda count system
- The Condorcet method
- The Schulze method

Each technique is explained with an emphasis on their application to software quality. The techniques described in this paper apply to a fairly narrow range of problems and do not apply to strictly quantitative scenarios where a best alternatively is clearly defined, non-collaborative scenarios where the decision is made by a single person, or scenarios where a group of people use discussion and negotiation to arrive at a consensus decision.

== The Pure Plurality Technique

The pure plurality technique is the simplest technique and one of the most common ways to perform a collaborative determination of a best alternative -- but one which is not the best approach in many software quality situations. Imagine you are developing a Web based software application for your external use and you create four significantly different user interface designs. Although you could ask a single person to choose the best design, in most situations a better approach is to ask a group of people to rank your four different prototypes. The process of performing a collaborative evaluation, and interpreting the results of the evaluation, are a bit trickier than you might expect. The pure plurality technique is simply to submit the alternatives to a group of evaluators and allow each evaluator to select, or vote for, just one alternative. The alternative that receives the most votes is declared the winner. Suppose for example, that you submit the four different prototypes (call them A, B, C, D) to 10 evaluators, and ask each person to choose the best prototype. Now imagine the results are: Prototype A is best according to 4 people, prototype D is ranked best by 3 people, prototype C is chosen by 2 people, and prototype B is selected best by 1 person. Therefore, you select prototype A for further development. There are several problems with this approach. Suppose that while your ten evaluators are looking at the four prototypes, they are mentally ranking each alternative in this way:

A > B > C > D according to 4 people
D > C > B > A according to 3 people
C > B > D > A according to 2 people
B > C > D > A according to 1 person

Notice that even though prototype A is selected as best by the pure plurality rule, that prototype is evaluated as the worst design by 60% of your evaluation group. Furthermore, in this situation, suppose that prototype A was compared against each of the other prototypes in a head-to-head fashion. Prototype A would lose each head-to-head comparison by a 60% to 40% margin. Additionally, in some sense the non-winning opinions do not contribute directly to the final result.

== The Majority Runoff Technique

The problems with pure plurality led to the development of another group selection technique which is usually called the majority runoff system. There are many possible variations on the idea, but the winning alternative is required to receive more than 50% of the first place votes of all votes cast. This can be accomplished either by performing multiple rounds of voting, or by performing just a single round of voting but requiring the evaluators to rank all options from best to worst. For example, if evaluators vote for the best prototype design according to the data above, after the first round of voting, no prototype would have a majority. So, the list of candidates would be reduced (typically to the top two options.) In this case the top two alternatives are prototype A with 4 votes and prototype D with 3 votes. A second round of voting takes place. If evaluators vote according to their original ranking preferences above, after the second round of voting, prototype A would still receive 4 votes as best but prototype D would now receive 6 votes and be declared the winner. Using multiple rounds of voting in a software development and quality environment often has significant disadvantages compared to using a single round of voting. In some software development scenarios, multiple rounds of voting simply aren't feasible. Furthermore, because you are dealing with human beings, multiple rounds of voting can become tedious. Also, situations can arise where a group of people are voting for losing alternatives over and over until they are forced to choose an option they don't really like. Because voters are most often stakeholders in the sense that they will be affected by the outcome of the voting, you can potentially create disenfranchised voters at the end of a physical runoff scheme. So instead of performing multiple actual rounds of voting, you can just perform one round of voting but require evaluators to rank all candidates from best to worst. This allows you to determine voters' preferences in virtual rounds of voting. Requiring ranking of all alternatives has disadvantages too. The technique often isn't practical when the number of alternatives is large (typically more than eight) simply because humans have difficulty comparing large numbers of alternatives. Also, research has shown that evaluators tend to get careless with their evaluations of options that are low on their list of preferences. And, when performing multiple, physical rounds of voting, evaluators may not rank a subset of alternatives in the same order in which they rank the entire set of alternatives.

== The Borda Count System

The problems with the pure plurality and majority runoff techniques led to the development of an analysis technique called the Borda count system. The Borda count approach is very simple and one that you have almost certainly used before. Suppose there are k alternatives. Each evaluator ranks the alternatives from best to worst. A top-ranked alternative is assigned a value of k-1 points, a second-ranked alternative is assigned k-2 points, and so on, down to the last ranked alternative which receives 0 points. The Borda count is the sum of the point values for each alternative. For example, suppose there are four alternatives, A, B, C, and D, and seven evaluators. Therefore a first place ranking is worth 3 points, second place is worth 2, third place is worth 1, and last place is worth 0 points. If voting results are:

B > A > C > D according to 4 people
A > C > D > B according to 2 people
D > C > B > A according to 1 person

then the Borda count for each alternative is:

A = (4 * 2) + (2 * 3) + (1 * 0) = 14
B = (4 * 3) + (2 * 0) + (1 * 1) = 13
C = (4 * 1) + (2 * 2) + (1 * 2) = 10
D = (4 * 0) + (2 * 1) + (1 * 3) = 5

and so alternative A is selected as the best option. The Borda count technique is widely used in many problem domains, especially in sports competitions. Advantages of the Borda count technique when used in a software development and testing environment are that the technique seems to make sense intuitively, and all evaluators' opinions are taken into account. However the Borda count system has several technical problems. Suppose that you decide to remove option D, which was a clear loser. Common sense suggests that removing an (apparently) irrelevant alternative should not affect the final result of voting. However the voting data becomes:

B > A > C according to 4 people
A > C > B according to 2 people
C > B > A according to 1 person

and so the Borda counts are now:

A = (4 * 1) + (2 * 2) + (1 * 0) = 8
B = (4 * 2) + (2 * 0) + (1 * 1) = 9
C = (4 * 0) + (2 * 1) + (1 * 2) = 4

and now option B becomes the best alternative instead of option A. In other words, with this particular set of data, removing an irrelevant alternative changes the outcome. It is even possible to construct Borda count examples where the best alternative actually becomes the worst alternative. This effect is sometimes called the Borda count winner-becomes-loser paradox. There are two other closely-related technical problems with the Borda count technique. The first is the pair-wise comparison problem. It is possible when using the Borda count technique to run into situations where the Borda winner would lose against a non-winner in a head-to-head comparison. For example, suppose there are four alternatives and 12 evaluators. If the ranking preferences are:

B > A > C > D according to 7 people
A > C > D > B according to 1 person
C > A > D > B according to 2 people
D > A > C > B according to 2 people

then using the Borda count method, alternative A is selected as best with 25 points. However if you examine head-to-head comparisons you will notice that alternative B is preferred to all other alternatives (B is preferred to A by a score of 7 people to 5, B is preferred to C also by 7 to 5, and B is preferred to D again by 7 to 5.) The other problem with the Borda count technique is that it is theoretically vulnerable to evaluator manipulation. In some situations it is possible to affect the outcome of a Borda count evaluation by adding a spurious alternative that is very close (in terms of evaluators' preferences) to an existing alternative. This can have the effect of diluting close alternatives and changing the winner of the analysis. The references at the end of this paper contain examples of this effect. Despite these technical flaws with the Borda count technique, the system is quite practical often works well for collaborative policy determination by small groups in a software testing environment. The Borda count technique is generally seen by voters as fair in a subjective way, and therefore if voters are stakeholders, the people who voted for non-winning alternatives generally accept the result of the voting and do not come away with a psychological bias against the winning alternative. The point is that if you do use the Borda count technique, check your data for the effect of removing an irrelevant alternative, the result of pair-wise comparisons between the Borda count winner and non-winning alternatives, and mid-process addition of a new alternative.

=== The Condorcet Technique

The Condorcet technique for collaboratively determining the best alternative from a set of options, was developed primarily as a reaction to the head-to-head pairing problem of the Borda count method. The Condorcet method is very simple. It requires evaluators to rank all alternatives, and then a comparison of the results between each possible pair of alternatives is performed. If one alternative beats every other alternative in a head-to-head comparison then that one alternative is declared the Condorcet winner. If there is no Condorcet winner then a separate, tie-breaking technique is employed. The tie-breaking technique can be any other technique (such as Borda count or pure plurality.) The main principle of the Condorcet system is that a unanimous head-to-head winner should automatically win over any alternatives chosen by any other criteria. In practice, the Condorcet system is not often used by itself as a group evaluation technique. Instead, evaluation techniques such as pure plurality, Borda count, and others are mathematically analyzed to see if they satisfy the Condorcet principle in all cases.

== The Schulze Method

One of the most interesting voting systems which satisfies the Condorcet principle has a variety of names including the Schulze method, the clone-proof Schwartz sequential dropping technique, and the beat-path method. The Schulze method is somewhat more complicated than the other techniques presented in this paper. The Schulze method is similar to the Condorcet system in the sense that the Schulze algorithm checks to see if one option dominates all other options. However, instead of directly using raw ranking data like the Condorcet system does, the Schulze method performs a head-to-head comparison using an indirect measure of the strength between alternatives. This indirect measure is called the Schulze path strength. The technique is best explained by example. Suppose you have four alternatives, A, B, C, and D, and 11 evaluators. And suppose the voting data is:

A > B > C > D according to 4 evaluators
B > C > D > A according to 2 evaluators
C > D > A > B according to 3 evaluators
D > C > B > A according to 1 evaluators
C > A > B > D according to 1 evaluators

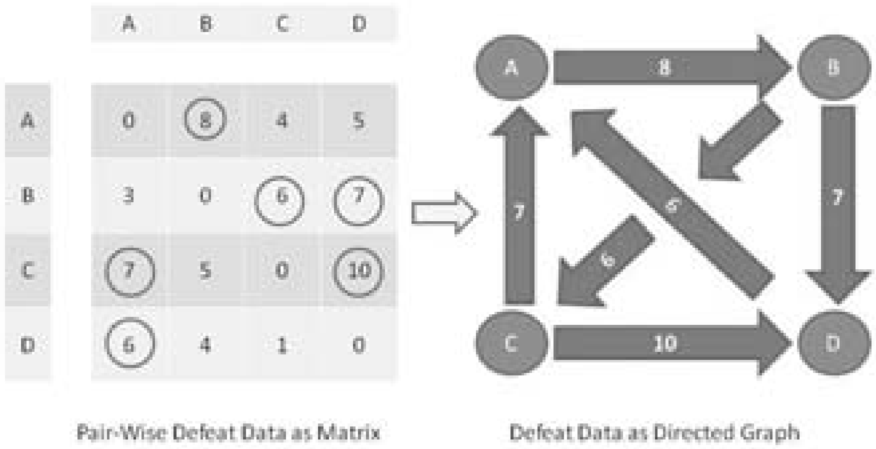
The Schulze method allows evaluators to rank options equal to each other, and also allows evaluators to leave out alternatives altogether. The first step in the Schulze method is to construct a matrix of pair-wise defeats. For the raw voting data above, the matrix of pair-wise defeats is:

```
0  8  4  5
3  0  6  7
7  5  0 10
6  4  1  0
```

The first row of the matrix means option A is preferred by 8 voters over option B, preferred by 4 voters over option C, and by 5 voters over option D. The second row means option B is preferred by 3 voters over option A, by 6 voters over option C, and 7 voters over option D. The third and fourth rows, for options C and D, are interpreted similarly. The pair-wise defeats matrix has 0 values on the main diagonal because these values are comparisons between an option and itself. At this point the Condorcet system would check this direct pair-wise data to see if there is a unanimous head-to-head winner but the Schulze method derives an indirect measure of strength between alternatives called the path strengths. For this example the path strengths are:

```
0  8  6  7
6  0  6  7
7  7  0 10
6  6  6  0
```

This is the key idea behind the Schulze method and is a very clever concept. The first row of this matrix means the strength of the path from option A to option B is 8, the strength from A to C is 6, and the strength from A to D is 7. Explaining the idea of path strength is best done visually. We can convert the raw ranking data to a matrix of pair-wise defeats. Next we can conceptualize the pair-wise defeats matrix as a directed graph as shown in Figure 1.



	A	B	C	D
A	0	8	4	5
B	3	0	6	7
C	7	5	0	10
D	6	4	1	0

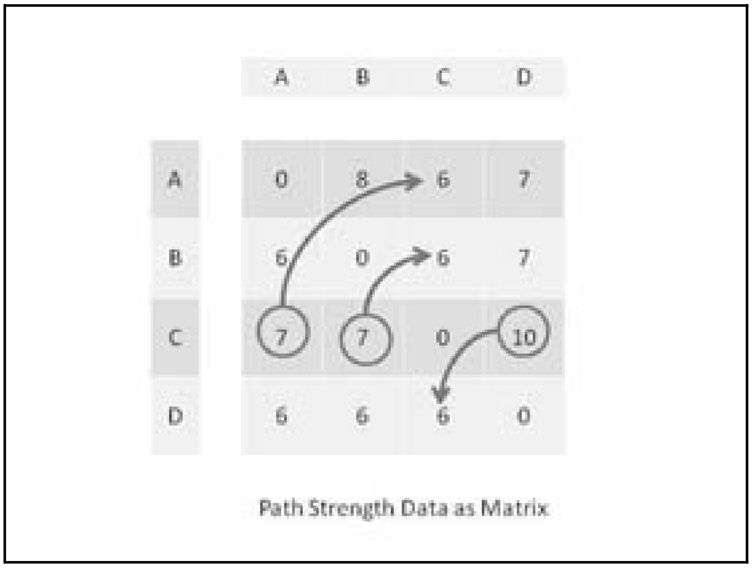
Pair-Wise Defeat Data as Matrix

Defeat Data as Directed Graph

<Figure 1 - Pair-wise Defeats as Matrix and Directed Graph>

The arrow from node A to node B means that option A is preferred by 8 evaluators over option B. We do not put an arrow from node B to node A because option B is preferred by only 3 evaluators over option A, so B dominates A. In this example, all the arrows are unidirectional but if two nodes are preferred by equal numbers of evaluators then you can draw a bidirectional arrow. Now consider the path between node/option A and node/option D. A "beats" B by 8 which in turn beats D by 7. We choose 7, the smaller of these two values, to represent the overall strength of path between A and D. The idea is that the overall strength between two nodes which have intermediate nodes is best represented by the smallest direct node-to-node strength. Notice that we can also take a different path from A to D and say that A beats B by 8, and B beats C by 6, and C beats D by 10, and select the smallest number, 6, as the path strength. In situations where there are multiple paths between nodes, you select the largest path strength, so in this case the final path strength between A and D is 7.

Computing the strength of each pair of nodes can be performed by a variation of the Floyd–Warshall algorithm. The standard Floyd–Warshall algorithm finds the shortest paths in a weighted, directed graph. Once the path strengths have been determined, you can determine the Schulze method winners by checking to see if there is any option where its path strength is greater than or equal to all other options' corresponding path strengths. In this case option C is the only Schulze winner because option C's path strength to option A is 7 (but option A's path strength to B is just 6), option C's path strength to option B is 7 (but C's path strength to C is just 6), and option C's path strength to option D is 10 (but D's path strength to C is just 6). On the other hand, option A is not a Schulze winner because option A's path strength is better than option B (8 to 6) but option A's strength is worse than option C (6 to 7). Figure 2 shows the matrix of path strengths that corresponds to the data in Figure 1 and demonstrates that option C is a Schulze method winner. Visually, an option is a Schulze winner if every path strength value on that option's row is greater than or equal to the corresponding value across the matrix's main diagonal.



	A	B	C	D
A	0	8	6	7
B	6	0	6	7
C	7	7	0	10
D	6	6	6	0

Path Strength Data as Matrix

<Figure 2 - Path Strength Data>

Although the Schulze method can be easily performed by hand, the process can be error-prone. Therefore, in practice the Schulze method is usually performed using a software tool. The Schulze method has strengths and weaknesses. One weakness of the Schulze method in some scenarios is that Schulze is not immediately intuitive. In situations where the evaluators are stakeholders, the Schulze method can possibly be viewed as a black box technique that produces a magic result. The primary advantage of the Schulze method is that is has been extensively analyzed and has been shown to meet many favorable criteria. Recall that the Borda count system can sometimes be affected by the removal of an irrelevant alternative. There are many voting system criteria. For example the monotonicity criterion can be loosely stated as the principle that a winning option cannot become a non-winner by one or more evaluators ranking that option higher. There are many proposed voting system criteria, and the Schulze method has been shown to meet most of these criteria. An interesting exception is that the Schulze method violates what is called the participation criterion. In informal terms, it is possible to add to an existing system, new voters who prefer the current winner to some other alternative, which results in the less preferred alternative becoming a Schulze winner. In spite of its shortcomings, based on my experience, the Schulze method is often very effective, especially in situations with a large, educated group of evaluators (meaning they have knowledge of the underlying Schulze method and therefore do not view the algorithm as mysterious) who are determining a policy alternative (as opposed to a product decision) and who are stakeholders in the final result. For example, the Schulze method is used by several Open Source groups to determine general policy decisions and to elect officers. In software development and quality scenarios, the Schulze method is generally an excellent technique to employ.

== Conclusions

When performing a collaborative determination of a best alternative, the interpretation of exactly what *best alternative* means is entirely defined by the approach used. Each technique has pros and cons and there is no single best approach in all situations. Some of the key factors you should consider when using collaborative evaluation techniques include the number of alternatives, the number of evaluators, whether the alternatives are policy decisions or product decisions, and the extent to which evaluators are affected by the final result of the analysis. In general, you should use more than one collaborative analysis technique when possible in order to use multiple results to cross-validate your analysis. If multiple techniques yield the same conclusion, you gain some measure of confidence in your results. If multiple techniques do not yield the same results then you are well advised to reexamine your assumptions and try and determine why you are getting different results. The situations where the collaborative techniques described in this paper are used are often quite subjective so your experience and intuition should play a big part in interpreting your results. In other words, you should not blindly accept the results of any collaborative technique.

As a general rule of thumb, the pure plurality technique is only effective in situations where ranking all alternatives is simply not feasible. Because the majority runoff technique has the potential to alienate up to 49% of your evaluators, you should generally use this technique primarily to supplement other techniques. The Borda count system, in spite of technical flaws, works quite well especially when the evaluators are stakeholders in the analysis. The Condorcet principle is very simple, and so can often be used to validate the results of other techniques. The Schulze method is an excellent general purpose technique in many software quality situations except when you are dealing with a relatively small group of evaluators who are influenced by the results, or evaluators who do not understand the Schulze algorithm.

Acknowledgment: I am grateful to Joshua Eisenberg (McAfee, Inc.) and Moss Drake (Dentists Management Corp.) who reviewed this paper and made significant improvements to its quality.

== References

Gaetner, Wulf. "A Primer in Social Choice Theory", Oxford University Press, New York, 2006.

McCaffrey, James. "Group Determination of a Best Alternative in Software Testing", MSDN Magazine, November 2008 (in press).

Nurmi, Hannu. "Comparing Voting Systems", Springer-Verlag, 1987.

Saari, D. G. "Basic Geometry of Voting", Springer-Verlag, New York, 1995.

<end of document>

Quality Software Engineering: *Collaboration Makes the Experience*

Diana Dukart and Brian Lininger

Abstract

Well-executed collaboration can often make the difference between a quality software system and one that falls short. Strong collaboration skills are necessary for the varied roles software engineers are required to apply when undertaking software and information technology projects. In fact, close coordination and communication is needed in all aspects of the software process, from the initial customer requirements definition through the detective-like collaborative work required to triage and resolve errors in the final system. Any flaw in these lines of communication can greatly increase the risk of diminished quality in the end product.

During the process of completing the Oregon Master of Software Engineering (OMSE) Practicum Project, the authors applied a variety of collaboration styles and technologies commonly practiced on software engineering projects today. Project aspects addressed by such practices include distributed team member location, variability of member experience and skills, multiple modes of stakeholder integration, and constrained schedules and resources.

This paper examines the "lessons learned" from the full range of collaborative styles and technologies that were encountered during the project. These insights will provide other software professionals ideas and guidance on how to navigate similar challenges in their future collaborative software projects.

About the Authors

Diana Dukart is a senior software engineer with professional experience at Tektronix, Inc. She is a recent graduate of the Oregon Master of Software Engineering (OMSE) program at Portland State University and holds a bachelor's degree in Computer Science from the University of Portland. Email: diana.dukart@comcast.net

Brian Lininger is a senior software engineer at Oracle, where he performs Quality Assurance activities on the Fusion Middleware Application Server. He is a recent graduate of the Oregon Master of Software Engineering (OMSE) program at Portland State University and holds a bachelor's degree in Computer Science from the California State Polytechnic University at Pomona. Email: Brian.Lininger@Oracle.com

Introduction

Software development is inherently a social activity, as are most other types of product development. The core activities in any kind of product development are: understanding the requirements, defining the product specifications, and validating the correctness of the final product. Fred Brooks describes software development as:

> "The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocation functions… *I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation."* [1]

As Brooks states, the critical part of any software development project is the "conceptual construct". For most software projects, a team of software engineers performs the project development, so the "conceptual construct" is a collaborative product influenced by each interaction of the members on the team. As a result, the collaborative experiences of the team will have a profound impact on the outcome of the software development project. Below are the experiences of a team of software engineers as they collaborate in the completion of a software development project as part of their educational practicum in software engineering.

Project Background

OMSE Overview & Requirements

The Oregon Master of Software Engineering (OMSE) program at Portland State University (PSU) is a graduate level software engineering education program designed for working software and information technology professionals. It provides participants both breadth and depth in the application of principles, methods, processes, and tools used in the dynamic and evolving software industry.

OMSE offers degrees, certificates, and professional courses tailored for working professionals. The degree program requires completion of 16 courses including a two term practicum experience. The practicum offers hands-on management and development experience in applying the skills learned, a variety of collaboration styles, and technologies commonly applied to software engineering projects today. Practicum participants are guided throughout the process by a member of the OMSE faculty and often work directly with an industry sponsor.

Sponsor Overview

Lifecom was the industry sponsor for the authors' 2008 practicum project. They are a small Portland business developing software for clinical cognition patient diagnostics. Lifecom is revolutionizing medical informatics and patient care by developing a new form of artificial intelligence to provide knowledge, quality assurance, and safety to medical practitioners. This ground-breaking software system is directly used for patient care as well as reducing administrative costs. With integrated patient records as well as built-in artificial intelligence using a large and current knowledge warehouse, Lifecom provides improvements on accuracy and timeliness of medical diagnosis.

Project Overview

As with all software systems, the Lifecom system needs to be updated periodically as new medical knowledge is obtained and added, as well as other enhancements. The practicum project was centered on this opportunity presented by Lifecom.

The Lifecom system runs on tablet computers used by medical practitioners. The applications running on these on-site tablets are what Lifecom needs the ability to update in a secure and timely manner. For these reasons, the practicum team named their project the Secure System Provisioning Project.

In addition to the main functional goal of updating the tablet applications, there are several ancillary goals. These include dealing with firewalls, running over the internet and the intranet, deploying custom updates for a specific customer, pushing out emergency updates, validation of file integrity, and producing an audit trail and version report.

Team Overview

The practicum team for this project was made up of four individuals who were completing the OMSE degree. The dynamics of this team lent well to the opportunity of gaining a wealth of collaboration experiences. These opportunities included:

- Distributed Team Communication
- Team Members with Different Backgrounds
- Learning a New Industry Sector
- Converging as a Team
- Team Expectations
- Work Product Management
- Working with Stakeholders

The following sections will delve into greater detail in each of these areas, providing a summary of the team’s experiences during the project, their lessons learned and recommendations on how fellow engineers might approach similar challenges. Given that this was a newly formed team with members who had never worked with each other before, the experiences will generally be more valuable when forming a new team rather than for teams already working together.

Collaboration Experiences

A Distributed Team

Experiences
The OMSE program allows people to take classes remotely. For this practicum team, one member was based out-of-state with a job that took him all over the country during the course of the project. From the beginning, the team needed to figure out how to bring this individual into the project and allow him to contribute along with the rest of the team.

In addition to a permanently remote member, travel was necessary for another member of the team taking him overseas for three weeks during the project. Since the team was already set up to communicate remotely (at this point in the project the team regularly communicated via Skype), the travel was initially viewed as “not an issue”. Unfortunately, the team did not anticipate or plan for electricity shortages limiting communication with the team member traveling internationally. The result was delays on planned work.

On top of this, a third member of the team was “unavailable” for several weeks due to his child’s birth. As with the out-of-state member, this was planned from the beginning of the project. Unluckily for the team, it occurred the same time they were missing their other member overseas.

Lessons Learned: Communication & Risk Management
It is critical in planning for a project to include issues that may come up outside of the project. Risk management is essential in being able to control the many aspects that come up when running a project such as staying on schedule, unplanned events, and breakdowns in communication.

In general, the team did well in learning the new technologies to make remote communication work. Skype was chosen and integrated early in the project and it worked nicely for team meetings as long as the infrastructure was in place. The unfortunate experience of relying on Skype overseas when brownouts were occurring was not a problem with the technology, but rather a problem of the moment. In this case, moving the responsibilities of the critical tasks from the member traveling overseas to a local member temporarily would have been ideal. Although this was not done for this particular project, the team viewed it as a lesson learned to be used when executing future projects.

This lesson was similar to the other case of having a team member gone for family reasons. Planning ahead of time to move responsibilities to other available team members would have been easier for the members left with the work and possibly even kept the project on track. In addition, there could have been better scheduling on completion of work in anticipation of planned absences.

There were also lessons learned in working with people remotely. For certain tasks, such as brainstorming solutions, the optimal work environment is face-to-face communication. Ideas can be diagramed to communicate ideas more quickly, as well as non-verbal communication coming into play with hand gestures and facial expressions. The same work can be accomplished through remote means, but it takes more time. This extra time needs to be planned into the project schedule.

Recommendations

When a team includes remote members, there are several details that should be taken into consideration. First, any outside demands on the time or placement of team members needs to be included in the project plan. Communicating remotely must rely on many pieces of the system including not only the software, but the entire hardware infrastructure. With this being the case, there must be a contingency plan for when things do not work.

Second, the plan should always include a backup member to each person responsible for a critical piece of the project. This is to prepare for those unplanned events that take a person away from the job. The backup person can then step in and keep the project going and on schedule. When this occurs, completing the work the backup person had to put aside to pick up the critical tasks also has to be addressed. Although this type of issue can be different for each project, many times there is flexibility in when the non-critical work has to get done and a backup person at least keeps the project moving forward.

Lastly, extra time needs to be scheduled into the plan for communication. Although technologies like Skype are available and do aide in remote communication, it will never be as efficient as face-to-face meetings for some tasks.

Varied Backgrounds

Experiences

Each team member came from a different background. This included different industry sector experience as well as work experience and educational backgrounds. This diversity was an asset to the team because of the many skills that could be leveraged. However, at the same time, it became a stumbling block as people started assuming what they thought other team members already knew.

An example was the decision to use Web Services for a primary piece of the project solution. One member of the team already knew this technology intimately, but the others did not. Assuming the others knew more than they did, the knowledgeable member sometimes skipped critical details when explaining pieces of the solution. This caused confusion and misunderstandings within the team and negatively affected the necessary collaboration efforts.

This is just one example of many assumptions that were made throughout the project and each team member can claim at least one mistake. As teams try to gel and get to know each other, issues like this will arise and the experiences of this team were no exception.

Lessons Learned: Getting to Know Strengths and Weaknesses

Team members getting to know each other and learning each other's strengths and weaknesses in the beginning can help alleviate problems later on in a project. This not only includes learning which technologies each person knows, but also learning processes, industry experiences, and expectations of each member. Only after this initial assessment of the team is complete can the team begin to start assigning roles and responsibilities that really leverage the team's skills.

Collaboration can only take place when every member is in full understanding of the problem as well as knowing where each team member fits into the big picture. Establishing that foundation is essential for the success of a project as well as for the quality of the end product. This is especially true with such a diverse group of individuals who had never worked together on a project before.

Recommendations

When a new team comes together, they need to get to know each other first. A project kickoff meeting is a good first step in this direction. It not only presents the initial overview of the problem the team needs to tackle, but also begins the socialization between the team members.

A kickoff meeting is a good place for team members to talk about their background and experiences in the industry. Since this information is so important for a new team, the meeting agenda should explicitly include this discussion. Each member needs to bring up relevant knowledge that can help the project as well as areas where they have an opportunity to learn.

New Industry Sector

Experiences

Medical informatics was the industry sector of the practicum project. This is an exciting field to be involved in today because of the amount of growth and the endless opportunities it presents. In this case, no one on the team had any experience in this sector.

Fortunately, many of the skills needed to solve software problems are applicable across industry sectors. In fact, one of the main goals of the OMSE program is to teach such skills. The team was able to use this commonality to leverage the common language and processes learned in the OMSE program coursework. Each team member had completed the same core classes and this common foundation provided a framework to identify the important issues of the project confronting them.

The first important issue confronting the team was to understand the context and the goals of the project. Research became an important starting point in which all the members contributed. This built the foundation to the requirements elicitation and documentation efforts. This research not only included the specific problem domain, but also the medical informatics industry sector as a whole.

Lessons Learned: Learning Opportunities

The computer industry has grown a staggering amount since the early days of Alan Turing (English mathematician and philosopher, widely known as the father of modern computer science). [3] Used daily by most of the population, software has become a standard way of life. It is not plausible that any individual could know everything there is to know about every sector in the software industry. However, continuous improvement of software skills is a requirement because of the constant change. Therefore, learning about a different industry sector is an essential part in the career of software engineering. This project was just such a learning opportunity for the team.

Starting out as a team, the work of deciding which project to carry out or the initiation phase was completed. The team researched and read papers on current work in the medical informatics field. Challenges and future needs also came up in this exploration such as creating a National Framework for Healthcare and Electronic Medical Records. There are many laws and politics around this field as well such as the Health Insurance Portability and Accountability Act (HIPPA). None of these things actually became pertinent because of the nature of the project; however, the team did use what was learned in knowing what questions to ask the customers. It was only after asking questions, that the team knew things like HIPPA did not apply to the problem space.

Recommendations

Learning as much as possible about a project's industry sector is invaluable when eliciting requirements from the customer. Experts in any field have a tendency to leave out details that seem obvious to them, but are vital to the understanding of the problem for others not as well versed in the subject. Therefore initial research becomes essential before meeting with the customer. Once requirements elicitation begins, further research is also needed as subjects come up that are unclear to the engineers involved.

Converging as a Team

Experiences

Being able to successfully converge as a team in many ways comes down to having good communication, working together rather than as individuals, and having good leadership. A breakdown in any of these things can lead to floundering in a team's attempt to collaborate.

There are many ways in which a team can communicate; such as email, voice and video communication, and direct human interactions. While email is a good medium for transporting binary data, it falls short in communicating other important cues, such as sarcasm or voice inflections. It is important for teams to communicate regularly via a live mechanism. In addition, even voice communication (such as Skype) is not as productive as direct human interaction. Since the team was constrained to certain communication styles because of remote members, communication became a big challenge.

For one thing, voice as well as video still lacks the ability to whiteboard ideas, which is critical for architectural and design discussions. The team had no choice for their project and discovered that work can be accomplished electronically; however, it took more time than initially planned. Fortunately, other work tasks, such as requirements and planning, are more amenable to voice and video and did not impact the schedule.

The team converged in other ways, for instance, handling the administrative project tasks such as note-taking and configuration management. They decided to rotate these tasks, which worked well in getting acquainted with each other's work and forcing the members to work together.

Leadership was also rotated during the project. Each member was responsible for taking the lead role six weeks during the project. This gave each member a chance to exercise the leadership skills they had acquired through work experiences and skills learned in the OMSE program.

Lessons Learned: Communication and Management

There was some confusion and misunderstandings on project goals. Since the remote member could not attend local meetings with the customer, there was a limitation on how much he could take away from the notes written by the other members. In addition, the communications between the team members themselves were constrained to email and voice communication. In this case, more could have been done to explicitly talk about each member's assumptions and expectations on project goals. Not understanding each other's point-of-view led to some problems later in the project that could otherwise have been worked out earlier if better communication had happened. In some ways this was the result of the communication medium and in others; it was the result of the team learning how to work together.

As far as rotating tasks, the team learned that it worked okay for the small group of four, but did not feel it would work for a bigger team. In fact, four people were sometimes difficult to manage because of time constraints and other outside factors and responsibilities causing delays in communication and completion of work. A four-member team is close to the limit for this type of project management style to work.

Rotating the team leadership position did not work well in this project. There needed to be one person who always focused on the big picture in order to keep the work on course. For instance, when team members had conflicting project goals, a person keeping an eye on the big picture could have picked up the signs of this conflict earlier than actually occurred. In rotating out the leader, this big picture focus was a little different for each cycle. In addition, setting up meeting agendas and following up action items could have happened better with a single person in this role.

Recommendations

There needs to be a single person with the specific role of focusing on the big picture and following up on work tasks. In line with this, roles and management style need to be adjusted for the type of project being addressed. In this case, the team was small, so rotating tasks worked. When this role is rotated among team members, there is a variation in the big picture view that will affect the direction of the project. The variation in task assignment and tracking leads to differing expectations among the team members, causing different working assumptions for the team members.

Communication of project goals and expectations should happen at the beginning of a project with these subjects explicitly called out. In addition, this type of communication should be revisited throughout the project to make sure everything is still on track and everyone is in agreement.

Team Expectations

Experiences

Whenever people are brought together for a project, each person brings with them unspoken expectations. These expectations can include gaining proficiency in a new technology or learning a new role in the organization. Unspoken expectations can work against a project if they remain unspoken, or if they are voiced in advance, can be leveraged to benefit the team.

This project team was no different in their possession of unspoken expectations. In some cases, expectations had no bearing on the outcome of a project; however, in this case a critical expectation was missed regarding how much the team could accomplish. After a scope for the project had been determined, the team moved forward with the project tasks but diverged approximately halfway through the project schedule. The problem was differing expectations of what the team could accomplish. Half of the team had set higher expectations and were proceeding to execute the project tasks in a manner that would allow them to exceed their planned scope. The other half of the team was executing their tasks to complete the project within the scope originally planned.

The result of these mixed expectations left the team frustrated and forced them to stop and evaluate their progress. It was at this point that the discussion of expectations occurred, taking into account the current state of the project. With this new insight, the team was able to refocus their efforts and complete the project as originally scoped.

Lessons Learned: Setting Expectations

Like all projects, communication is critical for success. It is important to identify the expectations of each team member at the outset of the project, as this will impact both technical and managerial decisions. In this instance, it would have been very useful to identify the differing expected project outcomes of each team member early in the project, as this could have been used by the team to avoid conflicts and manage project tasks more effectively. Knowing that half of the team had higher expectations would have allowed the team to partition their work in such a way that all members were satisfied.

Recommendations

As already discussed in the Different Backgrounds section, initiating a project kickoff meeting is a good start to any project. In this case, a discussion of each team member's expectations for the project needed to be included on the agenda as it would have averted much of the frustration felt by the team. The meeting needs to encompass all of the team members, but should be as hierarchy free as possible. The goal of this meeting is to allow the members of the team to state their expectations for the project so they can be used in planning and performing project tasks.

This exercise should not be used for employee evaluations but solely for the purpose of managing the expectations between team members and for allocating project tasks. In short, the main reason for knowing expectations is to gain insight and understanding of team members. In this example, differing expectations could have been the result of a team member expecting a new child during the project or another team member wanting a promotion as a result of the project's success. To get everyone working in the same direction, these expectations need to be explicitly discussed and managed.

Work Products

Experiences

Developing the Work Products within a distributed team can be challenging. Each team member was responsible for updating and/or reviewing each document multiple times during the course of its evolution. As each member of the team maintained a full-time work schedule and family life in addition to participating in this project, the schedule of each member was variable as well. As a result, the team needed a mechanism for managing the work being done on several documents by different members of the team on different schedules.

Lessons Learned: Document Management

Several mechanisms of collaborative document management were tried. One member used GoogleDocs for a document draft so that each team member could work on the document whenever it was convenient. The main issue with this was the rest of the team being unfamiliar with the tool and therefore, a learning curve had to be overcome before the entire team was effective using it. This included overcoming the security mechanisms that are necessary for protecting shared work.

Utilizing the merging capabilities of most source-code repository tools is another mechanism commonly used and was tried by the team. The team ran into issues; however, when any type of non-text document was used. A significant number of the documents the team produced were not simply text, but utilized diagrams and pictures to convey the intent and details of the system. As a result, a significant amount of work had to be redone when the repository could not resolve the conflict in the document versions.

One mechanism that worked well for the team was to use a Round-Robin style of work product updates. When work was required on a document, an explicit sequence was set for members to complete their updates. Each member would receive a notification that they could modify a specific work product and when their work was completed, they would notify the next team member in the sequence. This worked remarkably well for the team and became their default mechanism for the remainder of the project.

Recommendations

For a small sized team, a Round-Robin style of document management can be useful and effective. However, its use must be planned and explicit. A planned sequence must be stated at the outset and members must be held accountable for their work completion time as it can delay the entire team's work. Including the entire team on work completion notifications keeps the team informed of the current state and enforces a 'social justice' among team members. As already discussed, problems with communication or unavailability of team members must be part of the project plan as well and dealt with on a case-by-case basis.

Additionally, the ability to use common tools across a team cannot be ignored. If a process can be devised that allows the team to utilize their current tools and still perform the collaborative work necessary, this would be most desirable.

Multiple Stakeholders

Experiences

An experience of the team, which is likely common among most projects, was the existence of multiple stakeholders. In this case, there were two stakeholders that were looking for different outcomes from the project.

The first stakeholder, the project sponsor, was much like a typical customer. They provided as much input on system requirements as possible, participated in reviews, and provided feedback wherever possible. Their expected output from the project was the same as most customers, in this case, a set of work products that they could utilize in their company to extend their product-line. They had little interest in the project management side aspect, such as the work that went into generating a useful project plan and requirements document. They were interested in the end products: architecture, design, and prototype of the system.

The second stakeholder, the practicum instructor, was evaluating the project from a different perspective. In addition to providing input and evaluating the set of work products, the instructor's goal was to evaluate the team's effectiveness working together and their application of the knowledge the team had acquired during their coursework. The output from the team for this stakeholder was status updates and presentations that communicated the how and why the team had made certain decisions. In addition, the practicum instructor was interested in the internal work product qualities, such as the project plan and requirements documents, as part the evaluation process.

Lessons Learned: Stakeholder Input Benefits

Any software product that is going to be competitive in today's world needs a set of unique features that set it apart from what is already available on the market. With this being the case, each software project presents learning opportunities, such as applying software a little differently to a common problem or learning about a different industry sector.

In this case, the team learned both of those things and much of this input came from the project sponsor. It was a great opportunity for the team to learn from individuals who had been in the medical informatics sector for years. In addition, they were able to take other experiences in applying software updates and learn how to apply that knowledge in a new and different way.

As well as the input benefits from the project sponsor, the team also leveraged and benefited from the OMSE instructor's input. With this guidance and involvement, the team was able to maintain a macro-project focus and utilize processes that prevented them from falling into the typical pits of micro-task optimization.

Another learning experience for the team was dealing with the occasional conflict between the stakeholder inputs. In one instance, a disagreement of the detail of the project background came into question. One customer had an in-depth knowledge of the project domain, while it was new for the other. The lesson here was learning how to handle the expectations and needs from two very different perspectives. In this case, the team chose to include more detail so everyone could understand the project, even though this included details the one stakeholder already knew. In this way all parties were working from the same project basis.

Recommendations

Whenever a project has multiple stakeholders, there are two options:

1) View the different inputs as liabilities, or
2) View them as an opportunity to improve the project outcome.

In this case, the additional stakeholder inputs were an asset leveraged for the success of the project. Stakeholders provided information to the team that prompted them to ask further questions, define further requirements, and expose more views of the system.

As a result, the final product for both customers turned out more comprehensive and of better quality, a real-life example of “the whole is greater than the sum of the parts.”

In addition, differing inputs between stakeholders is a common occurrence that engineers will need to address. Collaboration skills again come into play, as the engineer must work with multiple stakeholders to come up with a solution. The job of the engineer after all, is to solve problems.

Conclusion

Collaboration is an essential piece to any software project for reasons of the inherently social activities involved. As such, there will be many experiences with lessons that can be taken away. In the case of the practicum project described in this paper, these lessons included learning how to work with remote members, leveraging different backgrounds, learning a new industry sector, converging as a team, handling team member expectations, managing project artifacts, and valuing customer input.

It is always less painful and more efficient to learn from the experiences of others. With the descriptions of experiences, lessons learned, and recommendations; this article hopes to assist professionals in similar circumstance to improve and enhance the quality of software project knowledge and understanding for the future.

References

[1] Brooks, Fred. “No Silver Bullet – Essence and Accidents of Software Engineering.” Proceedings of the IFIP Tenth World Computing Conference (1986): 1069-1076.

[2] Dukart, Diana, Lininger, Brian, Pierson, Derek, and Subramanian, Mahesh. “Secure Provisioning System.” OMSE Program Practicum Project. Portland State University, 2008.

[3] Lewis, Harry R., and Papadimitriou, Christos H. Elements of the Theory of Computation. New Jersey: Prentice-Hall, 1998.

Collaborative Change

Debra Lavell
Intel, Corporation

Abstract

At Intel, we have implemented a more effective approach for capturing and sharing lessons learned through retrospectives. A retrospective is a ritual where a team comes together several times during the lifecycle of the program to discuss what is working well and uncover opportunities for improvement. Retrospectives are a powerful way to help teams apply the wisdom learned with the intent of long-term behavior change.

As with any organization or business process change undertaking, one of the most difficult challenges to overcome is getting an entire company to change their culture and modify the way they work and behave. Introducing the retrospectives methodology into Intel has been no different. In this paper, we explore the people side of bringing retrospectives into an organization. The tips and tricks are from a facilitator's perspective. This paper will explore how one handles the human aspects of introducing collaborative change.

Bio

Debra has over 10 years experience in quality engineering. She currently works as a Program Manager in the Corporate Platform Office at Intel Corporation focusing on Retrospectives and Organizational Learning. Since January 2003, Debra has delivered over 200 Project and Milestone Retrospectives 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 Software Process Improvement Network Steering Committee. For many years, she was responsible for securing the monthly speakers. She currently is the President of the Pacific Northwest Software Quality Conference, Portland, Oregon. She holds a Bachelor's of Arts degree in Management with an emphasis on Industrial Relations. To contact please email: Debra.S.Lavell@intel.com

Introduction

Due to the increased complexity of our product and service development processes at Intel, as well as the expanded use of globally dispersed development teams, we have implemented a more effective method for capturing and applying our program and project lessons learned. This approach is known as the retrospectives methodology. A retrospective is a ritual where a team comes together several times, (we recommended three strategically placed) during the lifecycle of the program so the team can pause and reflect on what is working well (so they can keep doing it) and discover collaboratively where there are opportunities for doing things better (start or stop doing it). Retrospectives are a powerful way to help a team apply the wisdom learned with the goal of long-term behavior change.

As with any process change, one of the most difficult challenges to overcome is inspiring an entire organization to change their culture by collaboratively changing the way they work, and how they behave. Introducing the retrospectives methodology into Intel has been no different. This paper will explore the people side of bringing retrospectives into an organization. The tips and tricks are from a facilitator's perspective focusing how to handle the human aspects of introducing collaborative change.

Changing human behavior

One reason organizational change is so difficult, is that it requires sustainable changes in how people do their work and how they behave while performing that work. By working collaboratively, the team members identify what needs to change and what the team is willing to commit to do (rather than management dictating what they want to see change). This "wisdom of the crowd" deciding what is the most important to the success of the team dramatically increases the likelihood the change will happen. Introducing an effective retrospective methodology within an organization requires the management of many facets of change:

Adoption of a new way of doing things. Many program and project teams have never experienced continuous process improvement efforts through multiple retrospectives across the lifecycle. They normally wait until the end (typically called a Post-Mortem or Post Project Review) to discuss what happened, which is too late to affect the current program. Most team members move quickly on to other teams and are not interested in spending a day reviewing what happened on the program. The retrospective methodology requires a series of events where team members collectively meet face-to-face (if possible), at strategic points during the development lifecycle to discuss what is working and what needs to be improved. Collaborative discussions and activities are the basis of a retrospective.

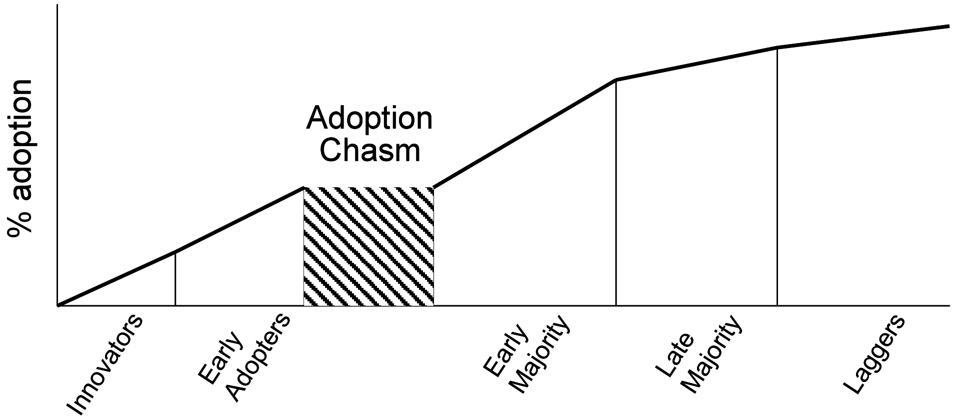
Management of a new deliverables. After a post mortem, some team's output is a bulleted list of items, which is never reviewed or discussed again. The deliverable from a retrospective is an action plan, written collaboratively by those who have a passion to fix the problem. Every action plan has next steps broken down into tasks with an owner, who is passionate about ensuring the change is implemented. The team manages the completion of the action plan by adding it to their weekly product development meetings, monthly goals, and possibly roll up to a group level as IMBOs (Intel Management by Objective) to ensure the changes desired are visible and progress tracked.

Updating existing systems. To ensure retrospectives and the lessons learned that came from them become institutionalized, work systems need to be developed or existing work systems must be modified to incorporate organizational learning as part of their normal processes. For example, we recommend three milestone retrospectives within the Product Life Cycle, which is the common development lifecycle framework that most teams utilize to manage their programs and projects. Before a team can exit a particular phase or milestone, a retrospective is held to capture learnings. Change happens when collaborative discussions facilitate a culture of "this is how we work to constantly improve."

Replacing old behaviors. This is the most challenging aspect of change. If you want to get the most out of retrospectives, it will require a change the way people work and associated business practices. When the team collectively decides what behaviors need to change, it becomes self managing and self policing. Nevertheless, not without an occasional reminder to help spark forward progress. At three, six, nine, and twelve months, reminders are sent via email and if no response, a 30 min meeting is scheduled to discuss what progress has been made in the implementation of the new behavior. This means that the human side of change has to be managed in order for it to become institutionalized.

Why does institutional change need to be managed?

In order for retrospectives to become a broadly accepted and utilized method for capturing program and project key learnings within an organization, and to see measurable change, the majority of the organization must acknowledge and embrace the change, as something they believe will benefit them. It is not uncommon for new ideas and ways of doing things to be tried, and if successful, accepted by a percentage or slice of any organization known as the "Innovators" and "Early Adopters." However, gaining widespread support for an idea can be a much tougher problem to solve. Below is Figure 1, where Geoffrey Moore and his book titled "Crossing the Chasm" describes the challenges gaining institutional acceptance of retrospectives within Intel. 1



Source: Crossing the Chasm (author modified)

Figure 1: Modified Version of the Technology Adoption Life Cycle

As described by Moore’s model, a chasm exists between people and project/program teams willing to experiment with retrospectives (innovators and early adopters), and the remainder of the organization that may not be willing to implement it as a standard method of improving their organizational learning capability (early majority, late majority, and laggers).

Collaborative discussions among the team members of what needs to change and securing advice and support from an influential sponsor, is a critical aspect for crossing the adoption chasm. In addition, fundamental to the process is having an understanding that the value proposition for method changes such as retrospectives is different on each side of the chasm. To the left of the chasm, the value proposition needs to be sufficient to convince the innovators and early adopters to experiment with the method. However, if you want to achieve broad adoption by the remainder of your organization, the value proposition on the right side of the chasm must be sufficient to convince the majority to change the way they collect and disseminate program learnings – a much more difficult task.

As part of our retrospective process, we schedule time to share the results of a retrospective with senior management so they can provide support to the team and remove any barriers to implementing change. It is always helpful to look at an example to better understand the impact of sharing the outputs from a retrospective with management. During the execution phase of a major platform, the Program Manager decided to “try out the new process.” After the retrospective, we held a management report out, the purpose of the presentation was to share the recommendations from the team (action plans) that resulted from the collaborative discussions, and to gain senior management support for broad adoption of the methodology across the business unit.

It quickly became apparent the senior manager was satisfied with the current method of performing a single post-program review at the end of the program, after launch of the product. The retrospective output was good, but it did not satisfy the value proposition for broad deployment. The significance of the methodology became obvious during the meeting when the collective voice of the team and the business unit champion successfully demonstrated the mid-execution phase retrospective output applied to the follow-on program that had just entered the planning phase was invaluable “early warning” learning. The business unit now had the means to quickly apply key program learnings to improve two programs in progress the execution improvements on the current program, and planning improvements on the follow-on program.

If the senior manager continued to stay with the old methodology of a single post-mortem, the planning phase learnings would continue to come in too late to affect the next program. At best, the changes may improve the third program on the roadmap – too late for effective change. As a result, the senior manager now requires all programs within his business unit to utilize the retrospectives methodology as part of the standard program management practices multiple times during the life of a project or program.

A Case Study

Over the past three years, we have collected a mound of data from our early adopters. To illustrate the power of collaborative change, meet Team Dolphin (not the real team name. The real name has been removed to ensure confidentiality of the product being developed). The Dolphin program team is responsible for effectively and systematically defining the platform planning deliverables such as platform requirements, value proposition, and scope for the product. The platform planner requested a four-hour retrospective prior to the hand-off to the execution team to:

- Improve platform planning efficiency through repeatability
- Align with platform planning team(s) during the key scope, feasibility and commit milestones
- Ensure all planning documents are achieving a specific work state to drive effective technical analysis

The sponsor decided to hold a six-hour retrospective (broken into two three-hour meetings) so both the planning and the execution teams could participate. In addition, the current platform planner invited the *NEXT* platform planner to sit in and listen to the discussions during the retrospective. For example, the Dolphin program team is just exiting the planning phase and is preparing to hand off to the execution team. The Dolphin planner invites the Penguin planner, who is next generation platform, to attend the retrospective so the Penguin team will have an early warning to issues and concerns. One of the most powerful aspects of their participation is for them to hear the emotion and passion from the Dolphin team when they share significant problems that plagued their planning efforts. Since the Penguin planner was just entering their planning phase, the learnings are immediately applied and impact to the program is instantaneous.

We are using a model at Intel where we locate a "recommender" and a "receiver" who commits to making changes on the next program. In the example above, the Dolphin team is the recommender and the Penguin team is the receiver. During the Dolphin platform retrospective, the team prioritized five areas for improvement. All of the five learnings (that came out of the current planning efforts) were shared (they are the recommender) with the subsequent planner (who has "received" the learnings) and has committed to implement the recommendations by saying "I will stop the repetitive mistakes and implement change."

What did the "receiving" team decide to do differently next time? The top three items are:

1. Ensure a documented plan is achievable. The next planning team has committed to creating and communicating a timeline that includes a much narrower scope with a better understanding of constraints.
2. Add a new position and staff it within the planning effort to focus on identifying opportunities, gaps, and constraints. The next planning team has been involved in the hiring of this key position to ensure the planners have one picture of the program.
3. Improve the use of tools to document and manage requirements. The next planning team is piloting an enterprise-wide Product Lifecycle Management (PLM) tool to integrate platform data and connect teams. The team is

committed to producing better requirements and managing changes as the project progresses.

So what happened?

It is too early to tell. What we do know is that a third program is just wrapping up their launch for the current product and two previous teams have captured learnings they are willing to share. Imagine five generations are sharing and comparing learnings to improve all aspects of the program. We believe collaborative discussions, focused on multiple generation learnings provide a more comprehensive learning opportunity as comparisons can uncover gaps which initiate conversations with experienced planners.

In conclusion, we have captured over 30 learnings within this business group, spanning five generations. All key learnings are captured in a common repository for planners, architects, and engineers to query, browse, and search for learnings, which allow various solutions and experts to talk with about specific problems they are trying to solve. The planner wants to document key practices in the repository so other divisions can benefit from their wisdom. This is a way for all planners to give back to other teams by sharing best practices and innovative solutions. For the repository to be useful, the planners tell us they want a robust search engine that returns logical and ranked learnings so they get to what they are really interested in, they want a web-based solution, and prioritized learnings with data from a trusted source.

Keys to managing collaborative change

Once the value proposition for broad deployment is established and influential sponsors are behind you, the real work begins to ensure retrospective methodology achieves full adoption. The five primary aspects for managing the transitional change management activities are:

1. Determine if your company is a "Learning Organization." The first step is to build the case for organizational learning as a means for continuous improvement, and retrospectives as the method for collaboratively capturing and applying the improvements is to do an environmental scan to find out what is current state. Ask questions such as:

- What is your process to gather lessons learned?
- How often do you require teams to stop and reflect?
- Do you document the lessons learned in a consistent format and in one location?
- Can you effectively search, monitor, and update the documented learnings?

Look for a consistent process to capture lessons learned, multiple touch points during the lifecycle, and a centralized location so the entire organization can query, search, and monitor the learnings. You are well on your way to being a learning organization if these three items are in place.

2. Tailor the methodology. Every organization is unique; therefore, the retrospective methodology must be flexible and scalable to meet the full needs of the team. Start by asking simple questions to understand the current program review process, and then negotiate how much change the team can absorb. For example, instead of starting with performing three retrospectives at strategic points across the program lifecycle, negotiate for one additional retrospective (in addition to the end) as a place to start.

3. Ensure someone is accountable for results. Action plans collectively created in a retrospective, have committed owners, and established due dates, but it becomes easy for the owner to procrastinate on the completion of an action plan when it is in addition to their normal program-related work. Below are three suggestions to prevent apathy from taking over and increasing the accountability of action plan owners.

- Select one opportunity the team whole-heartedly agrees must improve. During the brainstorming of issues, it is easy to get over zealous and want to "solve world peace." Resist the temptation!
- Document the one opportunity in the form of an action plan (working as a team) to fully understand the problem you are trying to solve. It is human nature to want to jump to solution space and fail to clearly state the problem in the form of a well-written problem statement. A good problem statement reads: Lack of X, resulted in Y. Brainstorm together various obstacles to making the improvement, identify who needs to support the change, and the next steps in short, two-week intervals.
- Ensure the completed action plan is accessible to the team. Many times the output from a retrospective is stored on someone's computer hard drive. If possible, create a centralized repository for all learnings so those who need to get to the data can do so quickly and easily. In addition, capturing and documenting the team member names associated with the recommended action improves accountability of the action.

4. Make changes to existing work systems. Change is easier for humans to embrace when it is embedded into well-established work systems. Companies who use a development "lifecycle" or a "software development life cycle" where a program team passes through various stages such as exploration, planning, development, and product launch can link lessons to the entry and exit criteria of the phases and milestones to help reinforce change.

5. Change reward systems: Establish a reward system to recognize and reward teams that make use of the retrospectives methodology and implement the necessary behavioral changes. We have found an effective reward is something desirable, timely, and provides an opportunity for reinforcement of the change. Small rewards, delivered for meeting smaller goals, are many times more effective than bigger rewards. These can be as simple as a hand written thank-you, gift certificate from a good restaurant, or movie passes. Bigger recognition efforts such as monetary awards such as $500 or $1000, extra vacation days off, or annual "extreme performer" incentive award trip to some exotic location. Whatever the reward, invest time and energy to create a reward system that reinforces the behavior changes you desire.

Conclusion

As with any change in the way people perform their work, one of the most difficult challenges to overcome is getting people en mass to modify the way they behave in order for the change to become part of the fabric of the organization. Change is hard; however, working collaboratively to identify what needs to change and then working together to help make the change(s) stick will be easier utilizing the retrospectives methodology.

The first challenge is to define and communicate the value proposition for broad adoption of the methodology and cross the chasm between the early adopters and the mass majority of the organization. Once this is accomplished, the five key steps in making collaborative change are:

1. Determine the starting point based upon an organization's current learning capability
2. Tailor the methodology to ensure problems are collaboratively identified by the team
3. Ensure someone is accountable for implementing the documented action plans
4. Make necessary changes to existing work systems
5. Change the reward systems to reinforce positive behavior

Working together, to collaboratively solve problems is the most effective way for significant change to happen. Going it alone is more difficult and isn't as much fun.

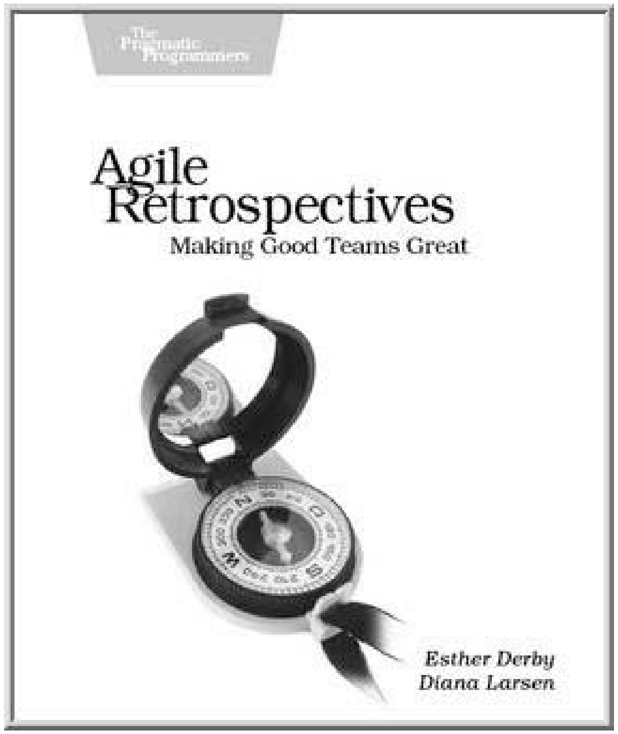
References

1. Moore, Geoffrey A. Crossing the Chasm. New York, NY: HarperCollins Publishing, 1991.

Agile Retrospectives:

Collaboration for Continuous Improvement

Diana Larsen
dlarsen@futureworksconsulting.com

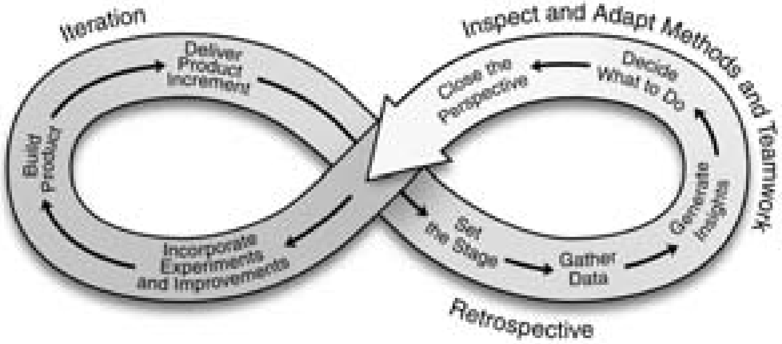


The Pragmatic Programmers

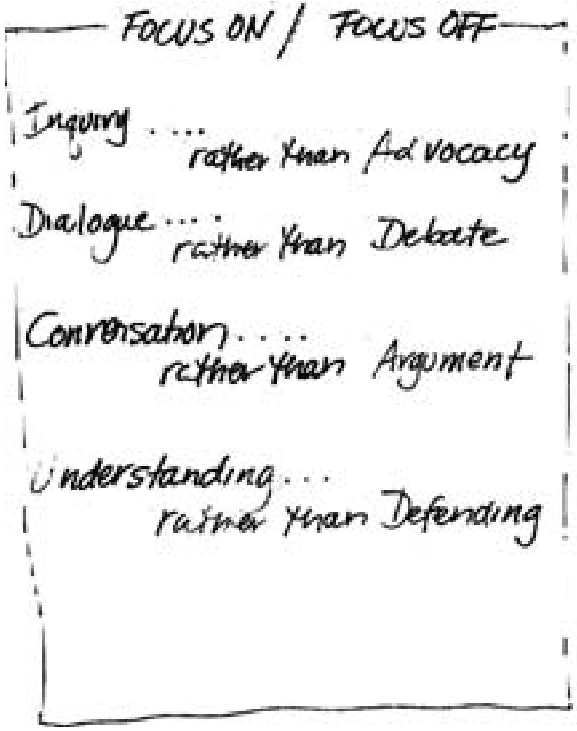
Agile Retrospectives
Making Good Teams Great

Esther Derby
Diana Larsen

Learning, Thinking, & Deciding Together



Set the Stage



FOCUS ON / FOCUS OFF

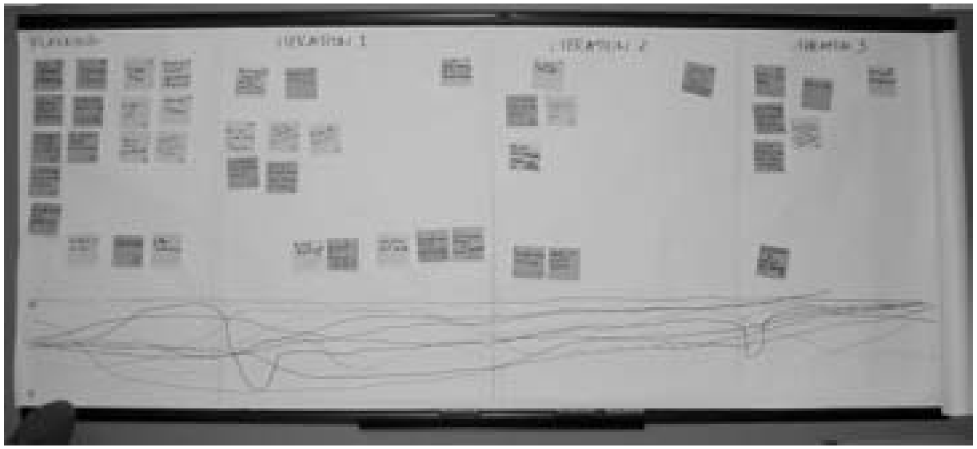
Inquiry . . . .
rather than Advocacy

Dialogue . . . .
rather than Debate

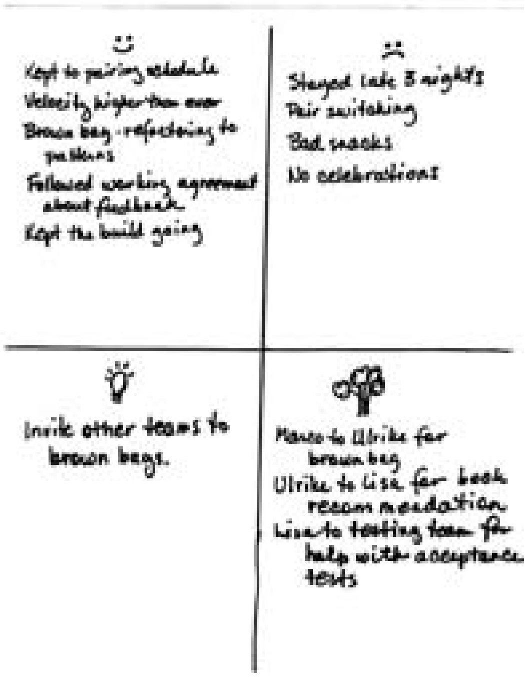
Conversation . . . .
rather than Argument

Understanding . . .
rather than Defending

Gather Data

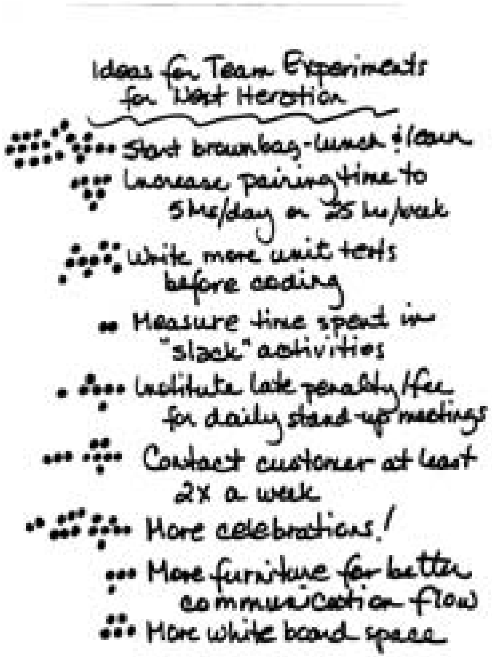


Generate Insights



☺	☹
Kept to pairing schedule Velocity higher than ever Brown bag - refactoring to patterns Followed working agreement about feedback Kept the build going	Stayed late 3 nights Pair switching Bad snacks No celebrations
💡 Invite other teams to brown bags.	💐 Mano to Ulrike for brown bag Ulrike to Lisa for book recommendation Lisa to testing team for help with acceptance tests

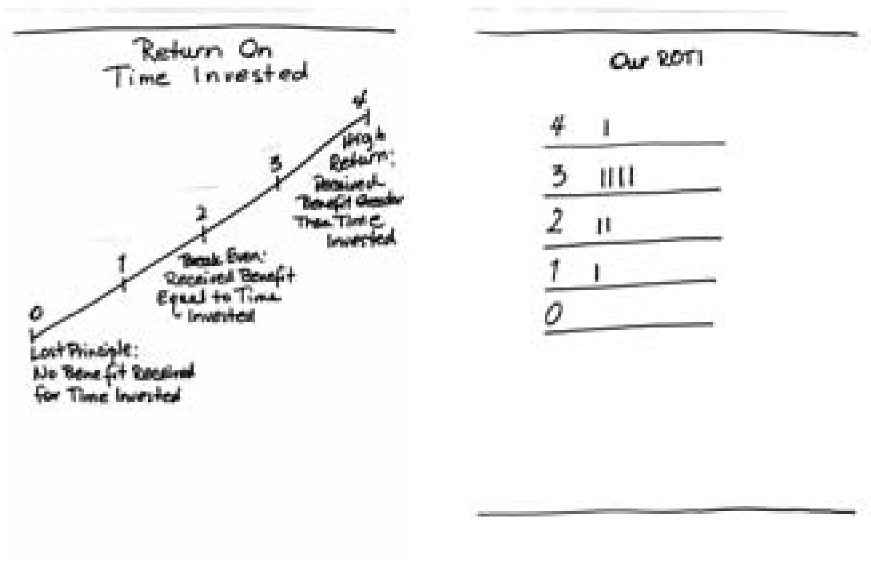
Decide What to Do



Ideas for Team Experiments for Next Iteration

- Start brownbag-lunch & learn
- Increase pairing time to 5 hrs/day or 25 hrs/week
- Write more unit tests before coding
- Measure time spent in "slack" activities
- Institute late penalty/fee for daily stand-up meetings
- Contact customer at least 2X a week
- More celebrations!
- More furniture for better communication flow
- More white board space

Close the Retrospective



Where does it get us?

- Improved Productivity
- Improved Capability
- Improved Quality
- Increased Capacity
- Higher Trust & Morale

Preparing

Choices
  Goal, Duration, Location

Invitation

Design Skills

Facilitation Skills

Support Change

Resources

retrospectives-subscribe@yahoogroups.com

Derby & Larsen. *Agile Retrospectives: Making Good Teams Great.*

Kerth. *Project Retrospectives: A Handbook for Team Reviews.*

Kaner, et al. *Facilitator's Guide to Participatory Decision-Making.*

Stanfield, ed. *The Art of Focused Conversation.*

http://www.estherderby.com

http://www.estherderby.com/weblog/blogger.html

http://www.futureworksconsulting.com/blog

http://www.cutter.com/offers/larsen.html

Biography

Diana Larsen consults with leaders and teams to improve project performance, support innovation, and establish satisfying, results-oriented workplaces.

With more than fifteen years of experience working with technical professionals, Diana brings focus to the human side of software development. Her clients value her collaboration in building their capability to interact, self-organize, and shape an environment for productive teams.

Current chair of the Agile Alliance board, Diana co-authored *Agile Retrospectives: Making Good Teams Great.*

She writes an occasional blog post at "Partnerships & Possibilities" http://www.futureworksconsulting.com/blog/ .

Find more information about FutureWorks Consulting, Diana Larsen, and additional resources at the website, http://www.futureworksconsulting.com .

Collaboration between Theorists – *Analyze This!*

Discussions leading toward better cost/benefit estimations and project management

Abstract

Between three different sessions at the Agile Open Northwest 2008, the author and another software professional developed a cost/benefit charting scheme that:
1) enables development teams and customers to prioritize features,
2) identifies features that should be quashed, pursued, or further decomposed, and
3) allows stakeholders to monitor and manage priorities over the course of development.

This collaboration amounted to about 15-20 minutes total and produced a diagramming technique that can be used in bleeding-edge agile shops and in shops using more traditional methods. The chart is a basic Cartesian plane with **Cost (Points)** on the x-axis, **Value** on the y-axis, and something other than dots at the intersections. We call this technique *Analyze This* as it focuses analysis where it is needed.

My collaborator is a noted Agilist with direct control of his team's practices: I am a software quality practitioner with influence in a more traditional plan-driven shop.

With differing needs and mutual respect, these two collaborators developed a useful tool in far less than an hour by listening, expanding, and deferring to one another's areas of expertise.

Keywords: cost/benefit, estimate-ranges, estimate-boxes, analysis focus

Bio

Ian Savage

A quality evangelist, Ian is a veteran software developer, quality assurance engineer, and manager with experience in the manufacturing, financial services, construction estimating, and security domains. For more than 30 years he has worked to improve productivity and software quality through rigorous development methods and processes and now through the pragmatic application of Agile methods.

Ian serves on the Software Association of Oregon's Program Committee. He authored the SAO Quality Assurance Special Interest Group charter and serves on the SAO QASIQ Steering Committee. He has contributed to American Society for Quality's certification program for software quality engineers and the Software Engineering Institute's Software Engineering Process Group conference. He is a member, and supporter, of the Agile Alliance.

Ian attended the very first Pacific Northwest Software Quality Conference in 1983. Since then he has served on PNSQC's board as President, Vice President, and Secretary. He has also chaired the PNSQC Software Excellence, the Strategic Planning, and the Program Committees. Ian is currently serving as the PNSQC Program and Conference Chairman.

His current interests include a) the soft side of quality assurance; b) coaching semi-agile, high performing teams; c) requirements engineering; and d) exploring the wonders of Central Oregon.

Collaboration between Theorists – *Analyze This!*

Discussions leading toward better cost/benefit estimations and project management

Overview

In the course of three offline discussions and one ad-hoc presentation, we may have invented something useful: a cost/benefit diagramming method that has applicability in agile shops and in project-oriented shops.

Built on value engineering and the axiom that estimates get better as time passes, we postulate that using a simple cost vs. benefit graph leads to more profitable software releases and better use of available resources.

The concluding section contains a twelve-step program that can improve product content decision-making and communication between product owners and engineering. Lastly, suggestions for further work are included.

Our story begins…

Agile Open Northwest is a fantastic conference. Recognized Agile leaders, many practitioners, and many others moving toward agile methods organize into small groups to talk about topics suggested by any of the participants.

At the 2008 version in Seattle (http://www.agileopennorthwest.com/index.php), Arlo Belshee[1] led one such session – "Kanban-based Planning." This session ended with a discussion costs, benefits, and estimates. Several participants gave their ideas about value. Arlo drew a standard cost/benefit graph (Figure 1) on the flipchart and something clicked in my head.

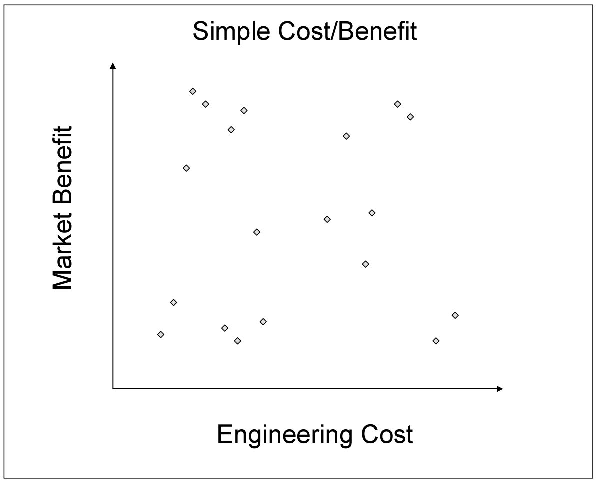


Figure 1: A standard cost/benefit chart where each dot represents a feature

The cost/benefit discussion was a tangent to Arlo's Kanban session, which was almost over, so I decided to discuss my epiphany with him after his session.

But before we go there, let's modernize the basic chart. As evidenced in the Kanban discussion, an emerging trend in the Agile movement is to use "**points**"[2] as a measure of cost and "**value**" as the customer expectation. So an Agile cost/benefit chart might look like Figure 2. The "Product Owner," of course, could be either a product manager or an on-site customer.

[1] Arlo is a former PNSQC director and author of "*Promiscuous Pairing and Beginner's Mind: Embrace Inexperience*" [Belshee2005].

[2] See Agile Points calculator: http://cgi.nordija.com/AgileMetrics.cgi and Kieth2006 http://www.agilegamedevelopment.com/2006/03/hours-vs-story-points.html]

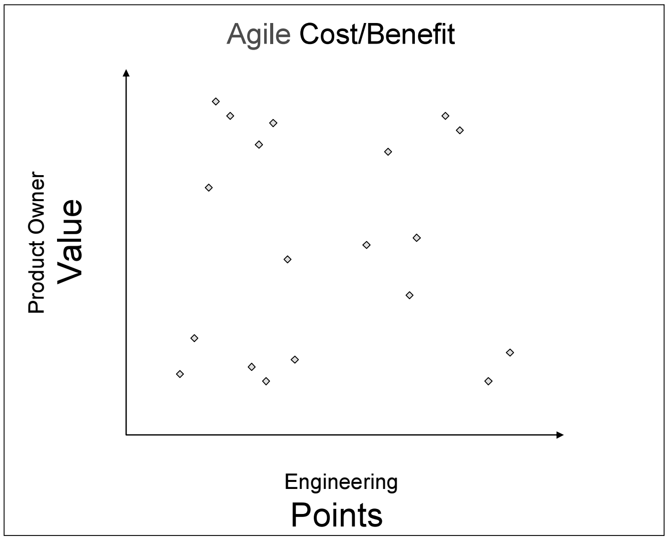


Figure 2: A standard cost/benefit chart using Agile terms

Returning now to AONW 2008… our collaboration started…

“Arlo, it occurs to me that the points on this cost/benefit chart are [expletive deleted]. They imply too much precision.”

The small dots on a typical cost/benefit chart are bogus. They imply that we can *a priori* determine the value of a feature and the impact that its development will have on the producing organization.

In real estate, the true value of a property is only known when it changes hands. Likewise, even the most astute product owners can only give us educated guesses about to potential income a specific feature will generate. In the software world, Steve McConnell has educated us about the foolhardiness of point estimates for developing features [McConnell1996].

What we really need are ***estimate-ranges*** for both engineering points and product values for each feature thereby creating ***estimate-boxes***. When we allow cost and the benefit to float, we have a more realistic picture of each feature’s impact on scope and resources.

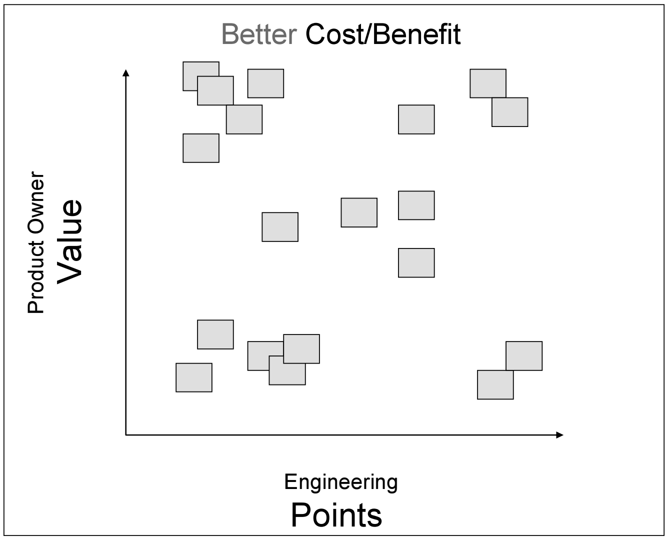


Figure 3: Points and Values expressed as ranges

Still, there's something artificial about these estimate-boxes. The standard deviation is zero – they are all the same size. The likelihood of each feature having exactly the same projected value and estimated cost is nil. Tim Lister advised us [Lister1993] that the acid test for quality is whether anyone is doing serious assurance of the project schedule. An audit of a cost/benefit chart with these identical estimate-boxes would correctly find that the estimates were contrived.

A more realistic view is represented in Figure 4. This reflects that fact that not features are all created equally and they have different value[3] in the marketplace.

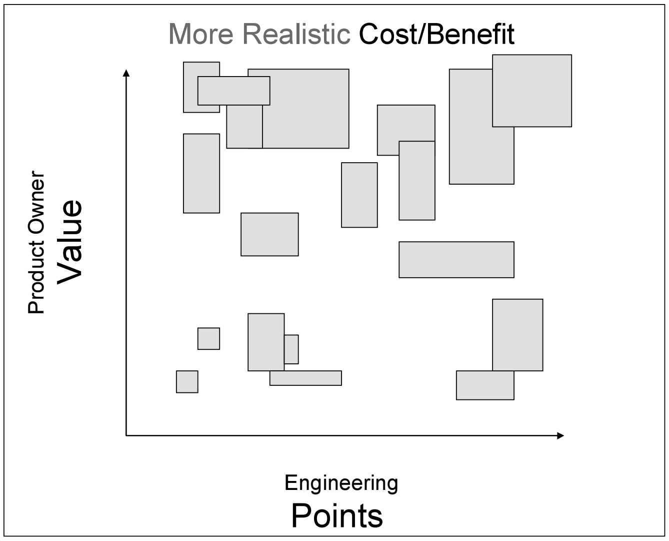


Figure 4: Estimate-Boxes of Differing Sizes

Again returning to AONW 2008…

At the next break, I approached Arlo again. "I'm thinking that the rectangles would change in size and location over the course of a project."

Going in a different direction, Arlo said "The scales are different. The points scale is linear and the value scale is logarithmic."

Fumbling, my response was something like "Huh? Why in the world…?" but then I remembered that Arlo is a trained mathematician. One thing we have learned at PNSQC is that experts often know the answers without thinking through all the steps. They can simply leap to correct conclusions. Three examples:

- Ace fighter pilots turn OFF audible warnings so they can focus on the dogfight.[4]
- Chess grandmasters (and presumably chief software architects) find solutions through pattern recognition. [Gobet2003]
- World-class athletes, in their best performances, rely on instincts. [Will1991]

I didn't need to challenge Arlo. I instantly surmised that he was probably right. His assertion passes the sanity test: Some features are worth millions, some are worth very little, and those in the middle are orders of magnitude more valuable than those at the bottom. So our emerging cost/benefit chart now has scales (Figure 5).

[3] Note: For simplicity, consideration of the time value of money is intentionally omitted from this paper.

[4] This does not appear in the literature for obvious reasons. It was reported in a PNSQC Keynote address.

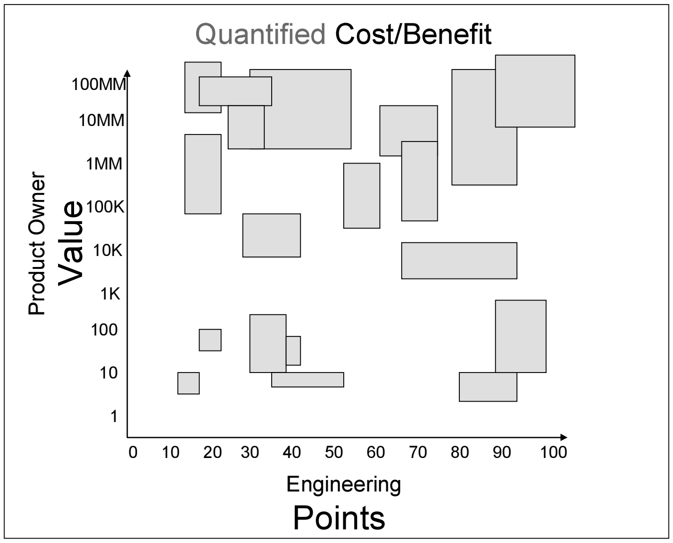


Figure 5: Points are linear but Value is *logarithmic*.

But he wasn't done. Showing his considerable value to this collaboration, Arlo then drew on his experience as a consultant.

"Now, the first thing a consultant will do is divide it into quadrants."

This simple addition was a second breakthrough for our collaboration. These quadrants provide a way to categorize the project features and they indicate different actions. Our heretofore academic collaboration suddenly became a powerful project management tool. The four categories in Figure 6 have different payoff matrices and require different development strategies.

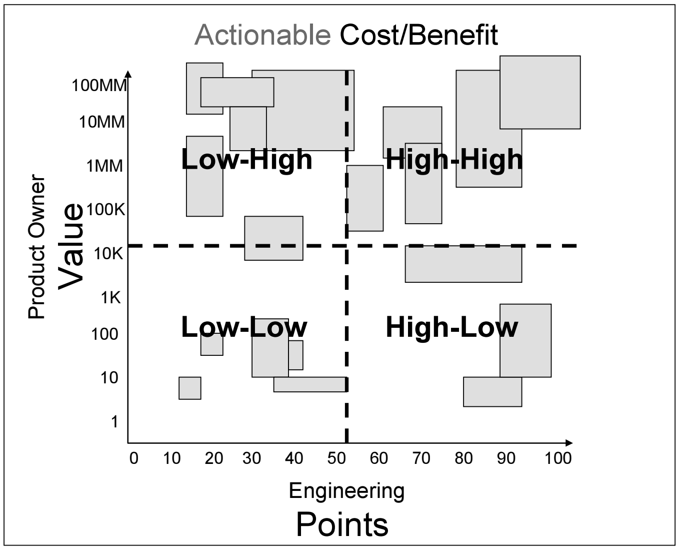


Figure 6: Actionable quadrants

As objectionable as the term "no-brainer" may be, some decisions are simply obvious. As Figure 7 shows, some decisions are the easy. Should we build those items in the low-cost/high-benefit quadrant? Yes, of course. How about those in high-cost/low-benefit quadrant? Obviously not.

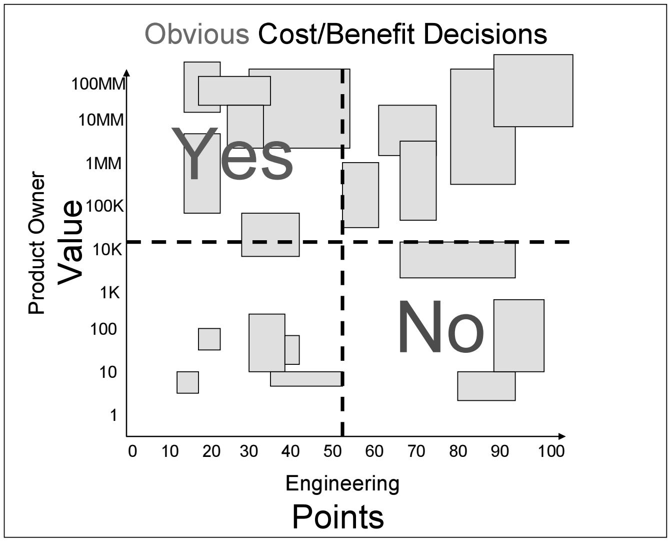


Figure 7: The two easy decision quadrants

This is analogous to risk-based testing – concentrating on those areas where problems exist and not the rock solid areas. Since high-quality software needs less testing than low-quality software, a prudent testing organization focuses its efforts on those areas that are known to contain errors. Taken to its extreme, a well-constructed, broad and shallow test could qualify a release for shipment if the tests constituted a statistically representative sampling and all other release criteria were met. See PNSQC Proceedings 2002 for papers on release criteria.

That leaves two quadrants. How should we treat those? Figure 8 shows the quandary quadrants.

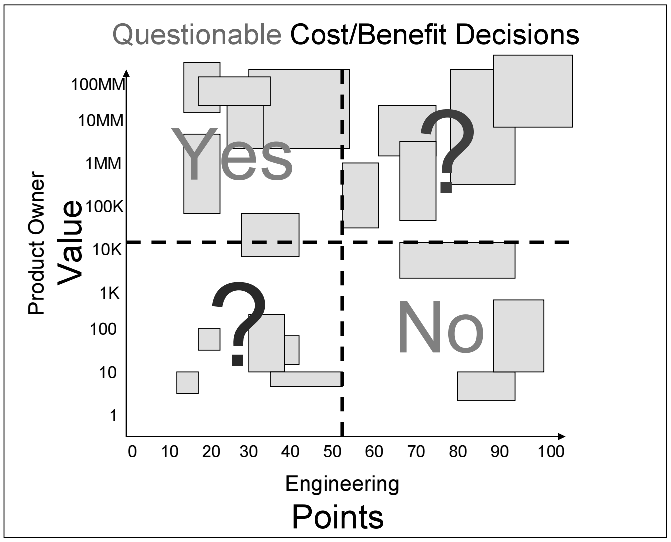


Figure 8: The Hard Part – What to do with these quadrants?

This is analogous to the two types of risks in sampling theory: alpha and beta. If a producer tests too lightly, she is subjecting herself and the consumer to alpha risk (this is also known as "consumer's risk"). If she tests too heavily, she is subjecting herself and the company to beta risk (this is also known as "producer's risk"). [Johnson1974]

Back to the collaboration again…

"Don't do the low-low things." Eh? Why so? Aren't these low-hanging fruit?

Arlo had me reeling now. But he explained that Yourdon wrote about these seemingly small things. They end up being not-so-small. They are the stuff of death marches. [Yourdon2006]

While a little disconcerting at first, this too makes sense. Firstly, since Parkinson's Law [Parkinson1958] holds that work expands to fill the available time, it's entirely feasible that these "small" things end up being medium things and the cost grows linearly – consuming the available points. Also, the industry practice is trailing the theory: One agile tenet is to *do the easiest thing that will possibly work* [Cunningham2003]. However, coders are still anticipating future uses.

Got it: Small features grow, they never shrink. If you do enough low-value things, you won't have time for the high-value things.

I resolved to read the Yourdon book.

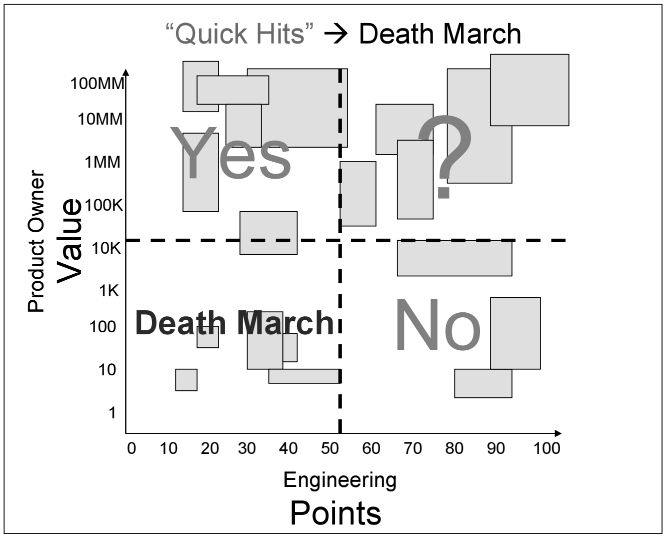


Figure 9: The myth of "quick hits"

We both jumped to the one remaining quadrant: Arlo said "Decompose" while I said "Analyze." Or was it the other way around? And this is where the payoff comes.

In large part, our industry has shunned doing comprehensive systems analysis. We have excellent methods and modeling tools. We don't use them. Yes, of course, there are many exceptions but let's face it – managing requirements is the software industry's biggest improvement area.

Yourdon [2006] suggests that our industry move to a triage model to deal with the preponderance of projects that become death marches. Applying structured analysis to features in the high-cost/high-value quadrant is one way of focusing resources where the potential return is highest. This will avoid "analysis paralysis."

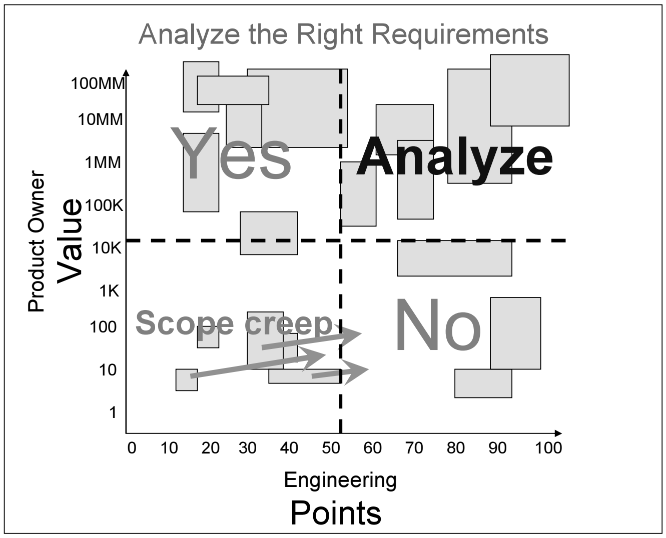


Figure 10: Focus requirements analysis in the high-high quadrant

Functional decomposition is a heavy-weight process if applied to the entire problem space. So do it if you must (for regulatory or safety reasons, but don't do it to maximize ROI. Rather, analyze those things that have high-value components. Parsing the high-high features into their constituent components allows you to pull some components into the low-cost /high-value quadrant (the YES quadrant). The other components, the fallout, add little or no value. They fall into the high-cost/low-value quadrant (the NO quadrant).

Value engineering (c. 1942) is another way to analyze these features. Its purpose is to find the *primary function* and *alternative ways of performing that function*. The Allies used VE extensively during World War II to conserve resources and to provide support to win the war. From Wikipedia:

> **Value engineering** is a systematic method to improve the "value" of goods and services by using an examination of function. Value, as defined, is the ratio of function to cost. Value can therefore be increased by either improving the function or reducing the cost. It is a primary tenet of value engineering that basic functions be preserved and not be reduced as a consequence of pursuing value improvements. [1].

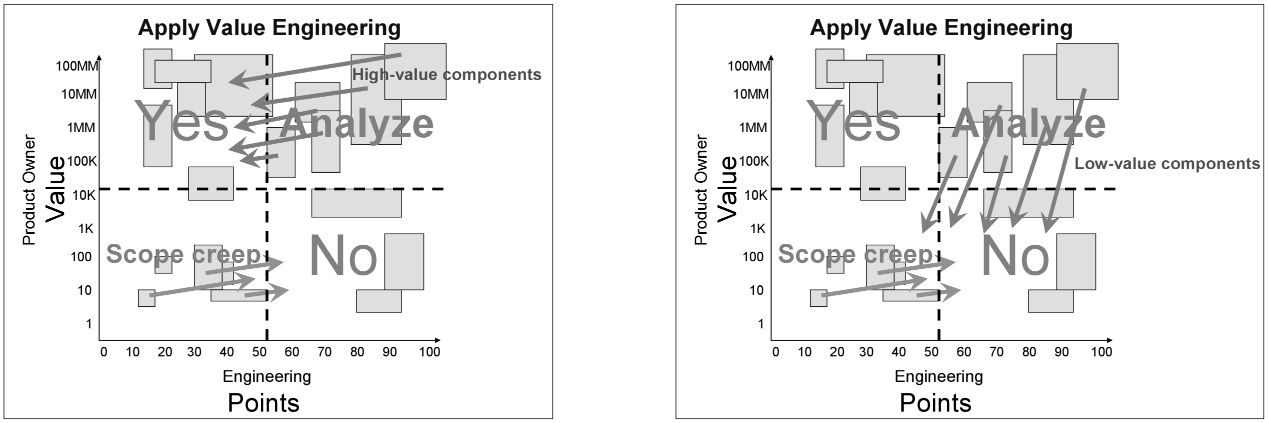


Figure 11: Value Engineering separates essential and non-essential functions

The ideal *Analyze This* chart is shown in Figure 12.

In this vision of Nirvana, we have:

- filled the low-cost/high-value quadrant.
- removed low-value components from the work queue,
- reduced the estimate-boxes sizes (increased the precision) through the deeper analysis, and
- an empty high-high quadrant - room for other high-value ideas as they arise.

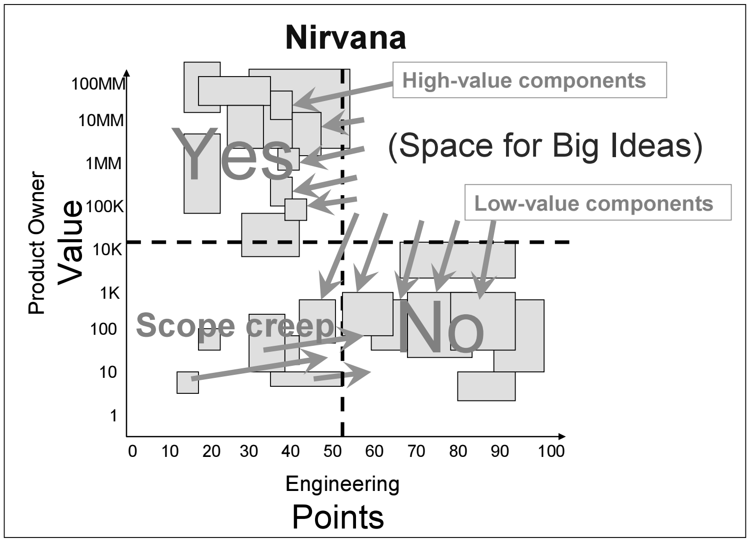


Figure 12: The ideal *Analyze This* chart

When to Use the *Analyze This* Technique

Who exactly should this *Analyze This* technique and when? Since circumstances, contexts, and cultures differ from shop to shop, the ownership and frequency decisions are left to you. But some suggestions follow.

In a typical plan-driven shop, at *project initiation* the product managers would set the vertical sizes and positions and the development managers set control the horizontal. Then make adjustments before each successive iteration and *whenever new information arrives*. It's entirely reasonable to ask a developer to update her task estimate once she has burned 50% of her initial estimate. Likewise, it's entirely reasonable to ask the product manager to update her sales projections once the competitor releases a similar product or a major sales opportunity opens up. With adequate tool support (see Next Steps), the cost of updating estimates is would be small.

During the course of every plan-driven project, requirements change. The *Analyze This* technique handles this nicely. The new requirement slide directly into the existing chart![5]

In an agile shop, the team unit can adjust both dimensions during each planning game and at daily stand-up meetings.

[5] For instance, my development manager asked this… but we still have only so many people and only so much time-to-market. How do we prioritize what we do next?
My response… Too bad you've headed off for [vacation]. I'd like to get clarification on your question. Always the one with opinions, I'll press on…

If you mean "***how do we prioritize among the resultant buckets***":
Well, the buckets higher in value and lower cost certainly seem good candidates.
In case of ties, I would go with the ones with lower spread – i.e. those about which we are more certain.
Else if you mean "***how do we prioritize within the buckets***":
I hadn't considered that issue. I had assumed that any features selected for implementation would need *all* their constituent components done. One could imagine applying Value Engineering to those buckets also.
Else if you mean "***how do we prioritize features as they are introduced during the project (e.g. via CCRs)***":
I hadn't considered that either. You ask the darnedest questions! Could we just slide them into the Value/Points matrix and see where they land?
Else:
Sorry, I don't understand the question.
Endif

Conclusion:

Perhaps there is a **twelve-step program** to high-value use of engineering/analysis resources:

1. Open the product backlog.
2. Nominate candidate features.
3. Estimate their engineering cost (Dev & QA managers).
4. Estimate their product value (Product Owner).
5. Assign the high-value, low-cost features.
6. Cancel the low-value, high-cost features.
7. Ignore the low-value, low-cost features.
8. Analyze the high-value, high-cost features.
9. Repeat steps 5-7 for these newly-analyzed features.
10. Revise estimates as you learn more about the market and costs.
11. Repeat steps 5-10 until you meet your project completion criteria.
12. Conduct retrospective and celebrate.

As Barry Boehm notes in his excellent book [Boehm2003], some conditions merit more “discipline” and that includes process improvements. One effect is that change takes longer in plan-driven shops than in agile shops. Ian is working with his product manager, development manager, and other stakeholders toward piloting *Analyze This* at McAfee where we are using elements of plan-driven and agile methods.

Suggestions for further study:

- Case studies of applying this *Analyze This* technique including lessons learned. Especially experiences with adjusting the value and cost estimates from iteration to iteration.
- A tool to make these charts visible for all stakeholders. A visual technique like *Analyze This* needs tool support. We may have such tool support to demo at the conference.
  - Any tool that supports *Analyze This* technique should display the total values and costs.
  - It should support “window, icon, menu, pointer” metaphors for resizing estimate-boxes.
- How to reduce the variability of initial feature estimates.
- Trending reports for cost and value estimates.
- Can we really assume that experts are right? What’s the risk/payoff picture?

References:

Belshee, Arlo: “Promiscuous Pairing and Beginner's Mind: Embrace Inexperience”: http://portal.acm.org/citation.cfm?id=1121987.1122100&coll=&dl=acm&CFTOKEN=6184618, 2005

Boehm, Barry and Richard Turner: Balancing Agility and Discipline – A Guide for the Perplexed, Addison Wesley, 2003.

Cunningham, Ward: http://en.wikiquote.org/wiki/Ward_Cunningham, 2003

Gobet, Fernand: http://people.brunel.ac.uk/~hsstffg/papers/Chabris_Hearst_www.doc, 2003

Johnson, Robert: Elementary Statistics, Duxbury Press, 1974

Lister, Tim: “The Acid Test for Quality” PNSQC Proceedings Keynote Address, 1993

McConnell, Steve: Rapid Development – Taming the Wild Software Schedules, Microsoft Press, 1996.

Parkinson, Cyril: Parkinson's Law: The Pursuit of Progress, London, John Murray, 1958.

Will, George: Men at Work: The Craft of Baseball, HarperCollins Publishers, 1991

Yourdon, Ed: Death March, Second Edition, Yourdon Press, 2006.

IT Collaboration at Stanford University
Beyond Quality

Claudia Dencker
Administrative Systems, Stanford University
Stanford, CA 94305
650-736-0599

Abstract

In February 2007, the Quality Assurance organization in Administrative Systems (AS), Stanford University was formally underway with two mandates in mind: to set up an independent testing function and to improve the quality of AS delivered software solutions. The challenges that these two mandates attempted to address are as follows:

- Inconsistent (and in some cases, non-existent) processes, tools and procedures across all practice areas within AS
- Lack of transparency of project activities within and outside of AS
- Bad software quality upon delivery

Today significant progress has been made on all three challenges. Silo'd practices are slowly merging into a cohesive whole, projects are more open and transparent with business partners actively participating in project progress, and software quality is improving in key areas.

This paper reports on the progress that the QA organization has achieved since its formation and highlights the unique role that collaboration tools and processes have played in improving business partner satisfaction in AS' delivered solutions at Stanford University. One project will be profiled, the PeopleSoft Student Administration and Human Resources version 9 upgrade project that affects all business offices in support of the university's mission of teaching, learning and research. Tools used are a wiki integrated with an issue tracker, test case management, project dashboards and more.

The more open mode of working with staff across different job disciplines, in different physical locations and in a few cases, across multiple time zones has led to a much higher degree of satisfaction for all team members. Additionally, the university has benefited where individuals are accountable, projects are more disciplined and structured and transparent through the use of an open collaboration set of tools and processes.

Note: Products listed by name in this paper do not imply endorsement. Rather they are listed to provide context for the tools being used by Stanford University, AS Department.

Bios

Ms. Dencker is Director of QA, Administrative Systems, Stanford University. She was formerly Program Coordinator of the newly consolidated program, SEQ (Software Engineering and Quality) at University of California, Santa Cruz Extension and President of Software SETT Corporation, a company that provided QA and software testing solutions to software IT professionals worldwide. She has trained professionals worldwide in software testing and test management as well as led and managed teams on many projects in Silicon Valley. Ms. Dencker is a Certified Software Quality Engineer (CSQE) and graduated from San Jose State University with honors.

Introduction

Bringing a Software Quality organization into a university unfamiliar with such a function has been an enormous challenge. But as most universities today increasingly rely on software solutions to run their "business" of learning, teaching and research, traditional software development practices are folded in. Stanford University is no different. Many of the business functions of student admissions, student records, financial aid, payroll, and more are solely driven not by punch cards and paper documents, but rather by highly sophisticated, web-enabled, self-service software solutions. All the off-the-shelf and custom-built solutions must be specified, developed, tested and supported following solid industry practices.

In 2006, the Administrative Systems (AS) group was reorganized into an independent function, apart from its sister organization, Information Technology Services (ITS). This new organization maintains and supports the following applications:

- PeopleSoft Campus Solutions® (Student Admissions, Human Resources), STARS (online training), time tracking and more.
- Oracle Financials® including various custom bolt-ons such as iOU, CMS (Commitment Management System – a forecasting tool) and more.
- Reporting and EDW (Enterprise Data Warehouse)
- Middleware and Integration Services

As a part of this reorganization a need was identified to establish an independent QA group whose mandate was to:

- Set up an independent testing function
- Improve the quality of AS delivered software solutions

Quality of solution delivery had been a troubling problem in past software deployments, and hence, led to the establishment of a group dedicated to solving this issue. But, as most readers will recognize, software quality is not owned by just one department, but rather by all.

As with any endeavor in which ramp-up and results need to be achieved quickly, looking for solutions that span multiple areas is always a big win. This was accomplished through the use of collaborative tooling.

Collaborative Tooling

One of the goals of the new AS, not just QA, was to bring a greater level of transparency to its efforts. This can be a challenge as many university business offices and/or staff don't understand the inherent dynamic nature and inherent susceptibility of software to instability, unreliability and poor performance. Opening up a process that many don't understand invited criticism, anxiety and intrusive micro-management that added a burden to the AS project teams. But, as in all technology driven endeavors, everyone had to learn and come up to speed as to what can and cannot be done.

The ramp-up for greater transparency was facilitated through the introduction and use of collaborative tools. In support of the QA mandate of improving quality, collaborative web-based tooling became an instrumental solution that spanned a number of QA roadmap goals. The collaborative tools that are discussed in the following sections are as follows:

- Issue Tracking, Jira®
- Wiki Collaboration, Confluence®
- Test Case Management and Tracking, Quality Center®

Issue Tracking

The first tool to be introduced was Jira, an issue tracker from Atlassian Software. During the initial phase of investigation as to bug trackers used throughout AS, practice areas were not in sync. AS teams used spreadsheets, a custom form add-on to the call tracking system and Bugzilla. Jira offered a nice upgrade path from Bugzilla, an excellent step-up solution from spreadsheets, and a more configurable solution from the custom call tracking form. Furthermore, this product has strong ties to the open source community which meant a strong base for quality (100's of developers, testers, users testing it) and rapid pace of enhancements (due to the pressure from a vocal open source community).

The issue tracker workflow was configured to map to optimum quality practices in support of the Find/Fix process. It was quickly rolled-out as a pilot solution for the Reporting practice which was still working off of spreadsheets. The pilot lasted about 6 months with a production Go Live when the Bugzilla data was migrated in.

To enhance use of the solution and to facilitate quick turnaround of issues during the course of a project, AS/QA opened the tool up to business users who participated closely with AS during the two major test phases:

- SIT (system integration test) which is an internal, independent AS test phase similar to the QA or test phase of commercial endeavors
- UAT (user acceptance test) where the Business Affairs central offices and their distributed users test to ensure fit to their needs.

Issue Tracking Benefits

Consolidating onto one web-enabled issue tracker had the following benefits:

- Consistency in how bugs are identified and tracked through the fix process
- Consistency in practice among a diverse group of users who report in to a wide range of organizations
- Transparency of pre-production issues which engages all participants, excludes no one

Wiki Collaboration

Another tool to be adopted was Confluence, a collaboration wiki from Atlassian Software. This tool had already been used by one of the practice areas in AS, the Middleware and Integration Services group as part of their agile practices. All their technical and test documentation, requirements analysis and design were conducted on the wiki. Confluence was embraced by the QA group to post group-specific information such as project updates, new team member info, group mission and mottos, team minutes, etc. The fact that this tool was from the same company that created the issue tracker was an added plus.

No workflow was necessary; we simply created workspaces and let the dynamics of the group determine what was posted and how it was to be used. The use of the wiki was rolled out to all of AS as of the Jira Go Live in December 2007. Some examples of workspaces are as follows:

- Project-based spaces whereby project teams posted information relevant to the project participants. **Note:** The use of a wiki on one of the largest projects in AS is described later in this paper.
- Group-based spaces such as the AS/QA space whereby information was posted relevant to the group members
- Personal pages in workspaces to maintain a log of events

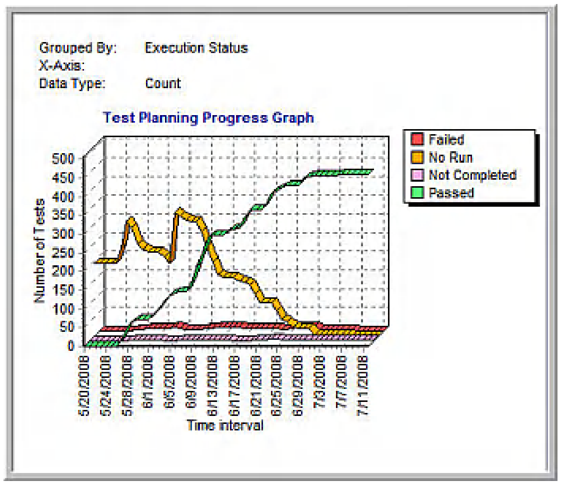
Wiki Collaboration Benefits

Consolidating onto one web-enabled wiki had the following benefits:

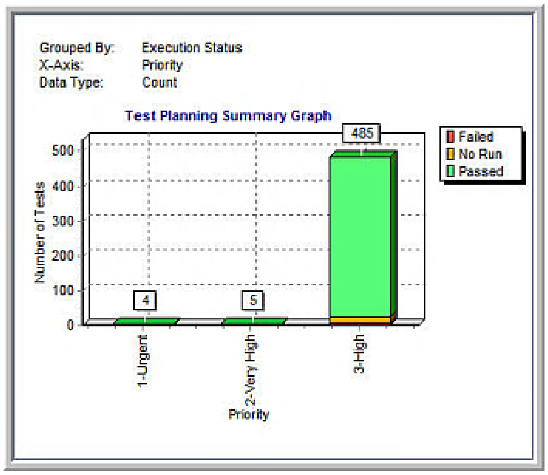
- Relevance of information (assuming information was maintained!)
- Enhanced ownership (team/group members had specific directives to update pages or sections)
- Consistent knowledge about a project or group
- Transparency of effort which was in sharp contract to the business as usual, opaque/black hole of the past

Test Case Management and Tracking

The third tool was Quality Center from HP for test case creation, management and tracking of test results. This tool isn't typically thought of as a collaboration tool as it was primarily used as a tracking tool from which to draw test case execution metrics. Nonetheless, information was posted in the wiki on a regular basis and as such was folded into the broader collaborative initiative started with Jira and Confluence. Two projects used this tool to upload their test cases, maintain the cases, and to log their test results. The reporting and query analysis is outstanding and hence made for a great addition to the project information that was posted to the wiki.



Progress graph showing planned tests that have passed, failed, not completed, not run.



Planned tests sorted by priority (urgent, very high and high priority) and their status of passed, failed, etc. Medium and low tests not shown.

Test Case Management and Tracking Benefits

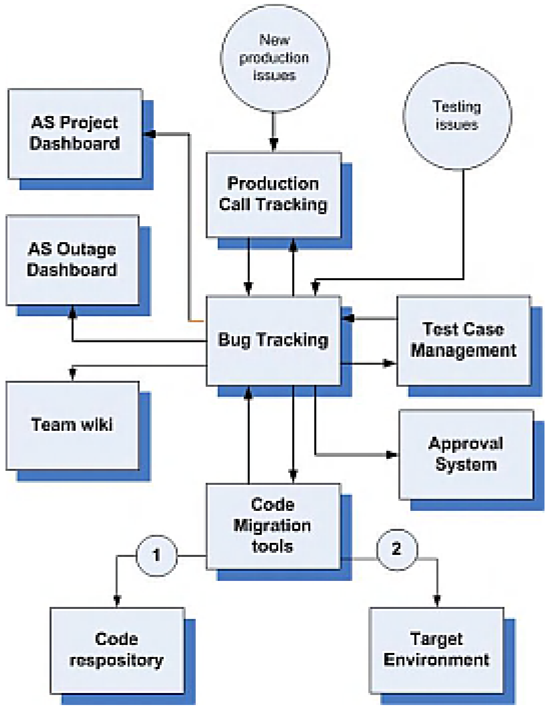
A key benefit from this tool is the accuracy of test execution progress. On many projects estimating progress in an objective manner is rarely done. Rather project team members invariably estimate based on "gut feel" as to their percent complete. These estimates tend to favor the optimistic and are typically wrong. Quality Center removes the guesswork from reporting testing progress subjectively, and maps progress to very concrete, objective criteria for a predictable, solid report to project and executive management.

Secondary benefits are as follows:

- Consistency in how tests are identified and tracked through the test execution process among a diverse group of users who report in to a wide range of organizations
- Enhanced reuse of tests
- Ease of creating test execution runs
- Transparency of test results

Tool Integration Roadmap

The three tools mentioned above are part of a larger roadmap of integrated tooling that will provide an holistic approach to the tracking and management needs of AS. The roadmap is comprised of off-the-shelf, vendor products and home-grown solutions with the intent of more effectively managing the AS systems and communicating changes. Today, four out of the nine applications are integrated.



Call Tracking. New production issues are initially logged and tracked in a Call Tracking system.

Bug tracking. 1) Issues that require technical work or technical assistance are automatically logged into the issue tracker. 2) New issues from various testing efforts are created and tracked in the issue tracker. This includes change requests, bugs, enhancements, tasks and supports a CR, find/fix, migration/approval process.

Team Wiki. Project, group and miscellaneous information is logged and published to all registered users. Issues are automatically published to the wiki. Users must be authenticated to view information.

Dashboards – Public websites. Executive view of AS projects and AS system outage. Details are found in the wiki, the issue tracker, team time tracker or MS-Project® which has a direct feed into the AS Project Dashboard.

Test Case Management – Test repository of test cases and test results. Test cases and test runs are created and run in support of all pre-production test efforts. Reports are published on the wiki project pages.

Approval system. Required approvals are sent via e-mail to university staff not in the issue tracker. Most other approvals for production migrations are handled directly in the issue tracker.

Code Migration tools (deployment manager tools) – Code fix migration information to any environment updated on issue tracker tickets for full end-to-end information about issues.

Code repository – Code repository that is referenced and tracked in deployment manager tools during builds and code migrations to desired target environment.

Tools not on the roadmap but that are used in AS and that foster collaboration are as follows:

- Shared calendar
- Instant messenger
- Online web conferencing
- LCD projectors that project one's laptop image
- Laptops for all staff and mobile devices for selected staff in support of the Stanford Work Anywhere initiative

PeopleSoft v9 Upgrade

So what do all these tools have to do with a real project? How are the delivery practices improved when a large cross-section of the university uses the same open tools? Well, a lot of great things happen. The reach is enormous and the results fantastic.

In 2003 Stanford University upgraded its PeopleSoft implementation from version 7.2 to 8.0. This project resulted in an enormous effort that involved many departments on campus. The primary vehicle for communicating among members was e-mails. Working spreadsheets, announcements and key information would be sent to members on e-mail distribution lists. As the project progressed, staying on top of key communications became more and more challenging as spreadsheets would be updated by multiple people and routed via e-mail, individuals would start silo'd discussions in e-mail inadvertently omitting others, and common, definitive posting of project information was not maintained.

In 2008 this changed as collaborative, server-based communications was instituted by AS/QA during the UAT (User Acceptance Test) phase of the project. Now team members (13 in number) would update wiki pages knowing with confidence that they were working with the most recent version of a page. All issues were tracked in Jira and test results tracked in Quality Center with stats posted weekly. The project dashboard represented accurate information as to overall project progress because the dashboard was populated directly from the time tracking system. Code fix migrations to the various pre-production environments were easily visible on the issue tracker tickets due to the code deployment integration. Essentially the goal was to make all project information and team involvement as open and transparent as possible and to introduce objective criteria by which to measure progress with the net result of building trust and a higher confidence level than on the earlier release. By and large this goal was achieved.

This upgrade was an enormous undertaking before it even reached the UAT phase. The project included the following pre-UAT activities:

- The services of an offshore IT vendor were engaged which included retrofitting of the application's 970 online and 380 batch customizations and conducting SIT (System Integration Testing) on Stanford systems. Over 25 engineers in three locations in India were involved in this effort along with an onshore staff of 5.
- The vendor's performance and quality of deliverables were audited by QA through careful analysis of breadth of testing and depth of coverage which resulted in an increase by 45 % more testing to close the identified gaps (for a total of 1,696 tests executed) and finding 1/3 of the missing tech docs.
- The AS analysts conducted ad hoc testing to complement the strategy employed by the vendor and logged new issues as well.
- Vendor and AS technical staff fixed and retested all logged issues.
- Over 10 environments were created throughout the project to support development, code control and testing.

During UAT, the following key activities took place:

- All collaborative tooling was spearheaded by the newly formed QA group with initial workspace and pages being set up and maintained by QA.
- All planning was done on the wiki and included the following information:
    1. Test team member profiles
    2. FAQ's for common questions raised in the beginning. Not maintained much once effort was underway.
    3. Weekly meeting notes and action items
    4. Team status and metrics (bug and test case metrics)
    5. Calendar (time-off plans, test cycle definitions, end-game plans, vendor contacts, DB links)
    6. Special test activities in support of parallel testing and Full System Integration Testing (FSIT) with FSIT ultimately being moved into Test Director

7. Test summaries

- Team members met once a week in person or dialed in if they weren't on campus. The Stanford business offices and groups that were involved were as follows:
    1. Admissions, which included undergraduate admissions as well as graduate school admissions for business (GSB) and law.
    2. Campus Community, comprising representatives from various offices
    3. Financial Aid, which included the central financial aid office as well as the GSB and Law School financial aid offices.
    4. Graduate Financial Support (GFS)
    5. HR, Benefits, and Faculty Affairs
    6. Payroll
    7. Registrar's Office, including undergraduate/graduate student records as well as the GSB and Law School
    8. Student Financial Services
    9. Oracle Financials (Controller's Office)
    10. ReportMart1 (RM1) reporting

    The following organizations not involved in this upgrade because their systems are separate from the university.
    1. Stanford Hospital, Medical Centers
    2. SLAC (Stanford Linear Accelerator)
    3. Stanford Management Group

- Test phases entailed the following:
    1. Smoke Testing. This testing was done by most offices at the beginning of UAT to ensure that the v9 functionality, customizations and environment were stable enough for further testing.
    2. Business Testing. This testing was completed by all Business Affairs offices to ensure that Stanford job objectives could be accomplished.
    3. Parallel Testing. Two payroll cycles were formally conducted (mid-month, month-end) to ensure that transactions (such as payroll, billing) in the production system matched the test system or were within acceptable limits in variance. Additional payroll testing was completed during the normal course of business testing.
    4. End-to-End Testing (FSIT). This testing was conducted at the end of UAT, involved all business units, both within Stanford and across vendors, working together to ensure that:
        1. Full system or life cycle processes (manual and/or automated, Registry, vendors, etc.) functioned as expected.
        2. All dependencies were accounted for and correct.
        3. The upgrade has not adversely affected the proper passing of data, events, files, etc.
    5. Performance Testing. This testing took place on the new production boxes to ensure that production performance was acceptable.
    6. Regression Testing. This testing was done throughout UAT to ensure that bugs were fixed correctly and that prior tests continued to provide the same results as before.
    7. Mini-FSIT. This testing was added to provide a higher level of assurance that the system truly was ready for Go Live. It was a repetition of some of the FSIT tests.

- Approximately 2,400 tests were run with test results logged in Quality Center.
- Approximately 1,000 issues were logged in the issue tracker with bug fixes tested by AS and QA analysts before being deployed to the primary UAT environment for final verification by the UAT testers.

- Team members were required to update their status on Confluence and in Test Director by 5 pm the day before the weekly meeting.
- QA Lead pulled defect and test case reports and posted in the wiki by 10 pm the same day as the leads.
- Project manager consolidated information into executive reports which were handed out at the weekly governing group meetings and posted on the Project Dashboard.

Team members who had participated in the upgrade years earlier repeated numerous times how much more effective and better the project was run this time around. Everyone appreciated the inclusiveness and openness that the collaborative tools brought to the team's effort. While the collaborative tools and the transparency of the effort were not perfect, it was a huge step in the right direction.

Adoption Challenges

Bringing new tools and processes to Administrative Systems was at times challenging for the following reasons.

Timesink

Change is not always appreciated as it involves an added burden to an already packed plate of work. The value-add of new tools must be compelling in order to break through the resistance that invariably occurs. This was fairly easy to achieve by using an open issue tracker. But, moving from e-mail to a wiki was slower as updating pages and maintaining information on the wiki was more burdensome. E-mail and IM are quick and easy (we're all used to this mode of communication now) but using a wiki requires a "rethink." The wiki has to be logged into, the user has to click on 'Edit', save their work periodically, clean-up the messed-up format of the page and then call it a day. Due to the current state of wiki functionality, it is not a perfect substitute for e-mail though there is a huge momentum by many wiki vendors to beef up the features. Regardless of the burden of wiki content maintenance (i.e., server-based communications), the effort is hugely worth it. Upon project completion, all team members would chant "Put it on Confluence!" when Go Live planning was underway. The benefits of open communications far out-weighed the added time (only in minutes) it took to bring the information to the wiki.

Open versus closed

In silo'd communications information can be more easily hidden or more easily taken out of context. In open communications this becomes much more difficult. If certain team members want to hide progress or project problems, it is considerably easier in the pre-collaborative mode. While it is possible in a collaborative world (just don't bother to track, publish or report on certain aspects of a project), the spirit of openness as represented by the collaborative tooling represents an opportunity for all team members to come together, educate others and truthfully represent what they are working on. This engages team members and management in a proactive manner and ensures that proper management support is brought to bear throughout the life of the project.

Timidity

One of the key hallmarks of wiki's is resistance to editing pages for fear of messing up (for all to see), losing key information (and not knowing how to pull lost information back) or stepping over someone else's edits (with no change marking present). Many people are more comfortable leaving someone else's work to stand on its own.[1] Most wiki's have minimized this by tracking all page changes and allowing users to reinstitute a prior page. To work through the initial timidity of using a new tool many users had to be explicitly directed to what page and where on the page to update. After a few times, everyone started to feel more comfortable and took ownership of their pages or sections of pages. All pages were by and large open to the team with certain critical pages having edit restrictions placed upon them.

Inability to turn off

In a collaborative world, the presence and availability of team members is more visible. We used IM extensively, received nightly news as to updated pages in the wiki over the prior 24 hours, could follow how an issue churned through the find/fix process by reviewing the issue history (who assigned an issue to whom and why), could get onto each other's calendar through shared calendaring and more. This on-going presence of the team 24/7 meant that it was at times difficult for team members to tune out, turn off, and recharge.

Conclusion

The AS/QA initiative of bringing open, collaborative tooling to projects had a major impact in building trust and confidence among all team members. While impacts to software quality are many (rigorous and inclusive review of requirements by all team members including the ultimate users of the solution, SDLC best practices, standard tools and supporting processes, independent testing, and much more), software quality is owned by all and supported through the use of collaborative tooling.

In fact, the experience by team members fit well to how Wikipedia defines collaboration[3]:

> *Collaboration is a recursive process where two or more people work together toward an intersection of common goals — for example, an intellectual endeavor that is creative in nature—by sharing knowledge, learning and building consensus.*

Recently I came across a slightly better definition of collaboration by Evan Rosen, *The Culture of Collaboration*.

> *A working together to create value while sharing virtual or physical space.*

The goal of openness, transparency, ownership and consensus among team members from many departments across campus was achieved on the PeopleSoft v9 project during the UAT phase and other AS projects through the use of these integrated collaborative tools.

Using cool new tools is always fun but these new tools alter the way we communicate in a fundamentally new way. Rather than to initiate the communication from our desktop (e-mail) in a client-centered view of the world, these tools allow us to communicate in a server-centered manner with an immediacy not realized to-date. We no longer push information out and wait for others to respond on their time table, rather we pull information in whenever, wherever and however we want it [2]; information that is open to anyone who cares to look. This engages team members more fully as they are not only passive viewers of the information, but in fact, active originators as well.

Bibliography

Books

Collaboration 2.0, Technology and Best Practices for Successful Collaboration in a Web 2.0 World, David Coleman and Steward Levine, January 2008. Published by Happy About.
[2] *The Culture of Collaboration, Maximizing Time, Talent and Tools to Create Value in the Global Economy*, Evan Rosen, 2007, Red Ape Publishing.
Unleashing Web 2.0 From Concepts to Creativity, Gottfriend Vossen, Stephan Hagemann, 2007. Morgan Kaumann Publishers.

Internet Links

[1] http://metamedia.stanford.edu/projects/traumwerk
http://c2.com/cgi/wiki?WikiDesignPrinciples
[3] http://en.wikipedia.org/wiki/Collaboration
http://en.wikipedia.org/wiki/Collaborative_software

Product Information

PeopleSoft Enterprise Campus Solutions® - http://www.oracle.com/applications/peoplesoft/campus_solutions/ent/index.html
Oracle Financials® - http://www.oracle.com/applications/financials/intro.html
Jira® - www.atlassian.com
Confluence® - www.atlassian.com
Quality Center® - https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24_4000_100__
Bugzilla - http://www.bugzilla.org/
MS-Project® - http://office.microsoft.com/en-us/project/default.aspx

It’s Not Just an Update: Using Status Reporting to Expand Collaboration

Michael Kelly

How quickly can you answer the following questions?
* What’s the status of your testing?
* What are you doing today?
* When will you be finished?
* Why is it taking so long?
* Have you tested _______ yet?

Your ability to answer those questions directly affects your ability to collaborate with those around you. Status reporting is an important collaboration tool, even if we don't often see it as such. It doesn't only communicate a boolean pass/fail, or a stop-light red/yellow/green. It can also communicate the intricacies of what you're working on, what you view as risky, and what can be done to help you be more effective in your work. In this paper, we’ll take a look at some of the aspects of "good" test reporting and two different examples of test reports. In the talk that this paper follows, we'll look at examples of how status reporting affects collaboration and we'll practice some exercises for more effective status reporting.

Good test reporting is difficult. Difficulties include tailoring test reports to your audience, clarifying confusion about what testing is, explaining how testing is actually done, and understanding which testing metrics are meaningful and when they’re meaningful. In addition, people to whom you’re providing your report frequently have assumptions about testing that you may not share. Some great examples of these assumptions:
* Testing is exhaustive.
* Testing is continuous.
* Test results stay valid over time. (My personal favorite.)

Aspects of a Good Test Report

Have you ever found yourself at a loss for words when someone asks for an ad hoc testing status? There you were, diligently testing, and someone asks how things are going, and then, bam—like some out-of-whack Rube Goldberg machine where the mouse flips though the air, misses the cup, and smacks against the wall—your mind freezes up. You mumble something about the technique you’re using and the last bug you found, and the person you’re reporting your progress to just stares at you like you’re speaking some other language.

That has happened to me more than once. But now I have a framework for thinking about my testing that lets me answer questions about my status with confidence and tailor my test report appropriately for the audience. In its simplest form, a test report should address a range of topics: mission, coverage, risk, techniques, environment, status, and obstacles. This holds true for both written reports and the dreaded verbal report.

Mission

A test report should cover what you’re attempting to accomplish with your testing. Are you trying to find important problems? Assess product quality or risk? Or are you trying to audit a specific aspect of the application (such as security or compliance)? In general, if you have a hard time articulating your status, it might be because you don’t have a clear mission. Having a clear mission makes it much easier to know your status, as you have a precise idea of what you’re supposed to be doing.

Coverage
A test report should include the dimensions of the product you're covering. Depending on your audience and the necessary completeness of your report, it may also include dimensions that you're not covering. A great heuristic (and the one I use the most) for remembering the different aspects of coverage is James Bach's "San Francisco Depot" (SFDPO):

* Structure
* Function
* Data
* Platforms
* Operations

Michael Bolton talks about SFDPO in his Better Software article on Elemental Models.

Risk
It's not enough to say what you're covering; you should also indicate why you're covering it. That's where risk comes in. What kinds of problems could the product have? Which problems matter most? You may find it helpful to make a mental (or physical) list of interesting problems and design tests specifically to reveal them.

Techniques
Once you've talked about what you're testing and why you're testing it, you might want to indicate how you're looking for it, if you think the audience is interested. That's easiest said by talking about specific test techniques. Examples would include scenario testing, stress testing, claims testing, combinatorics, fault injection, and random testing; this list goes on and on. Wikipedia has an excellent list of techniques with links to information on all of them.

Environment
The environment where your testing is taking place may include configurations (hardware, software, languages, or settings), who you're testing with, and what tools or scripting languages you're using. If you're using standard or well-understood environments for testing, you might just note exceptions or additions to the standard.

Status
Perhaps the most important aspect of a test report is your status. In this section of the report you answer questions like these:

* How far along are you?
* How far did you plan to be?
* What have you found so far?
* How much more do you have to do?

Tailor your status report to the needs of your audience: If they're concerned about potential risks, cover the negative. If they want to measure progress, cover the positive. In the same way, you can determine whether to present a status report that's summary or detailed.

Obstacles
A follow-up to your test status may include the obstacles to your testing. This can be simple:

"I didn't do X because of Y."

or detailed:

"If we had X we could do 20% more of Y and 10% more of Z, but that might also mean that we don't get around to testing W until Friday."

It can be helpful to think of questions like these:

* Do I have any issues I need help with?
* Is there anything I can't work around?
* Are there any tools that would allow me to test something that I can't test right now?

Putting It All Together

Figure 1 puts it all together to show the framework for test reporting. Notice that everything flows through the filter of your audience. If you're talking to the project manager, you might choose to focus on mission, status, and obstacles—talking about techniques and environment only when asked for details. On the other hand, if you're talking to a test manager, you might focus on risk and coverage, relating status and obstacles as appropriate. Finally, if you're talking with a developer, you might focus on techniques and environment. Each person gets a different perspective on what you're testing.

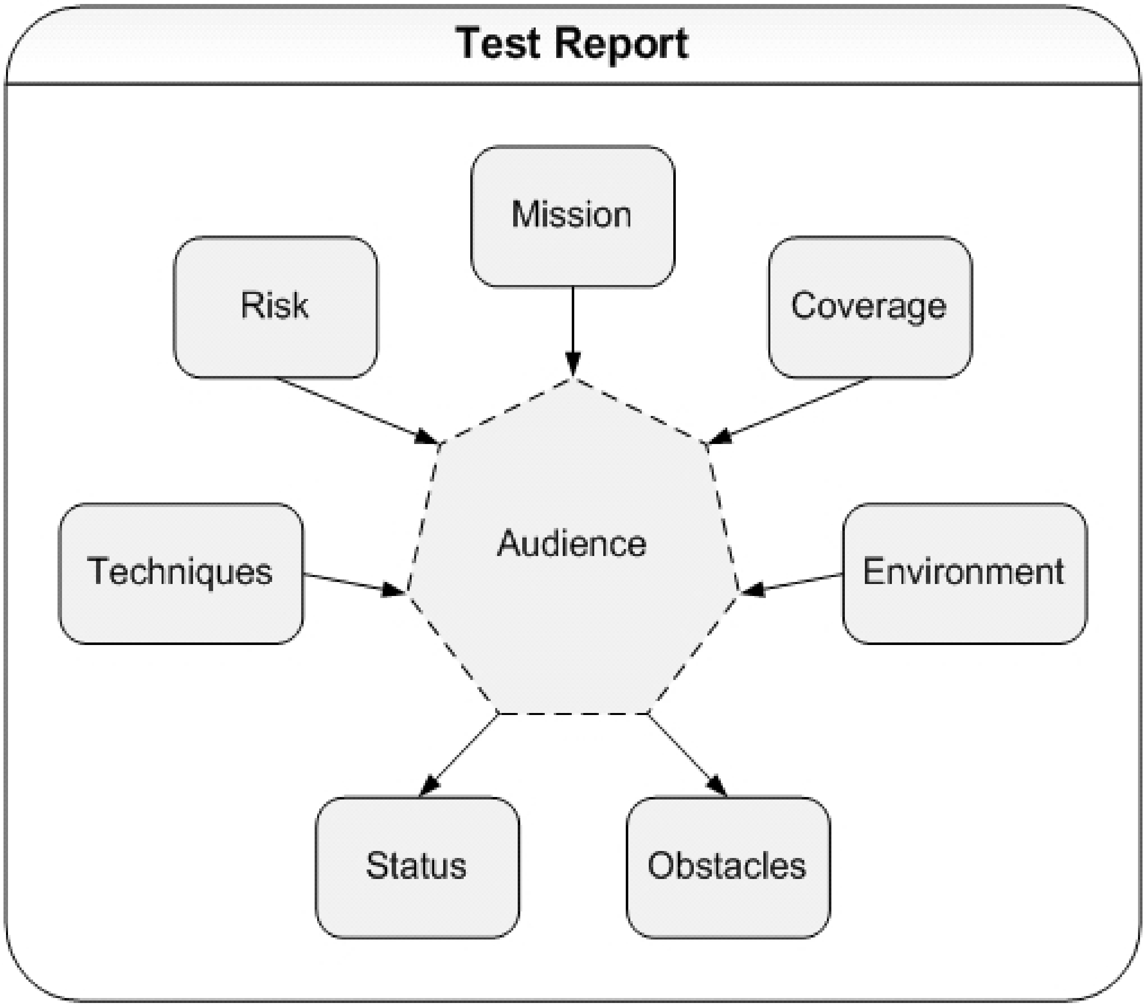


Figure 1 Framework for test reporting.

For any given aspect of your reporting, you need to be able to report at different levels of precision. Just as you filter what you report based on your audience, you need to be able to filter how much of something you report. Your mission can be a couple of words, or it can be a paragraph. Your risk and coverage can be entire documents, or sketched out on a napkin (something Scott Barber is fond of doing when modeling performance testing). James Bach talks about "dialing in your precision." A tester should be able to report to any level of detail, to any stakeholder, at any time. I like that idea. It makes me think of a testing control panel with different dials I can tune to get the report I need (see Figure 2).

Figure 2 "Dialing in" your reporting precision.

Examples of Test Reporting in Action

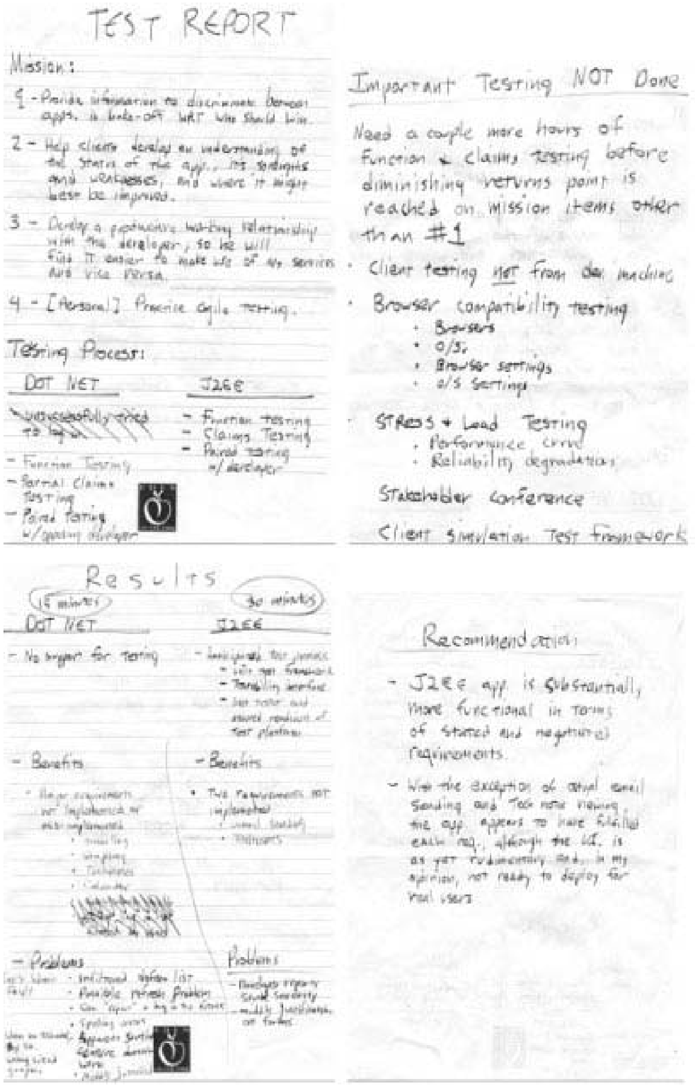
Here I'll share two examples of test reporting in action. Both examples are trivial in nature (what you might get after an hour of testing), but both illustrate the aspects of test reporting discussed earlier. As you read through the examples, you'll see some aspects called out specifically; others will be implied or included as part of the findings. Both of these reports are informal—that is, they don't follow a rigorous, predefined template—which is my preference. I'm not a big fan of templates, but that's not to imply that something like this couldn't be formalized for consistency across the team.

The first example is a test report I put together after about an hour of testing a sample application called ProSum, developed by Earl Everett. Notice that I call out mission, coverage, and risk explicitly. I talk about techniques and environment only as related to status. I also talk about what testing I didn't do.
Test Report on ProSum Version 1.4 by Earl Everett

```
Mission:
-----------------------------------
1) Provide information about the application in terms of potential defects.

2) Identify coverage, risk, and test strategy for this application.


Coverage:
-----------------------------------
Functionality:
- Application
 - Generate a random number
 - Spinner controls
 - Add numbers
 - Clear fields
- Error checking
- Calculation
- Testability

Data:
- Random numbers
- Bounds
- Types
- Rounding

Usability and Platform:
- User Interface
- Consistency
- Windows compatible
- Look and feel

Operations:
- Stress


Risks:
----------------------------------
- Incorrect addition
- Incorrect random number generation
- Incorrect error handling
- Other features implemented incorrectly
- Inconsistency with Windows features and user expectations
- Company image


Process:
----------------------------------
1) Variability Tour
 - I started with getting to know the application. Read About, clicked some
buttons, did the blink test.

2) Functional Exploration
 - I looked at functionality and used heuristic oracles based on consistency,
image, claims, expectation, and purpose.
 - I looked at usability in terms of me as a user and my expectations in terms of a
Windows application.
 - I did what limited stress testing I could think of that I thought was valuable.
 - The only data analysis I did was to identify boundaries for addition, determine
rounding for decimals, and to try some simple equivalence classes for input values.


Testing Not Done:
-----------------------------------
- There don’t seem to be any files written to the hard drive, but I did not do an
extensive check using any tools.

- Other than when pasting large character sets, there don’t seem to be any
noticeable performance problems on my laptop. However, I don’t know that there are
no performance problems that might not manifest themselves over time or with a high
volume of usage.

- I looked for a command line interface, but there does not seem to be one (or at
least I can’t get it to work).

- I did not test internationalization.

- I did not get to see the source code.

- I did not test on multiple platforms (only an HP Pavilion ze4600 running Windows
XP Pro SP2).


Results of Testing:
------------------------------
1) There are spelling, grammar, and stylistic problems on the About dialog. This is
inconsistent with purpose and is bad for the company image.

2) There are four buttons on the About dialog that allow you to exit the dialog. It
would appear they all do the same thing (close the dialog). This is inconsistent
with purpose because it may confuse the user (it confused me initially) and it is
inconsistent with the product because the error dialog has only one button (the OK
button) to exit the dialog.

3) On the About screen, the versions displayed (title bar and text) do not match.
This is inconsistent with the product.

4) The way Version is displayed in title bars is inconsistent. On the About dialog
the word Version is spelled out and it is 1.0 ("Version 1.0"). On the error dialog
it is just a V with a 1 ("V1"). And on the main screen it is V with a 1.0 ("V
1.0"). This is inconsistent with the product and bad for image.

5) The error dialog has some stylistic problems with the text (caps in the middle
of the sentence on "Integer", "Between", and "Only"). This is bad for image.

6) On the About dialog, the OK button has a hotkey and on the error dialog there
seems to be no hotkey for OK (or at least if there is, it is not the same as the
one on the About dialog). This is inconsistent with product.

7) Similarly for the main screen, each pushbutton has a hotkey with the exception
of the About dialog (as far as I can tell - if there is one it is not indicated in
the same way as the other buttons). This is inconsistent with product.

8) If I push the "Pick Two Random Numbers" button many times it does not seem to
generate negative numbers in the top field. This seems to be inconsistent with the
purpose of the random number generator.

9) If I push the "Pick Two Random Numbers" button many times it will sometimes
generate a 100 (or what appears to be a value in the 100 range) in both the bottom
and top fields. This is inconsistent with the product since valid values are from -
99 to 99.
10) The spinner control for each field seems to work independent of values entered
by hand into the text fields. This could be inconsistent with user expectations.
When I enter a 33 and click up, I expect a value of 34, not 1.

Etc....
```

You get the point. The full test report is available here: http://www.informit.com/content/images/art_kelly10_goodtest/elementLinks/testreport.zip

The second example, shown in Figure 3, comes from James Bach's Rapid Software Testing course appendices and is an excellent illustration of how simple a test report can be. (To see the images in full size, go to page 97 of the appendices.) Notice that this example includes a recommendations section. This is in line with the stated missions. Also worth noting is that only 45 minutes of testing seems to have been done (15 minutes on the .NET application and 30 minutes on the J2EE application). In just 45 minutes, you can already have this much to report!



TEST REPORT

Mission:

1 - Provide information to discriminate between apps. in bake-off WRT who should win.

2 - Help clients develop an understanding of the status of the app., its strengths and weaknesses, and where it might best be improved.

3 - Develop a productive working relationship with the developer, so he will find it easier to make use of my services and vice versa.

4 - [Personal] Practice agile testing.

Testing Process:

DOT NET

~~Unsuccessfully tried to login~~

- Function Testing
- Partial Claims Testing
- Paired Testing w/ [illegible] developer

J2EE

- Function testing
- Claims Testing
- Paired testing w/ developer

Important Testing NOT Done

Need a couple more hours of function & claims testing before diminishing returns point is reached on mission items other than #1

- Client testing NOT from dev machine
- Browser compatibility testing
  - Browsers
  - O/S.
  - Browser settings
  - O/S Settings
- STRESS + Load Testing
  - Performance curve
  - Reliability degradation

Stakeholder conference

Client simulation Test framework

Results

15 minutes

DOT NET

- No support for testing
- Benefits
  - Major requirements not implemented or [illegible] implemented
    - [illegible]
    - [illegible]
    - [illegible]
    - Calendar
- Problems
  - [illegible] list
  - Possible [illegible] problem
  - [illegible]
  - Spelling errors
  - [illegible]

30 minutes

J2EE

- [illegible] test [illegible]
  - [illegible] framework
  - Testability interface
  - [illegible] and [illegible] of test platform
- Benefits
  - Two requirements not implemented
    - email sending
    - [illegible]

Problems

- Developer reports [illegible] security
- [illegible]

Recommendation

- J2EE app is substantially more functional in terms of stated and negotiated requirements.
- With the exception of actual email sending and [illegible] viewing, the app. appears to have fulfilled each req., although the UI. is as yet rudimentary and, in my opinion, not ready to deploy for real users.

Figure 3 Hand-written test report by James Bach comparing J2EE and .NET applications.

I know that's difficult to read. That test report is available here: http://www.satisfice.com/rst-appendices.pdf

Next Steps

The first step to getting better at test reporting is to practice both written and spoken test reporting. A very easy way to do this is to partner with someone you work with (ideally several people) and at random points throughout the day ask for a quick one-minute or five-minute status report. (You'll be amazed how big a difference four minutes makes.) Also, write up a status report at the end of each day and send it to your team members for review and feedback. Can they understand what you did and didn't do? Do they know why you did it? Can they tell what techniques you used? You can also use a tool such as Spector Pro to record your test sessions so you can replay them and compare against your notes.

You may find it helpful to ask others to report their status to you. What types of questions do you find yourself asking? Take note of those questions and make sure that you're providing the same information when you report your status. What information do they provide that you really find useful? Use this information to develop a better model that fits your context. My model works for me, but yours might need to be different. Develop a model of your own. Once you have a model, share it with others, get feedback, and use it.

Armed with your new test reporting skills, you're ready to increase collaboration on your projects. Focus on reporting your mission (what you're trying to accomplish), coverage (what you're looking for), risk (why you're looking for it), techniques (how you're looking for it), environment (where you're looking for it), status (what you've found so far), and obstacles (what you could test if you have more resources). Either written or spoken, test reports that cover all of these dimensions have the makings of a good test report.

> **NOTE:** This paper was first published under the title "Dimensions of a Good Test Report" on InformIT.com. It draws heavily on the work of James Bach and uses several examples (such as the opening questions, his hand-drafted test report, and various other items) from his Rapid Software Testing course materials. In addition, I would like to thank Neill McCarthy and Scott Barber for their contributions.

Getting and Keeping Talent: Women in Software Development

Sharon Buckmaster and Diana Larsen, Futureworks Consulting
sbuckmaster@futureworksconsulting.com/dlarsen@futureworksconsulting.com

Abstract

Companies interested in gaining software quality through collaboration maximize the talents of their female software developers, testers, business analysts and quality assurance staff. Although women's participation is on the rise in many fields, including some of the traditionally male-dominated ones such as accounting and medicine, the percentage and number of women in the IT field is actually declining. The Computing Research Association reports fewer computing degrees awarded to women in 2004 than in 2000. Numerous academic and industry studies have documented that high exit rates for women from the IT arena contributes to an inability to fill roughly 500,000 information technology jobs nationally. With more than 50% of the current U.S. science, technology, and engineering workforce approaching retirement age, organizations must examine strategies to address the workplace conditions that attract capable women and men, and increase the likelihood of their continued employment.

Catalyst, a leading research and advisory organization, works globally with businesses to expand opportunities for women and business. In their 2007 landmark study on Women in IT, Catalyst examined drivers of satisfaction, retention, and advancement among women in technology. Learn to leverage these six drivers to recruit and retain talented women for your software development projects through an interactive discussion exploring which drivers make the most sense for your organization.

About the Authors

Sharon Buckmaster, Ph.D. and Diana Larsen are the principals of Futureworks Consulting, a firm specializing in bringing collaborative processes to organizations that want more productive, resilient workplaces. Both Sharon and Diana have many years of experience developing the generative capacities of individuals and teams that lead to higher quality products and services as well as a higher quality of organizational life.

Diana is known in the software industry for conducting project retrospectives and transitioning groups to Agile processes. She currently chairs the board of the Agile Alliance. Her publications include *Agile Retrospectives, Making Good Teams Great,* coauthored with Esther Derby. She consults and speaks internationally.

Sharon's research has focused primarily on women in leadership roles. She is the founder and past president of The Women's Center for Applied Leadership and is affiliated with the Center for Gender in Organizations at Simmons College. Sharon teaches in the Masters-level Applied Information Management Program at the University of Oregon and coaches executives and upper level managers.

The Current Situation for Women in Technology Positions

Many organizations face a serious constraint to growth in their inability to attract and retain sufficient numbers of qualified individuals to fill science, engineering, and technology positions (SET. This dilemma is particularly acute in the case of women. Contrary to popular belief, a large number of female scientists, engineers, and technologists have entered the workplace. However, these women don't stay in the technology field. They are abandoning their professions in droves. Numerous academic and industry studies document high exit rates for women from the IT arena, contributing to the daunting challenge of filling roughly 500,000 information technology jobs nationally (2007p. 121). With more than 50% of the current U.S. SET workforce approaching retirement age, organizations must examine strategies that attract and retain capable men and women.

According to The Athena Factor, a new Harvard Business Review Research Report, the female talent pipeline in science, engineering and technology (SET) for private sector firms is surprisingly deep and rich. Athena Factor survey data show that 41% of highly qualified scientists, engineers, and technologists on the lower rungs of corporate career ladders are female. Unfortunately, the female drop–out rate is huge. Fully 52% of highly qualified females working for SET companies quit their jobs, driven out by hostile work environments and extreme job pressures (Hewitt et al., 2008).

Looking more deeply to understand the reasons women exit the technology field, Catalyst recently completed a major study with 60,000 male and female respondents in global, high-technology organizations. Catalyst studied six areas:

- Companies as places to work
- Supervision and corporate leadership
- Career development and talent management
- Fairness and voice
- Job satisfaction, engagement, and commitment
- Work-life effectiveness

Study Results

For four of these areas, very few statistically significant differences between women and men, or between women in technical roles and women in all other roles emerged from the data. However, two areas showed important differences in perceptions. In comparison with men and with women in positions other than technology, women in technology roles were the **least satisfied with their supervisory relationships**. Tech women rated supervisors lower for three reasons: not giving adequate, timely feedback, lack of sufficient and effective communication, and lacking responsiveness to suggestions.

In addition, the findings show that women in technology roles were **less satisfied with their companies' approaches to fairness and voice** than *any* of the other comparison groups (Foust-Cummings, Sabattini, & Carter, 2008). For the purposes of this study, Catalyst defined fairness as procedures that result in the perception that management makes fair decisions regardless of a favorable or unfavorable outcome to the individual. Voice means having a say in the decision-making process, often contributing to perceptions of fairness.

Catalysts study data relates to the findings of other researchers noting that lack of respect poses a significant problem for many women in technology (Allen, Armstrong, Riemenschneider, & Reid, 2006). As Harris noted, "the isolated or single incident isn't the main problem. It's the culture of incidents; a lifetime of small, seemingly insignificant occurrences that create a hostile, even toxic environment" (1995,p. 121).

So What Should We Do?

Improving Supervisory Relationships

When asked, tech women suggested these steps as the most crucial to improving the supervisor-supervisee relationship:

- Communicating openly and directly with staff members
- Providing regular, performance-related feedback
- Providing access to more challenging "stretch" assignments with greater visibility
- Implementing stronger career and goal-planning processes

Enhancing Perceptions of Fairness

Tech women suggested that companies could enhance perceptions of fairness and voice by taking the following steps:

- Advancing and promoting more women
- Ensuring more diverse corporate leadership, particularly at higher levels
- Accepting diverse individual working styles

Successful Organization Initiatives Resulting in Higher Retention Rates for Women in Technology Roles…a few examples

Texas Instruments- reorganized hierarchical structures by creating multilevel, cross-functional teams. Women advanced more quickly because of increased visibility and greater access to developmental assignments.

Hewlett-Packard- organizes worldwide Technical Women's Conferences to showcase female engineers and scientists and to provide career development workshops.

IBM- The "Taking the Stage" program designed to show women how to achieve a strong leadership presence when speaking in any situation. Available to IBM women around the world, the four program components are accesses via intranet by individuals and small groups. Ideally, higher-level IBM females facilitate the groups, thus providing opportunities for role modeling and networking.

In Your Organization…

How can you benefit from this new information?

Determine whether this date rings true for your organization or what you notice in the industry

Talk about "fairness". What are the ingredients that contribute to fairness in technical workplaces? Learn what your female staff members think.

Do the recommendations of the women in this study appear to you to be the most critical issues to address…in the industry, in your organization, in your project team?

What other possibilities for organizational improvement that would result in higher retention rates for talented tech women come to mind?

How could you build a coalition of support to accomplish these goals?

Taking steps to improve the work environment for female staff members will improve your ability to recruit skilled male and female professionals.

References

Allen, M., Armstrong, D., Riemenschneider, C., & Reid, M. (2006). Making Sense of the Barriers Women Face in the Information Technology Workforce. *Sex Roles, 54*(11-12), 831-844.

Foust-Cummings, H., Sabattini, L., & Carter, N. (2008). Women in Technology: Maximizing Talent, Minimizing Barriers.

Harris, D. (1995). Grease the gears of equality. *Personnel Journal, 74*, 120-127.

Hewitt, S., Luce, C. B., Servon, L., Sherbin, L., Shiller, P., Sosnovich, E., et al. (2008). Athena Factor, Reversing the Brain Drain in Science, Engineering, and Technology. *Harvard Business Review Reports*.

Nobel, C. (2007). Why are Women Exiting IT? *Infoworld* (January 29 2007), 34.

Storytelling Techniques:

Reporting Product Status in a Meaningful Way

Karen N. Johnson

Who am I?

Karen N. Johnson	
Independent Software Test Consultant	www.karennjohnson.com
Hosted on Tech Target	http://searchsoftwarequality.techtarget.com
My blog	http://www.testingreflections.com/blog/3804
Co-founder of WREST workshop	http://www.wrestworkshop.com/Home.html
Director	http://www.associationforsoftwaretesting.org/drupal/executives

About this presentation

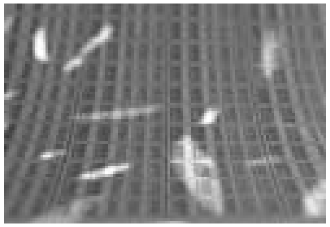
Let’s look at the elements of story.

- I’ve researched storytelling books, joined a storytelling guild, and attended storytellers events.
- Fascinating, right? But what did I learn and what can you get from this?
- Even better, how can you apply this to your work in software testing?

The Story

- Stories don’t fit in power point
- Stories don’t belong in bullets
- Stories belong in telling …

Stories bring data to life



> "Facts need the context of when, who, and where to become Truths."
>
> Annette Simmons
> The Story Factor

Storytelling Techniques © Karen N. Johnson, 2008 Slide 5

Mindmapping: the Story

Storytelling Techniques © Karen N. Johnson, 2008 Slide 6

Homework

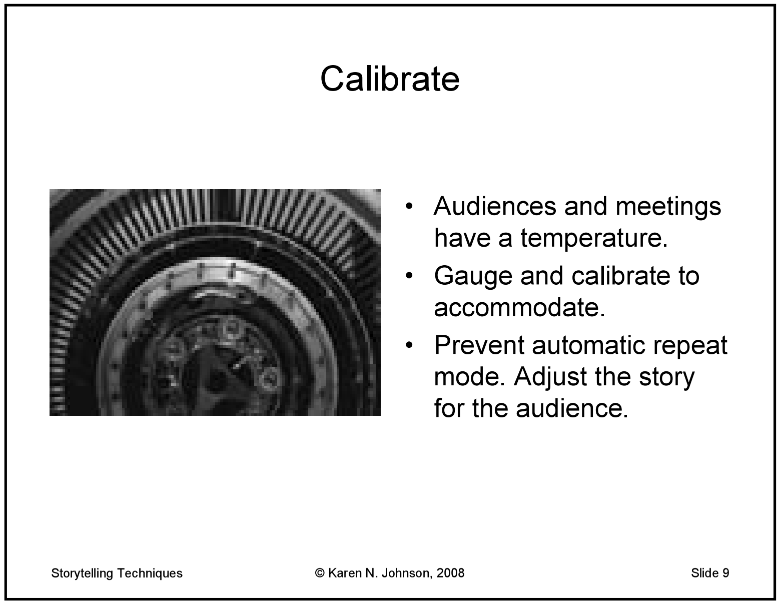
- There's no bypassing our investigation and analysis work.
- We still need to build, collect and interpret data.
- But in post analysis, there's an opportunity to use the story for delivery.
- There's room for story even in a hallway meeting.

Storytelling Techniques © Karen N. Johnson, 2008 Slide 7

Structure

- Identify the parts of the whole.
- Seek the narrative opportunities.
- Determine the type of story you want/need to build.

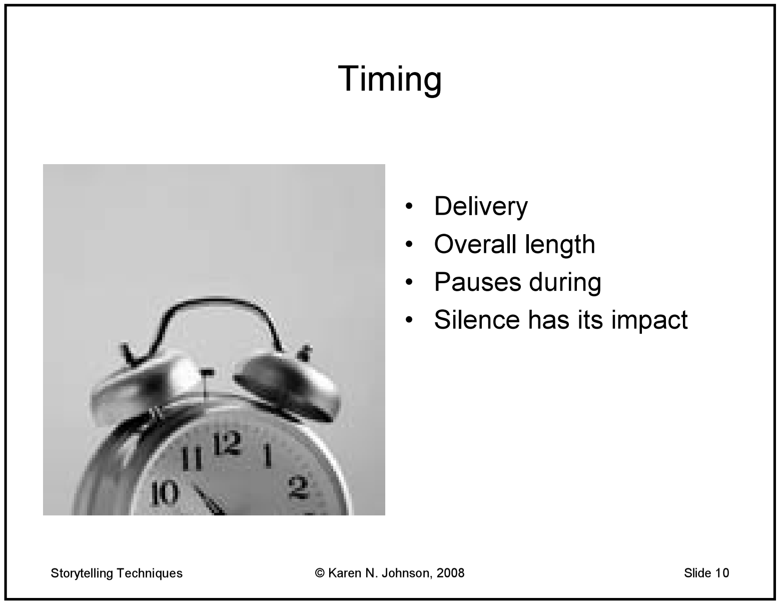
Storytelling Techniques © Karen N. Johnson, 2008 Slide 8



Calibrate

- Audiences and meetings have a temperature.
- Gauge and calibrate to accommodate.
- Prevent automatic repeat mode. Adjust the story for the audience.

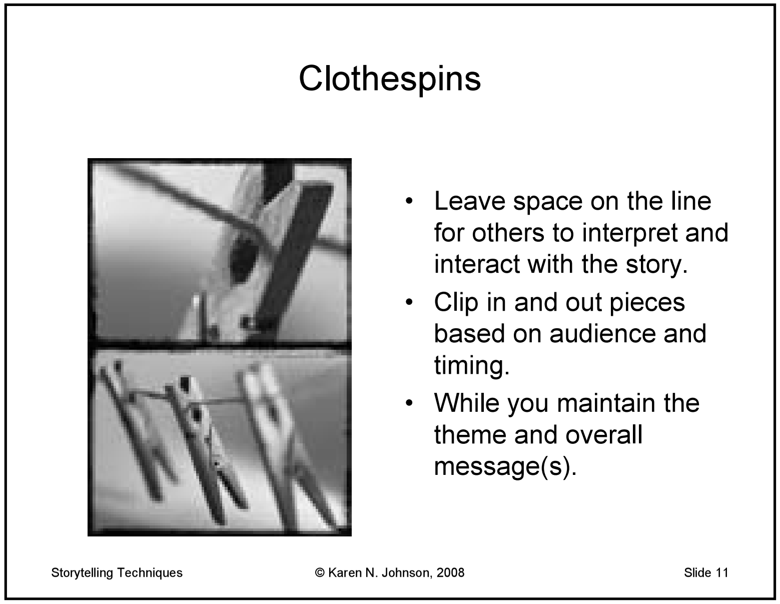
Storytelling Techniques © Karen N. Johnson, 2008 Slide 9



Timing

- Delivery
- Overall length
- Pauses during
- Silence has its impact

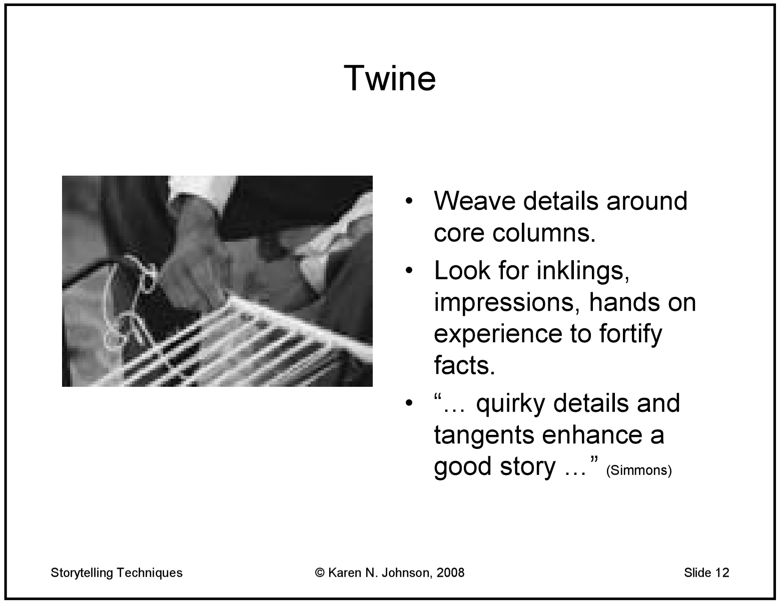
Storytelling Techniques © Karen N. Johnson, 2008 Slide 10



Clothespins

- Leave space on the line for others to interpret and interact with the story.
- Clip in and out pieces based on audience and timing.
- While you maintain the theme and overall message(s).

Storytelling Techniques © Karen N. Johnson, 2008 Slide 11

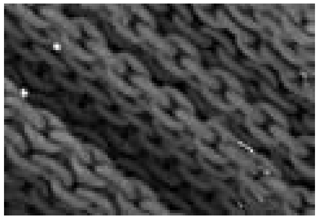


Twine

- Weave details around core columns.
- Look for inklings, impressions, hands on experience to fortify facts.
- "… quirky details and tangents enhance a good story …" (Simmons)

Storytelling Techniques © Karen N. Johnson, 2008 Slide 12

Linkage



- Transitions are essential in story.
- The linkage between the core columns create context, continuity and avoids the data dump.
- Storytellers often memorize key words. Phraseology matters.

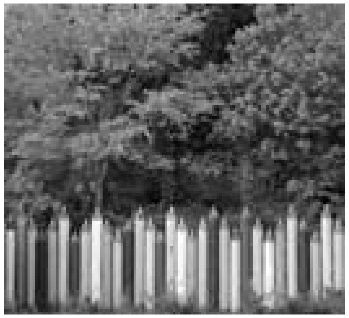
Roman Columns

- Know the core components of the story.
- Map out and memorize the core.
- With technical stories and details learn how to use the comprehension graph.
- Check-in with your audience to make sure each core column is understood.

Sensory Details

- Senses get dulled by volumes of data.
- Story gives opportunity to liven the senses.
- "… linear analysis misses the point…" (Simmons)

Think Unique



What do you have that could
be unique?

- A concept
- Memorable elements
- Interesting point of view

Send ahead's and leave behind’s

- Tie story to data.
- Data provides proof that backs the story.
- Data prevents story from becoming fable.
- To build a story of value, story has to correlate to facts.

Storytelling Techniques © Karen N. Johnson, 2008 Slide 17

Wrapping the pieces together: Opportunities to use Story

- Planning
- Execution
- Analysis & status
- Results
- Debriefs

Storytelling Techniques © Karen N. Johnson, 2008 Slide 18

Practical Application: Tying it together

Structure:

- What do you want to communicate?
- Monologue? Dialog?

Calibrate

- Who are talking with?
- Audience’s technical comprehension affects details offered.

Timing

- Snippet
- Full report

Storytelling Techniques © Karen N. Johnson, 2008 Slide 19

Practical Application: Tying it together

Clothespins

- Candidates for clothespins: metrics, milestones, on shore team vs. offshore team, test lab needs, project readjustments.

Twine

- How testing impressions relate to defects reported.
- How testing impressions relate to remaining work.

Storytelling Techniques © Karen N. Johnson, 2008 Slide 20

Practical Application:
Tying it together

Linkage
- Tester's hands on experience tied to critical defects.
- Tie reported customer issues to test strategy to open defects.

Columns
- Core information
- How can you organize information so you don't forget to deliver.

Sensory Details
- How can you deliver story so they don't forget what you've shared.

Acknowledgements

"The Story Factor"

Annette Simmons

Wonderful quotes, highly readable, practical applications to business.

"There are two main reasons people hold back they tell a story. The first reason is that they are afraid they will look stupid, corny, manipulative, or "unprofessional." … So we act "professional" and keep things tidy, logical, and rational. … Unfortunately our delivery becomes uptight, clinical, emotionless and b-o-r-i-n-g."

Acknowledgements

"Presenting to Win:

The Art of Telling Your Story"

Jerry Weissman

Concepts: Six roman columns, the data dump, the comprehension graph. Send ahead and leave behinds *

* Terms also referenced by Ed Tufte.

References

Title	Author
The Story Factor	Annette Simmons
Whoever Tells the Best Story Wins: How to Use Your Own Stories to Communicate with Power and Impact	Annette Simmons
Improving Your Storytelling	Doug Lipman
The Leader's Guide to Storytelling: Mastering the Art and Discipline of Business Narrative	Stephen Denning
The Springboard	Stephen Denning
Influencer: The Power to Change Anything	Kerry Patterson, Joseph Grenny, David Maxfield, and Ron McMillan

References

Title	Author
Rhetoric	Aristotle
Story Proof: The Science Behind the Startling Power of Story	Kendall Haven
Wake me up when the data is over	Lori Silverman
Nature Center Storytellers Guild	http://www.storynet.org/Programs/Guilds/il.htm
Beth Horner, storyteller	http://storytelling.org/Horner/

Selling Your Idea to Upper Management

Steven M. Smith

DAVID: Ruth, I think we should buy the ABC software to track trouble tickets and issues.

RUTH: There is no budget for that.

DAVID: But it takes me days to put together the information you want about the state of the product. And without an automated collection mechanism, I think many problems aren't being reported.

RUTH: Provide the best information available.

DAVID: @#!~

Like many technical people who don't know the basic ingredients and recipe for selling their ideas to management, David leaves this interaction feeling frustrated.

I've felt the hurt of management rejecting my ideas. When I look back on those experiences, I see a clear pattern—rejection was an understandable response to my failure to connect an idea to something that management considered significant.

Think back to a personal experience when someone was trying to persuade you to do something. If whatever they wanted you to do—such as buying insurance, volunteering your time, or making an investment—didn't offer you significant value, didn't you reject or ignore it?

Upper managers' thought process isn't different than yours. They need to know how an idea connects to something that is significant to them. If you want a thousand times better chance to sell an idea to management at any level, connect it to something that has significant value to that person.

Change the Perspective

Did David put Ruth's desires first during his dialog with her? No. His desires dominate the interaction. That's a huge mistake.

Effective selling focuses on the buyer, not the seller.

But David can change the perspective. It starts by doing research about Ruth. By recalling past interactions with Ruth, reviewing emails from her, and querying his network about her goals, David quickly finds many things that Ruth value. It surprises him that he never took the time to notice them before. Three things seem significant: 1) Ruth wants to hire additional testers; 2) her management has rejected her proposal to hire more testers; and 3) she wants her management to see that her organization is delivering more quality at a lower cost.

Change the Recipe

Learning about what matters the buyer is the vital ingredient for successful selling. But the flavor of that ingredient is enhanced when combined with other ingredients using a simple recipe.

The following recipe has served me well over the years when preparing for a selling interaction:

If you do X (the idea), you will get Y (the benefits).

Otherwise (if you do nothing) it will cost you Z (the cost of doing nothing).

Do I have your approval to do X?

In his first interaction with Ruth, David articulated X (the idea) but he didn't articulate Y (the benefits) and Z (the cost of doing nothing). And he didn't ask for approval to take action so Ruth didn't have to explicitly reject his idea. She was free to ignore it, which is what she did.

Using the recipe, David can put the ingredients of the interaction with Ruth in context. He sees that the idea (X) is for the company to buy the ABC software package to track trouble reports. The benefits (Y) to Ruth are related to the three thinks he discovered that are significant to her. The cost of doing nothing (Z) is unknown, which tells him to estimate that cost before the interaction. And finally the recipe reminds him to explicitly ask Ruth for permission to proceed with the purchase.

Change the Interaction

Let's look how David's interaction with Ruth might change after using the recipe and updating the interaction with what he has discovered:

DAVID: Ruth, if you approve the purchase of the ABC software to collect and analyze trouble ticket data, you could justify to your management the additional testers you've talked about hiring. The software will enable us to provide management with timely summaries of the troubles encountered by both the testers and beta clients. If we continue to be unable to articulate the quality of the product, we will continue to ineffectively prioritize the use of our developers. I estimate that poor prioritization is wasting 30% of the development's time, which works out to about $150K per month.

RUTH: I don't have the budget to buy the product.

DAVID: Are you willing not hire the additional testers you want and for the development organization to continue to waste $150K per month?

RUTH: @#!~

DAVID: Do I have your approval to buy the ABC software?

RUTH: Let me think about it.

DAVID: When should I check back with you?

RUTH: Friday.

Did David make an immediate sale? No. But Ruth did hear him and he is a thousand times better chance of action being taken on his idea.

Notice that David's interaction with Ruth also provided information to justify the idea to her manager, Stan. Selling to upper management starts by selling your manager and providing information that helps them sell their manager.

If Ruth rejects the ideas, David may want to appeal to Stan. He will want to change Y (the benefits) so they resonate with Stan. As you sell higher up the management chain, the benefits that matter come from the idea's impact on increasing revenue and reducing cost.

The same ingredients and recipe apply whenever you sell ideas to anyone, such as a teammate or client.

Summary

Many people with technical backgrounds consider "selling" to be a dirty word. That's a curious notion. I believe selling ideas is part of every aspect of life. Many organizations fail because they buy inferior ideas from people who know how sell. If you have superior ideas, I believe you owe it to your organization and yourself to learn how to sell them effectively. Start with the lesson of putting the desires of the buyer first and your own a distant second. And use the X, Y, Z ingredients and the recipe for combining them to sell your ideas to management. You will have a thousand times better chance of selling the idea.

.

Non-Regression Test Automation

Douglas Hoffman
Software Quality Methods, LLC.
Doug.Hoffman@acm.org
www.SoftwareQualityMethods.com

8/3/2008

Experience and qualifications:

Douglas Hoffman has over twenty-five years experience in software quality assurance. He has degrees in Computer Science, Electrical Engineering, and an MBA. He has been a participant at dozens of software quality conferences and has been Program Chairman for several international conferences on software quality. He architects test automation environments and automated tests for systems and software companies.

He is an independent consultant with Software Quality Methods, LLC., where he consults with companies in strategic and tactical planning for software quality, and teaches courses in software quality assurance and testing. He is a Fellow of the ASQ (American Society for Quality), founding member of SSQA (Silicon Valley Software Quality Association) and AST (Association for Software Testing), and is also a long time member of the ACM and IEEE. He is Past Chair of the Santa Clara Valley Software Quality Association (SSQA) and Past Chair of the Santa Clara Valley Section of the ASQ. He has also been an active participant in the Los Altos Workshops on Software Testing (LAWST) and dozens of its offshoots. He was among the first to earn a Certificate from ASQ in Software Quality Engineering, and has an ASQ Certification in Quality Management.

Non-Regression Test Automation

Introduction

In my experience, most automated tests perform the same exercise each time the test is run. They are typically collected and used as regression tests, and are unlikely to uncover bugs other than very gross errors (e.g., missing modules) and the ones they were specifically designed to find. Testers often think of test automation as GUI based scripted regression testing, using scripts to mimic user behavior. Tool vendors actively sell the automating of manual tests. These are very narrow views of the potentially vast possibilities for automating tests because they are limited to doing things a human tester could do. When we think of test automation we should first think about extending our reach – doing things that we can't do manually. This topic describes getting past the limitations of automated regression suites and generating more valuable kinds of test automation.

The difficult part of automation is determining whether or not the software under test (SUT) responds correctly. Automated tests can easily feed huge numbers of inputs to the SUT. Variation of inputs in automated tests can use data driven approaches or random number generators. In the absence of an excellent mechanism for recognizing expected SUT behavior (an oracle), verification is time consuming and extremely difficult. With an oracle, automated tests can be designed using potentially huge numbers of variable inputs to evaluate the responses of the SUT – without doing exactly the same test exercise each time. This is not to say that the tests are not repeatable, as described in the section **Mechanisms for Non-Regression Automation** below. This type of automated test has the chance of uncovering previously undiscovered bugs.

Regression Testing Defined

According to IEEE Standard 610.12-1990, "**regression testing** [is] *Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements*." Mathur defines "*the word regress means to return to a previous, usually worse, state. Regression testing refers to that portion of the test cycle in which a program P' is tested to ensure that not only does the newly added or modified code behave correctly, but also that code carried over unchanged from the previous version P continues to behave correctly*."[i] Wikipedia describes the idea as applied to software testing: "**Regression testing** *is any type of software testing which seeks to uncover regression bugs. Regression bugs occur whenever software functionality that previously worked as desired, stops working or no longer works in the same way that was previously planned. Typically regression bugs occur as an unintended consequence of program changes. Common methods of regression testing include re-running previously run tests and checking whether previously fixed faults have re-emerged*."[ii] Although the definitions are consistent with one another, common usage in software testing is the rerunning of previously run tests and getting the same results.

Sources of Regressions

From a practical standpoint, regressions occur due to poor source code management (SCM), incomplete bug fixes, or unintended consequences (side effects) of code changes and bug fixes.

The first two should not occur or should be immediately discovered. Unintended consequencs are difficult, if not impossible to eliminate because in this case the change in one place causes a bug in some remote part of the system. Unintended consequencs, by definition cannot be anticipated.

Arguably, regression tests were initially created as a response to SCM problems. However, bugs should not be introduced through poor SCM today because the principles and techniques were mastered in the 1970's and many tools exist which make SCM straightforward. Some organizations fail to follow well established source code management techniques, with the expected consequence of introduction and re-introduction of bugs. In these instances regression tests can find reintroduced bugs. This first category of regressions are virtualy eliminated by the use of current SCM tools. Regression tests that find this type of regressions are really finding development process problems.

Incomplete bug fixes may occur for many reasons; e.g., because of misunderstanding of the bug's scope, fixing of a different bug, resource constraints, or ineffective developers. These are generaly caught when every bug fix is verified in the initial build containing it. The likelihood of getting a complete fix is high when changes are fresh and the developers are most familiar with the code.

Although the regression test case may be used to verify the veracity of a fix, bug fix verification is usually done specifically for each fixed bug, not as a general part of running the entire regression suite.

Advantages of Manual Tests

Automation of a manual test reduces its variability. Even though a manual tester may be attempting to follow the same steps each time, humans are prone to making mistakes and the 'mistakes' sometimes lead to the discovery of new defects. Mis-keying an input, clicking on the wrong spot on the screen, typing "Yes" in a date field, or pasting a file's contents instead of its name are all examples that might turn up a bug. Even though the human may quickly correct for the error without apparent changes in SUT behavior, there is some chance that an error may be exposed by this activity. Variation leads to unexpected behavior, and in these situations validation and recovery are very difficult problems to solve in an automated environment. Automated regression tests always provide exactly the values programmed and expect the same result each time.

A regression test does the same thing over and over. It covers the same conditions and inputs each time it is run. Since software doesn't break or wear out, we would not expect to find a functional error the second (or subsequent) time we run a test on a build. [1] An automated version of the test becomes progressively less likely to find errors each time it is run.

[1] In some respects, the second and subsequent times we run a test we are looking for different kinds of errors – e.g., variable initialization, test design, interference problems, etc. The question we asked the first time the test was run has been answered, and each subsequent time it runs the software is starting from a slightly different set of conditions.

Limits of Regression Tests

As defined above, a regression test runs to test current versus previous behavior of the SUT. The majority of existing automated tests are automated versions of manual regression tests. These are not the most powerful tests we could make (in terms of the likelihood and types of bugs to be found) and there are important and valuable automation we could be creating. Even so, there are many circumstances where automating regression tests is well justified.[iii]

However, regression tests are usually not an effective way to look for new defects. The vast majority of defects ever found by automated regression tests are actually found by the manual running of the test prior to automating it.[iv] We further reduce the chance of finding bugs by doing *exactly* the same thing each time.

To better understand the likelihood of finding bugs with automated regression tests, try the following [mental] exercise. Think of how a [manual] regression test is created and compare that with creating a demo script.

Creating a Manual Regression Test

The most common regression test creation technique:

- Conceive and write the test case
- Run it and inspect the results
- If the program fails, report a bug and try again later
- If the program passes the test, save the resulting outputs as expected results
- In future tests, run the program and compare the output to the expected results
- Report an exception whenever the current output and the saved output don't match

What we do for a demo is try to minimize the chance of encountering a bug. Now, think of how a demonstration exercise might be created (e.g., for a trade show).

Creating a Demo

A demo creation technique for new products (finding a "happy path"):

- Conceive and create the demo exercise
- Run it to see that it demonstrates the desired attributes
- If the program fails, report a bug and either wait for a fix or find a different way to do the exercise that doesn't fail
- Remember the specific steps and values that work when the program passes
- In future demos, do not deviate from the steps or try new values to minimize the chance of failure
- Be at risk for embarrassment if the program fails to do the demo correctly

The two processes are nearly the same. The demo developer finds a path that works and then sticks to it. A regression test developer is doing virtually the same thing. Once the regression test has been created, the likelihood of encountering a bug has been minimized. The only bugs likely to be found are those reintroduced through poor source code management or gross errors (so obvious that any action will expose them). *Any* other exercise of comparable length will probably find more bugs because in addition to finding the gross errors there is some chance of finding legacy bugs that regression tests have no chance of finding.

Non-Regression Automation

There are many alternatives to automating regression tests. In addition to looking for more bugs by varying the tests, as mentioned in the Introduction, automated tests can extend testing into seeking new types of errors that can't be practically found through manual testing. Non-regression automated tests can do something differently each time they are run and most may not be performed manually at all..

Some types of errors sought through testing are only practical using non-regression automated tests:

- Buffer overruns, some types of security issues (e.g., found through potentially massive numbers of variations on input, huge input files, large data sets, etc.)
- Non-boundary special cases (e.g., internal boundaries, divide by zero, state machine errors)
- Memory leaks, stack overflows (e.g., accumulation errors)
- Memory corruption, stack corruption (e.g., in memory errors)
- Resource consumption/exhaustion (e.g., system effects)
- Timing errors (e.g., errors with small windows of opportunity)

Mechanisms for Non-Regression Automation

There are many types of non-regression automated tests. Most use a computer's pseudo-random number generator to introduce variation from run to run. Pseudo-random numbers are interesting because the series of values is statistically random but repeatable given a seed value. A test can generate a new random sequence or use a seed to rerun a sequence for fault isolation. Pseudo-code for the typical mechanism (for 100,000 possible seed values) is shown below:

```
IF (SEED parameter not present) /* Use SEED if it's present */
    /* Generate random seed if SEED not present */
    SEED = INT( 100000 * RND() ) /* SEED between 0 and 99,999 */
    PRINT SEED /* Print out SEED so the series can be repeated if necessary */
ENDIF
FIRST = RND(SEED)/* Use SEED to generate the first random number */
/* RND( ) will generate the next random number in the sequence after the first one */
```

The list below provides some of the types of non-regression tests in use today. Each is described briefly in the Appendix.

Nine examples of non-regression automation:

- Data driven / data configured
- Model based
- Random walks (both stochastic & non-stochastic)
- Function equivalence (using random input)
- A/B comparison (using potentially massive numbers of input values)
- Statistical models
- Real-time external monitors (e.g., memory leaks or data base corruption)
- Cooperating processes
- Duration testing, life testing, load generation (combining existing automated tests)

There are several approaches to automating non-regression tests based upon test style and how the test communicates with the SUT.

Test case style examples:

- Data driven commercial program
- Real-time monitoring utility
- Driver/stub combination
- Configurable/data driven custom program
- Individual program/test

There are also many touch-points where an automated test can interface with the SUT:

- Public API based
- GUI API based
- Non-GUI API based
- Individual program/tests
- Trusted objects

An example may serve to explain how the various characteristics work together. In the article, *Heuristic Test Oracles*[v] I describe testing a sine function in a system library. The test generates randomly selected sequences of inputs and checks to see that the sine function returns sequences of monotonically increasing or decreasing results. It looks for non-boundary special conditions (discontinuities) using a statistical model (monotonic increases and decreases) in an individual program (custom written test) that interfaces through the public API.

Test Oracles

The key element required to make any tests worthwhile is the oracle (method to check for expected/unexpected behavior). A person runs manual tests, using five senses and an untold number of creative oracles to observe and check SUT behavior. Whether using a specification or intuition, a person notices time between events, feels or hears clicks on a disk, sees sparkles on a screen, or smells over-heated wiring. Automated tests only check elements identified for it, and some kind of an oracle must be available to provide the individual values to determine whether or not the behavior is expected. Oracles for automated tests are critically important and can be quite varied.[vi],[vii],[viii]

Oracle examples:
Reference Functions – equivalent function or saved values

- Previous values or previous version
- Competitor's product
- Standard function
- Custom model

Computational or Logical Modeling

- Inverse Functions – "round tripping"
    - Mathematical inverse
    - Operational inverse (e.g., split a merged table)
- Useful mathematical rules –
    - e.g., sin2(x)+cos2(x) = 1
    - e.g., time of event order = event number order

Heuristic Functions[ix] – incomplete but usually right

- Almost-deterministic approach
    - Check only some of the outcomes
- Compare incidental but informative attributes
    - Durations
    - Orderings
- Check (apparently) insufficient attributes
    - ZIP Code entries are 5 or 9 digits
- Check probabilistic attributes
    - X is usually greater than Y
    - Statistical distribution (test for outliers, means, predicted distribution)

Conclusions

Regression testing is the rerunning of previously run tests and expecting the same results. Automated regression tests are the primary mechanism used in test automation. Although improvement in tools and software development processes have virtually eliminated the original cause of regressions, there are still circumstances under which regression testing makes sense. Yet, regression tests are weak at finding new bugs and do not extend the scope of bugs found. Like product demos, regression tests minimize the opportunity for finding defects by design. Automating regression tests further reduces the chance of finding defects.

There are a great number of alternatives to simple regression automation that provide powerful tests for finding bugs that manual testing cannot, such as memory leaks or subtle timing errors. Non-regression test automation gives us capabilities that a manual tester does not have.

Many different types of non-regression automated tests are possible and there are many mechanisms that can be used to build non-regression automation. They can be designed to look for types of bugs that typical regression tests cannot. There are also several test architectural styles, from custom program/tests to data driven commercial programs. Many possible touch points and monitoring techniques can also be used for SUT stimulation and result monitoring.

The key element required to make any tests worthwhile is the oracle. Oracles for automated tests are critically important and can be quite varied as described in the referenced papers and slides.

What automated tests can't do for us (yet):

- Know the expected results in all cases
- Notice things that we haven't specifically told the test (or the test mechanisms) to look at
- Analyze boundary conditions, partitions, models, etc., to determine the best test conditions to cover [With the exception of some tools that will tell us how to test that the code does what the code does]
- Decide on new courses of action (that aren't specifically written into the test) based on detection of potentially interesting occurrences

Appendix

Types of non-regression automation described

Data driven tests read input values with corresponding expected results. In this way the values being used in the tests can be repeated or different each time. Data input is not restricted to specific arithmetic or alphabetic values but may include function calls, parameter names, and other application specific attributes to be tested.

Data Configured tests read input values to enable/disable or configure the test case code. (e.g., printer paper sizes or printer dot densities)

Model Based tests use a model of the SUT. (e.g., a state machine or menu tree)

Random Walks use a series of pseudo-random values as input.

Stochastic tests are ones where the sequence matters. (e.g., logging-in before beginning a financial transaction)

Non-Stochastic tests are ones that are [theoretically] independent of one another. (e.g., order of loading of printer fonts)

Function Equivalence is a form of random walk when a true oracle is available and therefore any random input can be used whether the function is stochastic or not. (e.g., there are two versions of the same product)

A/B Comparison is used when a very large number of results are recorded and compared from one run to the next (e.g., potentially massive numbers of result values)

Statistical Models

Real-time External Monitors are programs run at the same time as the test but are monitoring characteristics external to the actual SUT (e.g., memory leak or data base corruption detectors)

Cooperating Processes are tests in which the test or monitor is communicating as a peer process with the SUT.

Duration Testing is done by running a series of tests (or one test repeatedly) continually for a period of time.

Life Testing is done by running a series of tests (or one test repeatedly) continually until the system fails.

Load Generation is one or more series of tests run in the background when a specific test is run. (e.g., doing performance analysis or to test that a file can be accessed even though other processes are accessing it)

Statistical Model is based on the statistical characteristics of the data rather than the data values. For many functions, input with a specified mean and standard deviation will generate results with a corresponding mean and standard deviation. Many other statistical characteristics are available and may be used, especially when dealing with very large data sets and unpredictable expected results.

References

[i] Mathur, Aditya P., **Foundations of Software Testing** (2008, Dorling Kindersley (India) Pvt. Ltd) ISBN 81-317-1660-0

[ii] June 14, 2008 at http://en.wikipedia.org/wiki/Regression_testing

[iii] Bach, James; "*Reasons to Repeat Tests*" http://www.satisfice.com/repeatable.shtml

[iv] Kaner, Cem, "*Avoiding Shelfware*" http://www.kaner.com/pdfs/shelfwar.pdf

Merick, Brian; "*Classic Testing Mistakes*" http://www.exampler.com/testing-com/writings/classic/mistakes.pdf

[v]] Hoffman, Douglas "*Heuristic Test Oracles*" STQE Magazine, April, 1999 http://www.softwarequalitymethods.com/Papers/STQE%20Heuristic.pdf

[vi] Hoffman, Douglas "*Using Test Oracles in Automation*" Software Test Automation Conference, Spring 2003 http://www.softwarequalitymethods.com/Slides/Oracle%20Auto.pdf

[vii] Hoffman, Douglas "*Advanced Test Automation Architectures*" Conference for the Association for Software Testing (CAST), 2007 http://www.softwarequalitymethods.com/Slides/TestAutoBeyondX2-CAST07.pdf

[viii] Hoffman, Douglas "*Using Oracles in Automation*" PNSQC, 2001 http://www.softwarequalitymethods.com/Papers/Auto%20Paper.pdf

[ix] Hoffman, Douglas "*Heuristic Test Oracles*" STQE Magazine, April, 1999 http://www.softwarequalitymethods.com/Papers/STQE%20Heuristic.pdf

PSNQC Conference Paper
Presenters: Jagannathan Venkatesan and Craig Merchant
Contributor: Manuel Tellez
Microsoft Corp.

Web UI Automation – A Browser Agnostic harness for Web UI Testing

Abstract

This paper describes a test harness that can be used for developing test automation for websites. The harness provides a set of classes that a test developer can use to develop custom coding solutions for controlling (automating) and verifying websites. The harness is completely browser agnostic meaning a single code base is developed that can execute on multiple browsers. The current implementation is targeted for the Windows platform.

Introduction

Last year, our department began development of a server management website. Our team (the UI test team), was given the challenge of developing a test harness that would allow for the development of a completely automated regression suite and would give the capability of running against multiple browsers using a single code base.

Our initial idea was to develop a harness compatible with IE7 (Internet Explorer 7) and Firefox. However, we wanted to make it easily extendable to other browsers and wanted to shield the test developer from needing to write two sets of code implementing the same test case.

We addressed this challenge by developing a set of classes that allows the tester to author tests independent of the target browser (IE7 and Firefox in our case). The test code implements a single test case that could be targeted towards different browsers. The test code is completely browser agnostic. Nowhere would one read anything specific to a particular browser. The following example shows our intention:

Scenario 1

Test Spec

1. Open the browser and navigate to a live.com
2. Enter search criteria "cats"
3. Click search button
4. Verify www.catsite.com is found
5. Click on the link if found
6. Close the browser

Implementation (shown in pseudo code)

```
//Create browser control object , browser
browser = new Browser()

//navigate to live.com
browser.Navigate("http://www.live.com");

//try to find the expected search result
browserInput input = FindInputElement("www.thecatsite.com");

//verify the search result
If (input == null)
        LogFailure
Else
        LogSuccess
        Input.Click

Browser.Close()
```

Existing Solutions

Microsoft has a significant number of test teams writing website automation. We define website automation as the ability to programmatically control a website and verify content of the website. We found that most solutions fell into one of three buckets: Record and Playback methodologies, JavaScript solutions, and UI Automation based solutions.

The characteristics of these solutions appear below:

Record and Playback

1. Advantages
   - Easy to record a script. Just Point and click. A script of all user input is created automatically from the recording.
2. Disadvantages
   - Different scripts need to be recorded for different browsers. This type of approach is definitely not browser agnostic.
   - A slight change in the UI requires scripts to be re-recorded. Thus, recording tasks must be duplicated for each UI revision.

JavaScript

1. Advantages
   - Low level DOM(Document Object Model) access.
   - No special development environment required other than cscript. Notepad is the editor of choice.
2. Disadvantages
   - It is generally harder to debug JavaScript than it is to debug higher level languages. We need the ability set break points for failure investigation instead of relying on print statement debugging.
   - JavaScript is procedural language vs. object-oriented which in our experience we have observed naturally caters to UI automation).

UIAutomation

1. Advantages
   - Provides extremely low level access to all controls on the window.
   - Fast and reliable.
   - .Net classes available in System. Automation
2. Disadvantages
   - Different browsers have different automation ids for the same control. In fact, the programmer has no control over the automation id. Browser agnostic tests would not be possible here.

We found that no single solution satisfied all our requirements. Thus we decided to produce a custom harness. The design of the harness is detailed in the next section.

Design

History

Our test team does leverage existing tools when possible. Prior to the proposed web based management solution, our team worked on traditional windows software. Our team developed DFS (Distributed File System) Management Console, Share and Storage Management Console, and some other management applications on Windows Server 2008. From our prior work in UI automation, we had a UI automation harness. The harness was designed with extendibility in mind. It provided a skeleton on which to build the new web automation harness. Building off the existing harness allowed our new tests to have an identical structure to our non-web based tests and allowed us to reuse code for common functions like logging.

Our existing harness was designed to allow users to mix and match back end automation techniques. Thus the browser agnostic web automation harness was a natural extension and did not involve any re-design work. We designed two new plug and play components for dealing with Firefox and IE7. At runtime, the harness loads the desired plug and play component depending on the browser being tested and from there the harness uses the loaded plug and play component to control the browser.

Additionally, our current harness is .Net based (C#). Test development and debugging can be done in MS Visual Studio. Being implemented in a high level object oriented language makes it accessible to a wide developer audience.

Structure

The harness is structured such that a test case derives from the BaseUITest class which derives from the TestCaseBase class. This object oriented approach allows the tester to immediately have access to pre-existing functionality such as logging, environment management for running in different machine/domain settings, resource management for localization testing, performance monitoring for tracking runtime statistics such as memory usage and CPU utilization, test data management, and other areas. The tester can extend or override existing code in order to implement the test case. The test developer can instantiate or use any of the provided objects. For today's discussion, we will discuss Browser classes which are used to control and test the website. Recall that the test developer is not aware of the particular browser being tested. The developer is only dealing with an object that wraps common browser functionality. The browser objects are DOM-like meaning that the test developer who is familiar with DOM will be very comfortable with these classes.

The harness has two main layers. They are the Test Case Layer and the External layer. (Please see Figure 1 for a block diagram of the layers.)

The TestCase layer provides several static classes such as a logger, an environment mapping resolver for running in particular machine/domain environments, a resource manager for providing support for running the tests in different locales, and a performance monitor for collecting performance statistics. It also provides several dynamic classes for dealing with user interfaces. This is the layer at which we added an extension to allow test cases to control the browser.

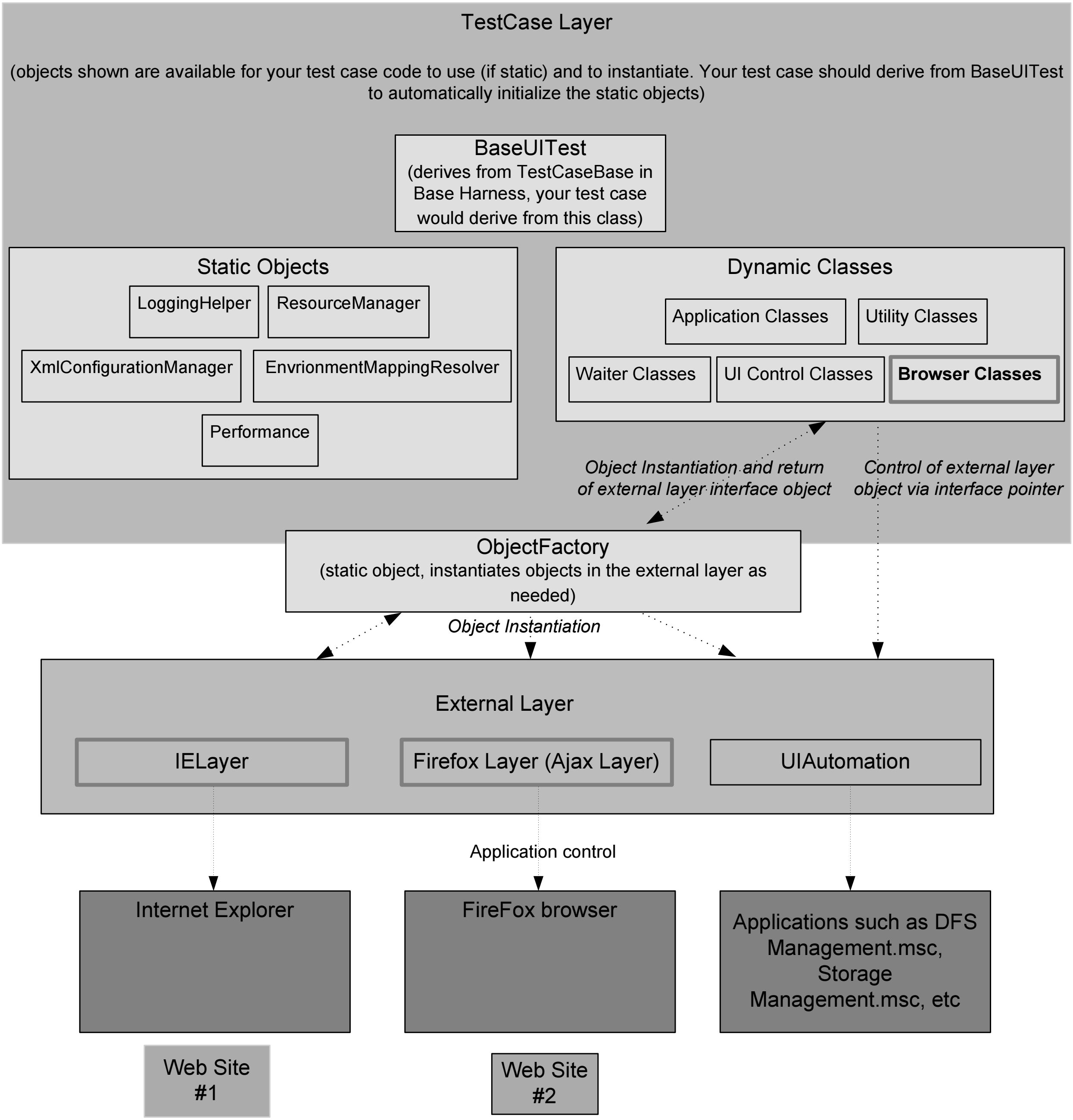
The External Layer level provides specific control to particular browsers such as IE7 and Firefox. It is at the external layer where we added the plug and play components to control IE7 and Firefox. When we say plug and play, we are indicating that each component will implement the same interface. In this way, the objects in the test case layer can instantiate the objects in the external layer and use them interchangeably to achieve browser agnostic behavior. The test developer deals with the objects in the test case layer only and is unaware of which plug and play component has been loaded. He/she simply calls into the test case layer objects which handle the communication to the external layer component.

Comparison to Existing Approaches

The following chart shows how our design compares to the existing approaches to web automation mentioned above,

	Browser Agnostic	Low level access to all controls	Debugging capabilities	Object oriented	Special Dev environment	Resilient to UI changes	Requires custom programming
Record & Playback	NO	NO	YES	NO	NO	NO	NO
Javascript	YES	YES	NO	NO	NO	YES	YES
UIAutomation	NO	YES	YES	YES	YES	YES	YES
Our harness	YES	YES	YES	YES	YES	YES	YES

Design Diagram (Figure 1)


TestCase Layer
(objects shown are available for your test case code to use (if static) and to instantiate. Your test case should derive from BaseUITest to automatically initialize the static objects)
BaseUITest
(derives from TestCaseBase in Base Harness, your test case would derive from this class)
Static Objects
LoggingHelper
ResourceManager
XmlConfigurationManager
EnvrionmentMappingResolver
Performance
Dynamic Classes
Application Classes
Utility Classes
Waiter Classes
UI Control Classes
Browser Classes
Object Instantiation and return of external layer interface object
Control of external layer object via interface pointer
ObjectFactory
(static object, instantiates objects in the external layer as needed)
Object Instantiation
External Layer
IELayer
Firefox Layer (Ajax Layer)
UIAutomation
Application control
Internet Explorer
FireFox browser
Applications such as DFS Management.msc, Storage Management.msc, etc
Web Site #1
Web Site #2

From a tester's vantage point, he derives a single class from BaseUITest and then uses any of the static objects and instantiates any other object as needed in the test case layer in order to write the test case code. Note, the test case code doesn't know anything about the target browser. The harness handles the target browser.

Implementation

Test case layer – Browser Objects and the Object Factory

The test case resides in the test case layer and can instantiate any existing test case layer object or use any available test case layer static object. In the case of website testing, these are the Browser Objects. The Browser objects call into the corresponding external layer (either IE or FF) to control the specific browser. The tester and his code are blissfully unaware of the browser they are running on as all of the browser communication is abstracted away.

There are two factors that make the abstraction possible. The *first factor* is that the browser object creates an instance of the corresponding External layer object and caches that object. The object that the test case layer caches is actually an object implementing an interface. In other words, we have an interface pointer. *Secondly*, the test case code calls into the browser object and then the browser object calls corresponding methods in the external layer interface object. Since the IE layer and FF layer implement the same interfaces, how the corresponding external layer talks to the particular application/browser is of no concern to the test case layer.

Several selected TestCase layer browser objects are described below and related back to Scenario 1. Appendix A shows a complete listing of objects and their methods and properties.

BaseBrowser

The responsibility of this class is to instantiate the external layer's browser object and to call methods of the external layer browser object. By instantiating the external layer browser object, the browser window is actually displayed. For example, the constructor is defined as:

Method 1

```
public BaseBrowser()
{
    m_Browser = CObjectFactory.CreateObject(OBJECTID.CUIharnessBrowser)
     as IUIharnessBrowser;
}
```

Here you can see that we call into the object factory. The object factory (see below) takes an object id; in this case, the CUIharnessBrowser and it returns an interface point to IUIFramworkBrowser.

Relating to Scenario 1, instantiating this object opens the browser (either FF or IE7 depending on the external layer being used). The **test case code** to instantiate this object is,

Browser browser = new Browser();

The test developer is then free to call methods in this class such as:

Method 2

```
public bool Navigate(string URL)
{
    bool bRet = m_Browser.Navigate(URL);
    Refresh();
    MainWindow.BringWindowToTop();
    MainWindow.Maximize();
    return bRet;
}
```

Here, we call the external layer object's Navigate method. After navigation is complete we call the browser's refresh method (Note, refresh in this context means to clear any cached items the browser object may have and re-instantiate them on an as needed basis).

Relating to Scenario 1, the ***test case code*** would call,

Browser.Navigate(http://www.live.com)

Or,

Method 3

```
public BaseBrowserElement ElementFromCond(IBaseBrowserCondition cond)
{
    BaseBrowserElement elRet;

    foreach (BaseBrowserDocument doc in Documents)
    {
        elRet = doc.RootElement.FindDescendant(cond);

        if (elRet != null)
        {
            //set the refresh condition for this element (tells how the
            element was found)
            elRet.RefreshCondition = cond;
            return elRet;
        }
    }

    return null;
}
```

Here, we search all Documents in the browser for elements matching a particular condition. In the case of Scenario 1, we are looking for the element with an href equal to http://www.thecatsite.com/. The ***test case code*** for finding this element is,

```
BaseBrowserElement element = browser.ElementFromCond(new
        BaseBrowserCondition("href", http://www.thecatsite.com)
```

BaseBrowserElement

The responsibility of this class is to provide methods for the test case so that it can interact with the html element on the web page. A constructor is shown below. It takes as input an element id and the BaseBrowser containing the element. It finds the element using the browser object's ElementFromCond method and caches the corresponding external layer object. Method 2 shows how the cached object is used to click on the element.

Method 1

```
public BaseBrowserElement(string elId, BaseBrowser browser)
{
    m_Browser = browser;
    m_RefreshCond = new BaseBrowserCondition("id", elId);

    m_Element = browser.ElementFromCond(m_RefreshCond).NativeObject;
}
```

Method 2

```
public virtual void Click()
{
    if (Browser.UseDOM)   //using DOM (Document Object Model) methods
    {
        m_Element.Click();
        return;
    }

    Point pt = new Point(BoundingRectangle.X +
                    BoundingRectangle.Width/2,
                    BoundingRectangle.Y + BoundingRectangle.Height/2);

    BaseUIObject obj =
          BaseUIObject.UIObjectFromPoint(Browser.MainWindow, pt);

    obj.Click(pt);
}
```

This method shows two things. First, it shows we are calling the external layer element's click method if a flag is set. Also, it shows that if we do not want to call the external layer's method, we can click on an element directly using UIAutomation if we can find the element's (X,Y) position. Since the external layer element gives us this, we can retrieve the UIAutomation object from its coordinates and click it via UIAutomation.

Relating to Scenario 1, the ***test case code*** call to click on the element is

Element.Click()

Scenario 1 implementation (Test Case Code)

The follow code is test case code. It shows how the tester would call into the testcase layer browser objects in order to implement scenario 1.

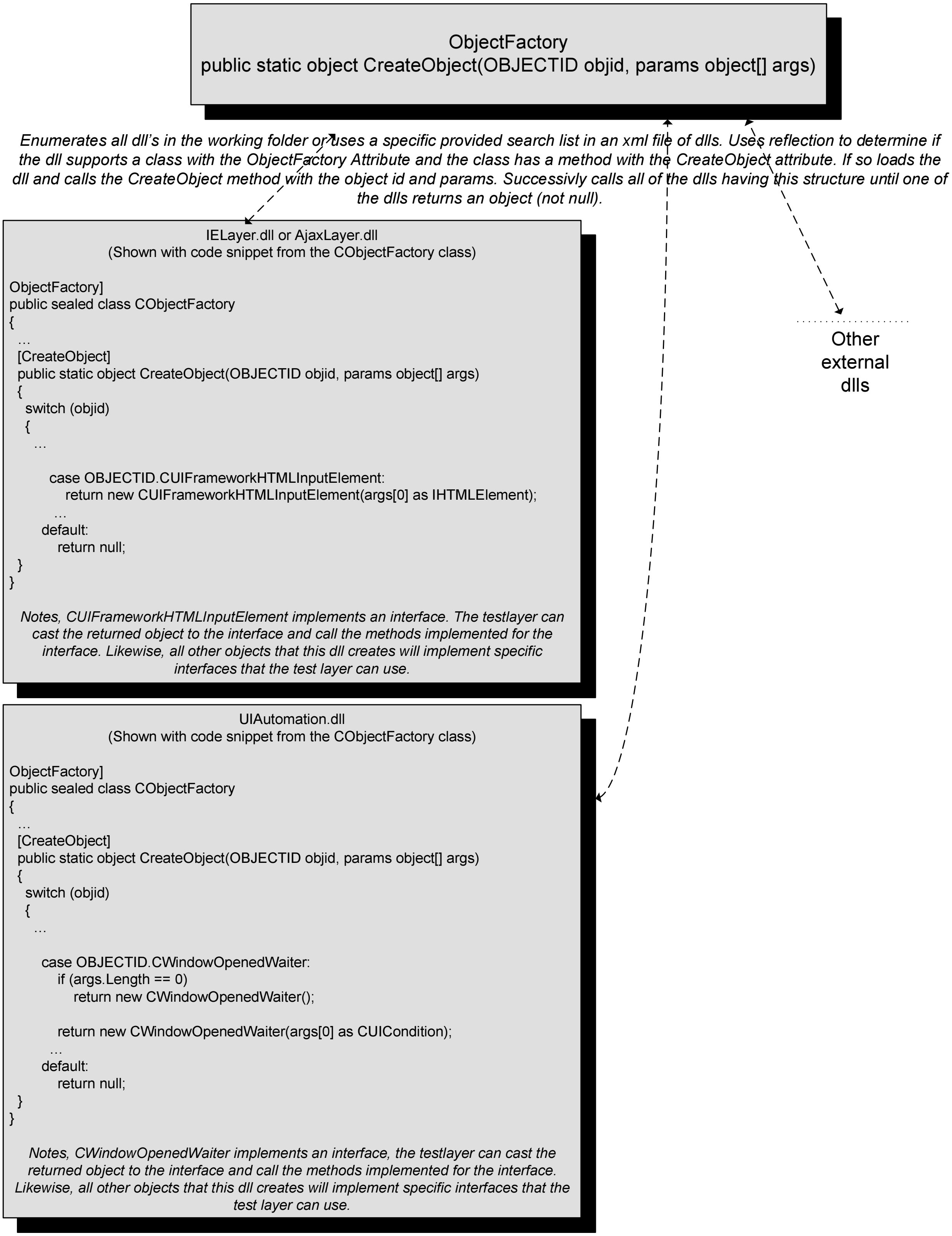
```
Browser browser = new Browser();
Browser.Navigate(http://www.live.com)
BaseBrowserElement element = browser.ElementFromCond(new
     BaseBrowserCondition("href", http://www.thecatsite.com)
If (element == null)
      LoggingHelper.LogFail("Unable to find www.thecatsite.com");
Else
{
      LoggingHelper.LogInfo("Successfully found the link");
      Element.Click();
      Browser.WaitForDocumentComplete();
      //Do some additional validations here
      LoggingHelper.LogPass("Successfully clicked on the element
      and went to website");
}
Browser.Close();
```

Creating the Desired External Layer Object

Up to this point, we have just assumed that the test case layer is able to pick out the desired external layer and instantiate objects on it. There are two methods that can be used to help the harness load the correct external layer dll. If a specific known external layer dll is desired, the test can specify it using xml. Otherwise, the harness scans all of the dlls in the current working directory until it finds the first dll that contains certain methods and attributes (see illustration below).

External layer objects are created using an object factory. The harness provides a preset list of object ids (for example, OBJECTID.CUIharnessBrowser). The test case layer passes the object ids into the object factory of the external layer dll. If the external layer dll can create an object with the particular id, it attempts to do so, otherwise it returns null and the next external layer dll is queried. This can be viewed as follows:

Figure 2



ObjectFactory
public static object CreateObject(OBJECTID objid, params object[] args)

Enumerates all dll's in the working folder or uses a specific provided search list in an xml file of dlls. Uses reflection to determine if the dll supports a class with the ObjectFactory Attribute and the class has a method with the CreateObject attribute. If so loads the dll and calls the CreateObject method with the object id and params. Successivly calls all of the dlls having this structure until one of the dlls returns an object (not null).

IELayer.dll or AjaxLayer.dll
(Shown with code snippet from the CObjectFactory class)

```
ObjectFactory]
public sealed class CObjectFactory
{
  …
  [CreateObject]
  public static object CreateObject(OBJECTID objid, params object[] args)
  {
    switch (objid)
    {
      …

        case OBJECTID.CUIFrameworkHTMLInputElement:
           return new CUIFrameworkHTMLInputElement(args[0] as IHTMLElement);
          …
        default:
           return null;
  }
}
```

Notes, CUIFrameworkHTMLInputElement implements an interface. The testlayer can cast the returned object to the interface and call the methods implemented for the interface. Likewise, all other objects that this dll creates will implement specific interfaces that the test layer can use.

UIAutomation.dll
(Shown with code snippet from the CObjectFactory class)

```
ObjectFactory]
public sealed class CObjectFactory
{
  …
  [CreateObject]
  public static object CreateObject(OBJECTID objid, params object[] args)
  {
    switch (objid)
    {
      …

       case OBJECTID.CWindowOpenedWaiter:
          if (args.Length == 0)
              return new CWindowOpenedWaiter();

          return new CWindowOpenedWaiter(args[0] as CUICondition);
         …
       default:
          return null;
  }
}
```

Notes, CWindowOpenedWaiter implements an interface, the testlayer can cast the returned object to the interface and call the methods implemented for the interface. Likewise, all other objects that this dll creates will implement specific interfaces that the test layer can use.

Other external dlls

External layers

We have currently implemented two external layers for web testing. We will discuss the IE external layer first because it is significantly less complicated than the Firefox Ajax layer. It controls IE7 using ieframe.dll and mshtml.dll (installed with IE7) to talk directly to the browser. The specifics of these dlls are found in the references section. Briefly, the dlls are used in conjunction with each other and provide the programmer with an implementation of the DOM. The Firefox layer, on the other hand, uses a brokered approach whereby the FF layer talks to a web service running on the same site as the web application being tested. The web service commands and sends data directly to and from a JavaScript frame. The JavaScript frame then controls a second frame containing the website to be tested. Both layers are discussed in detail below and Appendix B shows the interface definitions that both layers implement.

IE Layer

As indicated above, IE layer uses ieframe.dll and mshtml.dll. Since the code is implemented in C#, we created interop dlls around these native dlls.

Several selected methods of the browser object of the IE layer objects are presented below.

Method 1

```
public class CUIharnessBrowser : IUIharnessBrowser

      public CUIFrameworkBrowser()
      {
            Initialize(true);
      }

      private void Initialize(bool bVisible)
      {
          m_IEC = new InternetExplorerClass();

          //make the browser visible
          m_IEC.Visible = bVisible;

          //create a document complete handler
          m_IEC.DocumentComplete +=
            new
            DWebBrowserEvents2_DocumentCompleteEventHandler(DocumentCom
            pleteHandler);

          //create an event to signal when the document is completely
          //loaded
          m_DocLoaded = new AutoResetEvent(false);
      }
```

Here, we are creating an instance of InternetExplorerClass using ieframe.dll in the shdocvw namespace. This effectively loads Internet Explorer. We then attach an event handler for the document complete event. The document complete handler looks like,

Method 2

```
private void DocumentCompleteHandler(object pDisp, ref object
URL)

{
    if (m_IEC.ReadyState == tagREADYSTATE.READYSTATE_COMPLETE)
    {
        //set the document done event
        m_DocLoaded.Set();
    }
}
```

Method 3

Corresponding to the navigate method in the test case layer, there is an equivalent method in the external layer.

```
public bool Navigate(string URL)
{
    //dummy parameters for the IE object
    Object obj1 = null,
           obj2 = null,
           obj3 = null,
           obj4 = null;

    m_URL = URL;

    //call the IE objects navigate method with the url and dummy
      //params
    m_IEC.Navigate(m_URL, ref obj1, ref obj2, ref obj3, ref
obj4);

    //wait for the document to complete
    WaitForDocumentToComplete();

    return true;
}
```

Ajax Layer for Firefox

We chose a slightly more complex approach for the Firefox implementation. As mentioned above, this layer uses a brokered or indirect approach to automate the Firefox browser. Essentially the methods in this layer query and command a web service running on the same site being tested. The web service receives the commands and executes those commands in a frame executing JavaScript. The JavaScript then directs the commands to a second frame (in the same frameset) hosting the website being tested.

Why did we use this approach? Although Firefox provides an XPCOM interface, we did not want to rely on external components for the automation. Also, we were creating a prototype to be later extended to other AJAX enabled browsers such as Safari, Opera, etc. on windows platforms.

Why not just use this same approach for IE7? The answer is simply that the IE layer provides superior performance. This layer, though extremely effective in automating Firefox, has shown itself to be significantly slower than the IE layer implementation. Typically, our IE layer tests run two to four times faster.

Let's look at an example to understand this process. Our example will focus on a test case that clicks the search button on http://www.testwebsite.com. Note that testwebsite.com lives on a server along with our web service and the browser is hosted on a different machine altogether.

TestWebsite.com Example Test Case Code

```
BaseBrowser b = new BaseBrowser();  //causes Step 1 to occur in the
                                    //sequence list below
BaseBrowserButton button = b.ElementFromId("Button");

                  //The following statement will perform the
                  //sequence listed below.
                  //The command will be sent via webService(2)
                  //and then fed to the ajaxdriver.aspx (3)
                  //which will execute the command (4)
                  //and return the results (5)
                  //The sending and receiving of commands is
                  //coded inside the firefoxlayer and thus
                  //has no impact in the test code

button.Click()
```

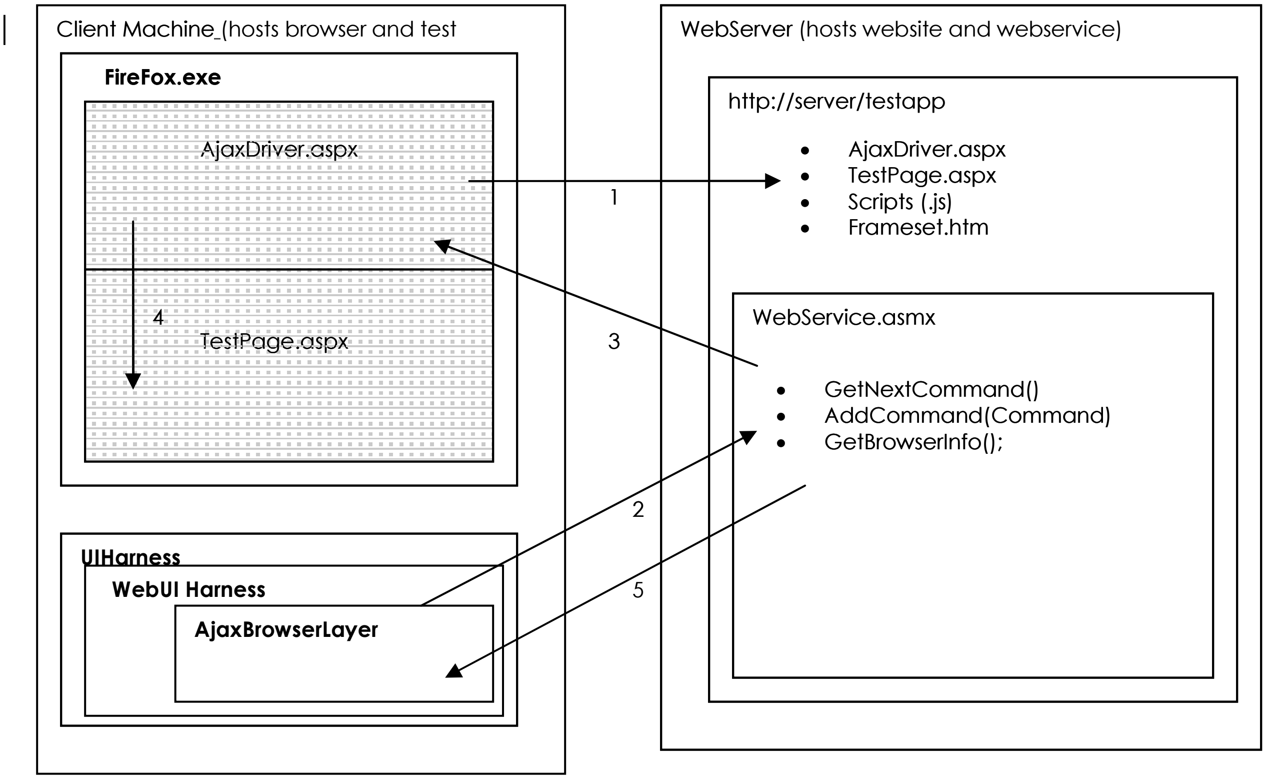
Corresponding Sequence of Actions in the Ajax Layer:

Test Machine with browser	Website and Web service Host
1. An instance of the appropriate browser (Firefox.exe) is created via	

new BaseBrowser constructor call. During initialization the browser will automatically request the main frameset from the server, which will leave it polling for commands. The main frameset will load by default http://www.TestWebsite.com in the second frame.	
2. When the test calls a button.Click() method, the call will pass through the harness and finally to the AjaxBrowserLayer which will invoke the "ExecuteCommand" function of the Web Service.	
	2. The "ClickButton" command is enqueued
	3. On the next poll slot (the webservice is sequentially polling for commands and subsequently executing those commands), AjaxDriver.aspx (part of the JavaScript engine) invokes the GetNextCommand() from the Web Service using the Ajax auto generated JavaScript proxy. It receives the command to click button (elements are identified with an HTML element id).
	4. AjaxDriver.aspx finds the appropriate object on its sibling frame (the page under test) and through its DOM clicks the button.
	5. The AjaxDriver returns the Web service's response to the AjaxBrowserLayer.

Pictorially (shown with the above step #s)

Figure 3



It is important to keep in mind here that this process is self contained in the Firefox Ajax Browser external layer. The test case code knows nothing about this entire process. It is just calling the same interface methods that it would call if the IE Layer were being used.

Examples

Example 1 – Scenario, Test Case Data, and Test Case Code

We will be navigating to an external website and then finding the text area with a rows attribute of 10 and columns attribute of 30. We will then enter the text ‘abcd’ into the text area.

Example test data comes from the xml snippet given below. Note, the test case code will use the XmlConfigurationManager object to read the xml file data.

```
    <Test Enabled="true" Name="Test1" Description="text area test">
      <Steps>

        <Step Descr="Navigate to URL" Name="Step1" >
          <Action Type="URL" Value="http://www.w3schools.com/html/showit.asp?filename=tryhtml_textarea"/>
        </Step>

        <Step Descr="Enter text and click button" Name="Step2">
          <Action Type="TextArea" Id="rows,10*cols,30" Value="abcd" />
        </Step>

      </Steps>

    </Test>
```

Example code to do this looks like:

```
//create a browser condition in order to find the element
IBaseBrowserCondition icond = ConditionFromId(strId);

//switch based on type of element
switch (strType.ToUpper())
{
    case "URL":
        Browser.Navigate(strValue);
        break;
    case "TEXTAREA":
    {
        BrowserTextArea iTA = Browser.ElementFromCond(icond) as
                                 BrowserTextArea;
        BrowserElementPropertyChangedWaiter wt = new
            BrowserElementPropertyChangedWaiter(iTA, "innerText", strValue);
                                                iTA.Text = strValue;
        wt.Wait();
        break;
     }
}

//create a browser condition from the id given in the xml file
IBaseBrowserCondition ConditionFromId(string strId)
{
    //strId will be a set of attr,value pair: attr1,val1*attr2,val2*...
    string[] strPairs = strId.Split('*');

    //split at the comma
    int iIndex = strPairs[0].IndexOf(",",
                                    StringComparison.OrdinalIgnoreCase);
    string strAttr = strPairs[0].Substring(0,iIndex);
    string strValue = strPairs[0].Substring(iIndex + 1);

    IBaseBrowserCondition retCond = new BaseBrowserCondition(strAttr,
                                                            strValue);

    //for each condition after the first one, And it with the existing
    //condition
    for (int i = 1; i < strPairs.Length; i++)
    {
        iIndex = strPairs[i].IndexOf(",",
                                     StringComparison.OrdinalIgnoreCase);
        strAttr = strPairs[i].Substring(0, iIndex);
        strValue = strPairs[i].Substring(iIndex + 1);

        retCond = retCond.AndWith(new BaseBrowserCondition(strAttr,
                                  strValue));
    }

    return retCond;
}
```

How do we actually switch between browsers?

All of our test dlls have a corresponding test xml file as shown above. There is a section in each xml file indicating the external layers. The browser under test is switched by switching the external layer assembly listed in the xml. The base class, TestCaseBase, automatically loads the desired external layer. The test developer only needs to include the xml below to select the desired browser.

To run Firefox:

```
    <ExternalLayerAssemblies>
      <ExternalLayerAssembly>.\FirefoxLayer.dll</ExternalLayerAssembly>
      <ExternalLayerAssembly>.\UIAutomationLayer.dll</ExternalLayerAssembly>
    </ExternalLayerAssemblies>
```

To run IE7:

```
<ExternalLayerAssemblies>
  <ExternalLayerAssembly>.\IELayer.dll</ExternalLayerAssembly>
  <ExternalLayerAssembly>.\UIAutomationLayer.dll</ExternalLayerAssembly>
</ExternalLayerAssemblies>
```

Future Directions and Possibilities

The web harness was developed to provide a vehicle for developing an automated regression suite on a product that does not currently make use of rich web applications by using SilverLight or AdobeFlash. In today's internet the penetration of these technologies is increasing substantially and because of that it is a feature that we would like to implement in the next version of the harness. How might we extend the harness to support plugins? In the flash case, we could develop a flash browser object in the test case layer. Then, in each of the external layers, we would implement a class that would control the corresponding flash plugin. If the plugin happens to be the same plugin on both browsers, we could simply implement a test case layer object with no underlying external layer object.

Currently the test harness is implemented to run on the Windows platform. But the concept can be applied on non-windows platforms as well.

We mentioned earlier that the harness can be extended to support other browsers or even different versions of the same browser. Let's suppose that we have the need to test on IE7 and prior versions. If the current IE layer is not compatible with prior versions, we would only need to write another external layer to support the earlier versions. If we have the need to test on other browsers such as Safari or Opera, it would just be a matter of writing new external layers.

References

For further learning, please visit the following websites:

Topic	URL
MSHTML	http://msdn.microsoft.com/en-us/library/aa741317.aspx
UIAutomation	http://msdn.microsoft.com/en-us/library/ms747327.aspx
WebBrowser control	http://msdn.microsoft.com/en-us/library/2te2y1x6.aspx
XPCOM	http://www.mozilla.org/projects/xpcom/

Conclusions

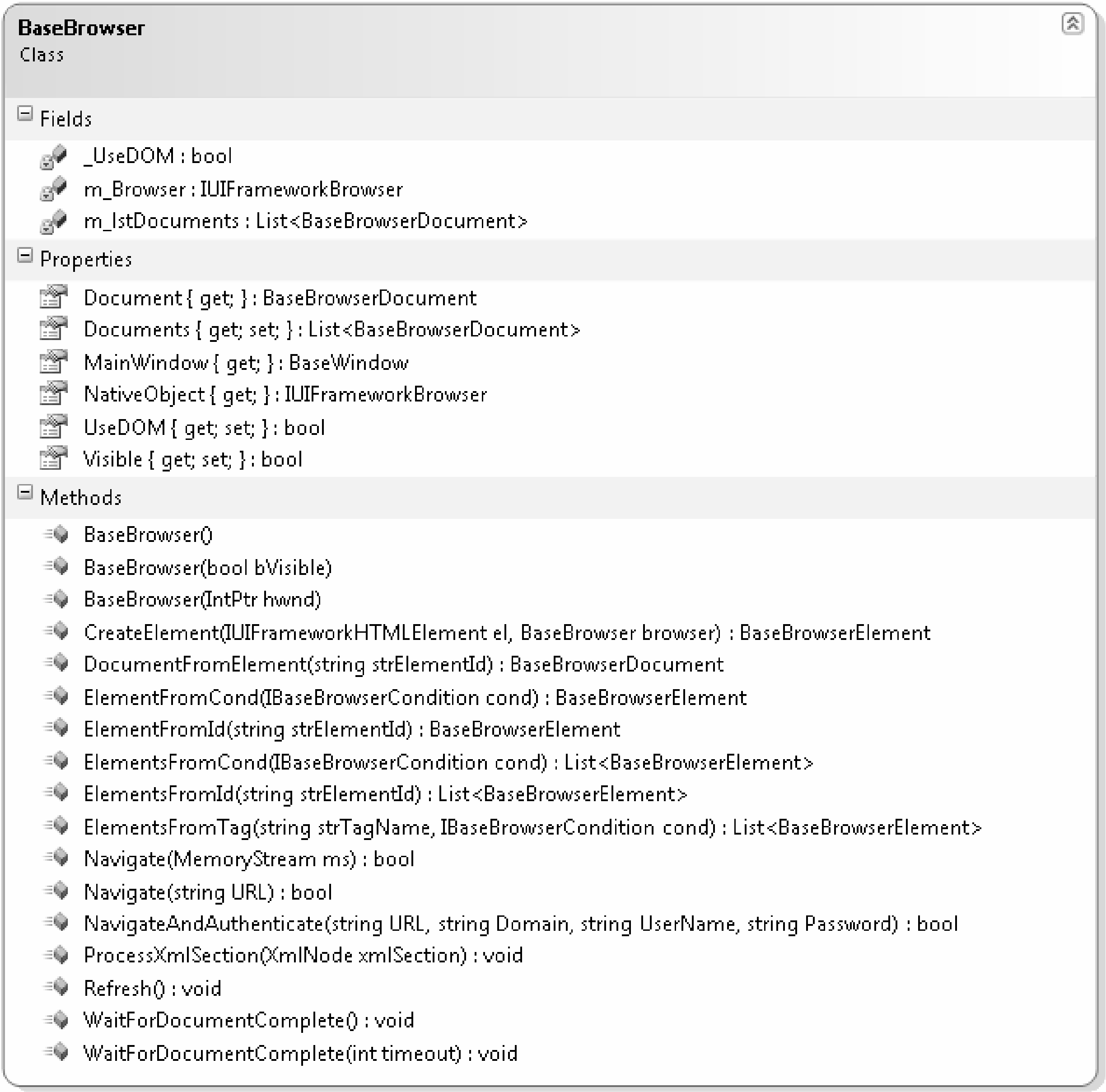
We have found that our test automation development time is now cut by more than half as the test code is browser agnostic. In fact, if we did not have this browser agnostic approach we would need to manually test Firefox. Manual testing isn't generally repeatable and we might miss regressions specific to a particular browser. The coding time is further cut down due to the object oriented nature of the harness.

We are currently using our harness to develop test automation for a storage management web UI. We will use the harness for this and future versions. Additionally, our division is developing other storage solutions that are web based and the harness will be used for automation here.

We plan on extending this harness to include support for other browser such as Safari and Opera.

Overall, the development process has been a great learning experience for us and will have a significant impact on the quality of our solutions as we can run test automation on many different browsers with minimal effort.

Appendix A – Browser Objects



BaseBrowser
Class

Fields

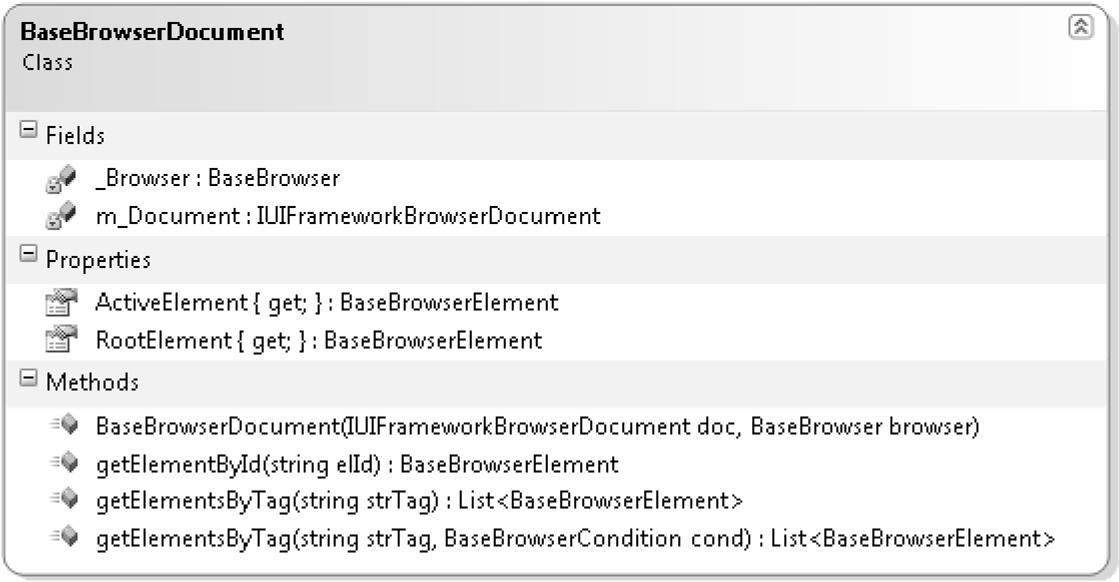
_UseDOM : bool
m_Browser : IUIFrameworkBrowser
m_lstDocuments : List<BaseBrowserDocument>

Properties

Document { get; } : BaseBrowserDocument
Documents { get; set; } : List<BaseBrowserDocument>
MainWindow { get; } : BaseWindow
NativeObject { get; } : IUIFrameworkBrowser
UseDOM { get; set; } : bool
Visible { get; set; } : bool

Methods

BaseBrowser()
BaseBrowser(bool bVisible)
BaseBrowser(IntPtr hwnd)
CreateElement(IUIFrameworkHTMLElement el, BaseBrowser browser) : BaseBrowserElement
DocumentFromElement(string strElementId) : BaseBrowserDocument
ElementFromCond(IBaseBrowserCondition cond) : BaseBrowserElement
ElementFromId(string strElementId) : BaseBrowserElement
ElementsFromCond(IBaseBrowserCondition cond) : List<BaseBrowserElement>
ElementsFromId(string strElementId) : List<BaseBrowserElement>
ElementsFromTag(string strTagName, IBaseBrowserCondition cond) : List<BaseBrowserElement>
Navigate(MemoryStream ms) : bool
Navigate(string URL) : bool
NavigateAndAuthenticate(string URL, string Domain, string UserName, string Password) : bool
ProcessXmlSection(XmlNode xmlSection) : void
Refresh() : void
WaitForDocumentComplete() : void
WaitForDocumentComplete(int timeout) : void



BaseBrowserDocument
Class

Fields

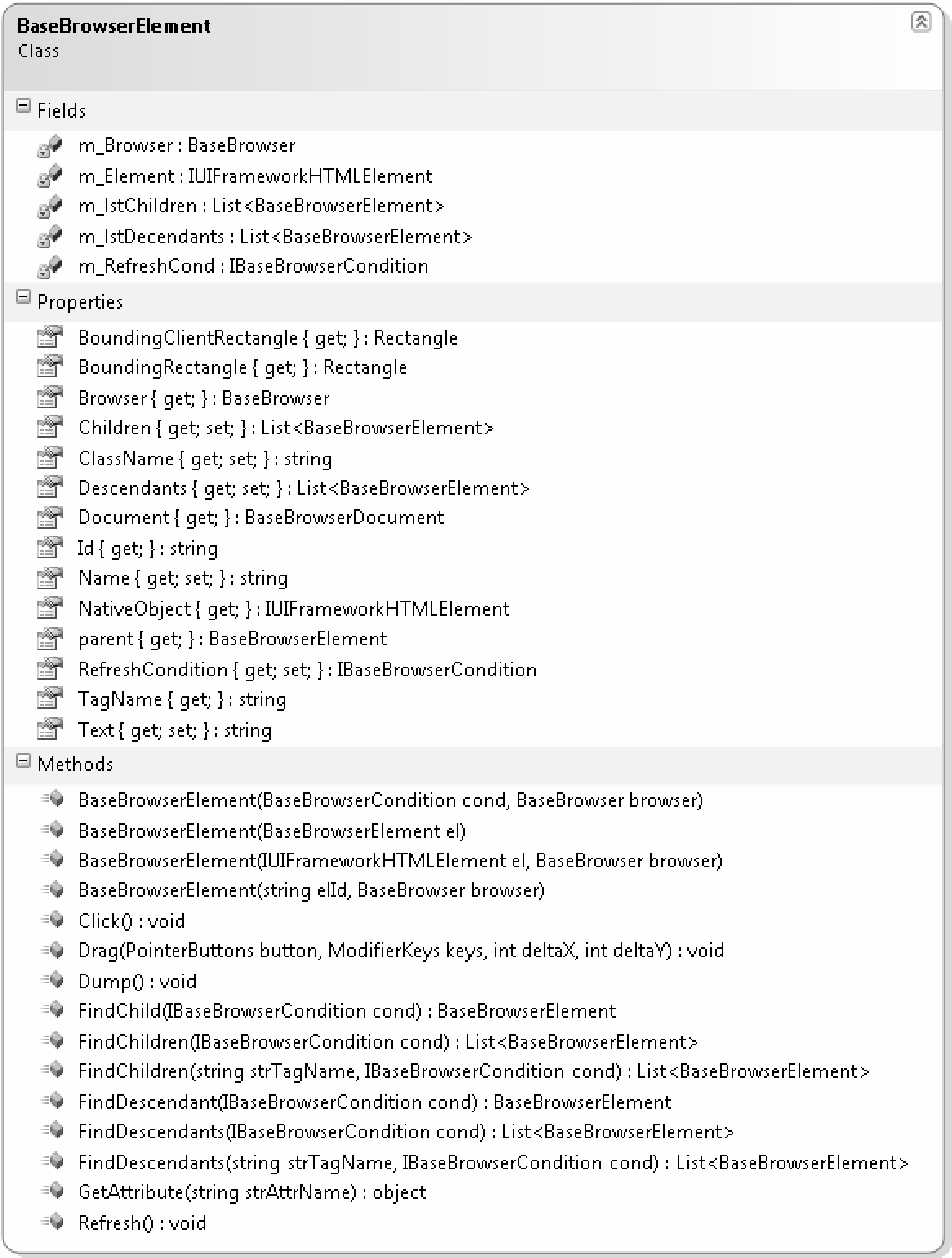
_Browser : BaseBrowser
m_Document : IUIFrameworkBrowserDocument

Properties

ActiveElement { get; } : BaseBrowserElement
RootElement { get; } : BaseBrowserElement

Methods

BaseBrowserDocument(IUIFrameworkBrowserDocument doc, BaseBrowser browser)
getElementById(string elId) : BaseBrowserElement
getElementsByTag(string strTag) : List<BaseBrowserElement>
getElementsByTag(string strTag, BaseBrowserCondition cond) : List<BaseBrowserElement>



BaseBrowserElement
Class

Fields

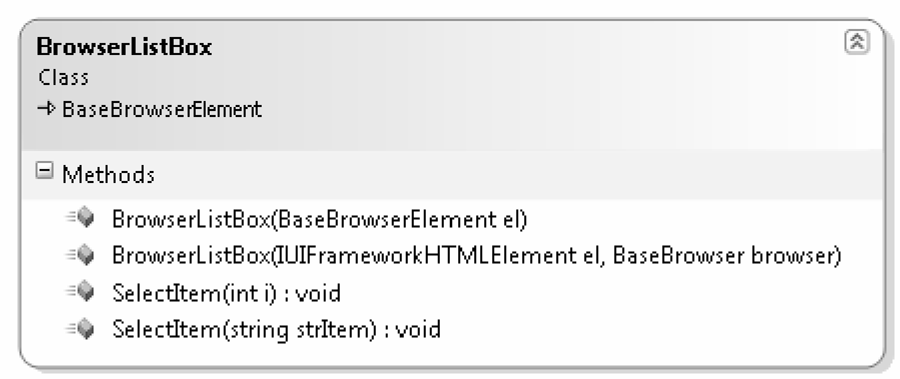
m_Browser : BaseBrowser
m_Element : IUIFrameworkHTMLElement
m_lstChildren : List<BaseBrowserElement>
m_lstDecendants : List<BaseBrowserElement>
m_RefreshCond : IBaseBrowserCondition

Properties

BoundingClientRectangle { get; } : Rectangle
BoundingRectangle { get; } : Rectangle
Browser { get; } : BaseBrowser
Children { get; set; } : List<BaseBrowserElement>
ClassName { get; set; } : string
Descendants { get; set; } : List<BaseBrowserElement>
Document { get; } : BaseBrowserDocument
Id { get; } : string
Name { get; set; } : string
NativeObject { get; } : IUIFrameworkHTMLElement
parent { get; } : BaseBrowserElement
RefreshCondition { get; set; } : IBaseBrowserCondition
TagName { get; } : string
Text { get; set; } : string

Methods

BaseBrowserElement(BaseBrowserCondition cond, BaseBrowser browser)
BaseBrowserElement(BaseBrowserElement el)
BaseBrowserElement(IUIFrameworkHTMLElement el, BaseBrowser browser)
BaseBrowserElement(string elId, BaseBrowser browser)
Click() : void
Drag(PointerButtons button, ModifierKeys keys, int deltaX, int deltaY) : void
Dump() : void
FindChild(IBaseBrowserCondition cond) : BaseBrowserElement
FindChildren(IBaseBrowserCondition cond) : List<BaseBrowserElement>
FindChildren(string strTagName, IBaseBrowserCondition cond) : List<BaseBrowserElement>
FindDescendant(IBaseBrowserCondition cond) : BaseBrowserElement
FindDescendants(IBaseBrowserCondition cond) : List<BaseBrowserElement>
FindDescendants(string strTagName, IBaseBrowserCondition cond) : List<BaseBrowserElement>
GetAttribute(string strAttrName) : object
Refresh() : void

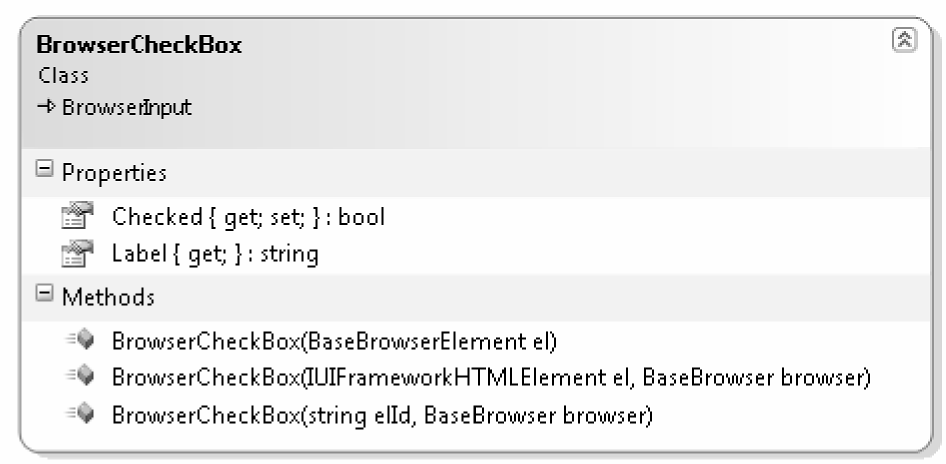


BrowserListBox
Class
→ BaseBrowserElement

Methods

- BrowserListBox(BaseBrowserElement el)
- BrowserListBox(IUIFrameworkHTMLElement el, BaseBrowser browser)
- SelectItem(int i) : void
- SelectItem(string strItem) : void

Note, we have BrowserButton, BrowserTextArea, and BrowserInput which are just renaming BaseBrowserElement.



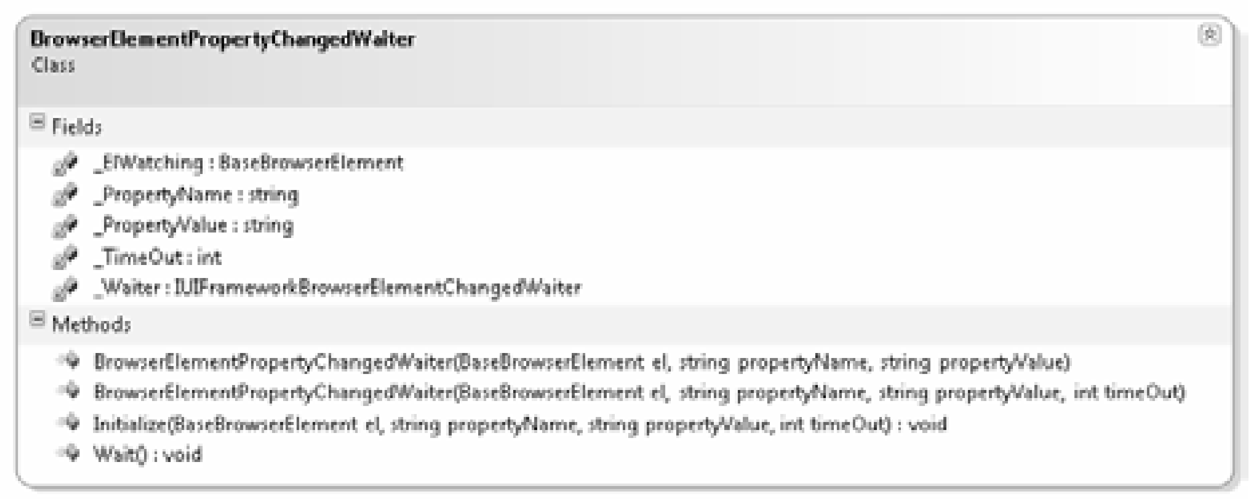
BrowserCheckBox
Class
→ BrowserInput

Properties

- Checked { get; set; } : bool
- Label { get; } : string

Methods

- BrowserCheckBox(BaseBrowserElement el)
- BrowserCheckBox(IUIFrameworkHTMLElement el, BaseBrowser browser)
- BrowserCheckBox(string elId, BaseBrowser browser)



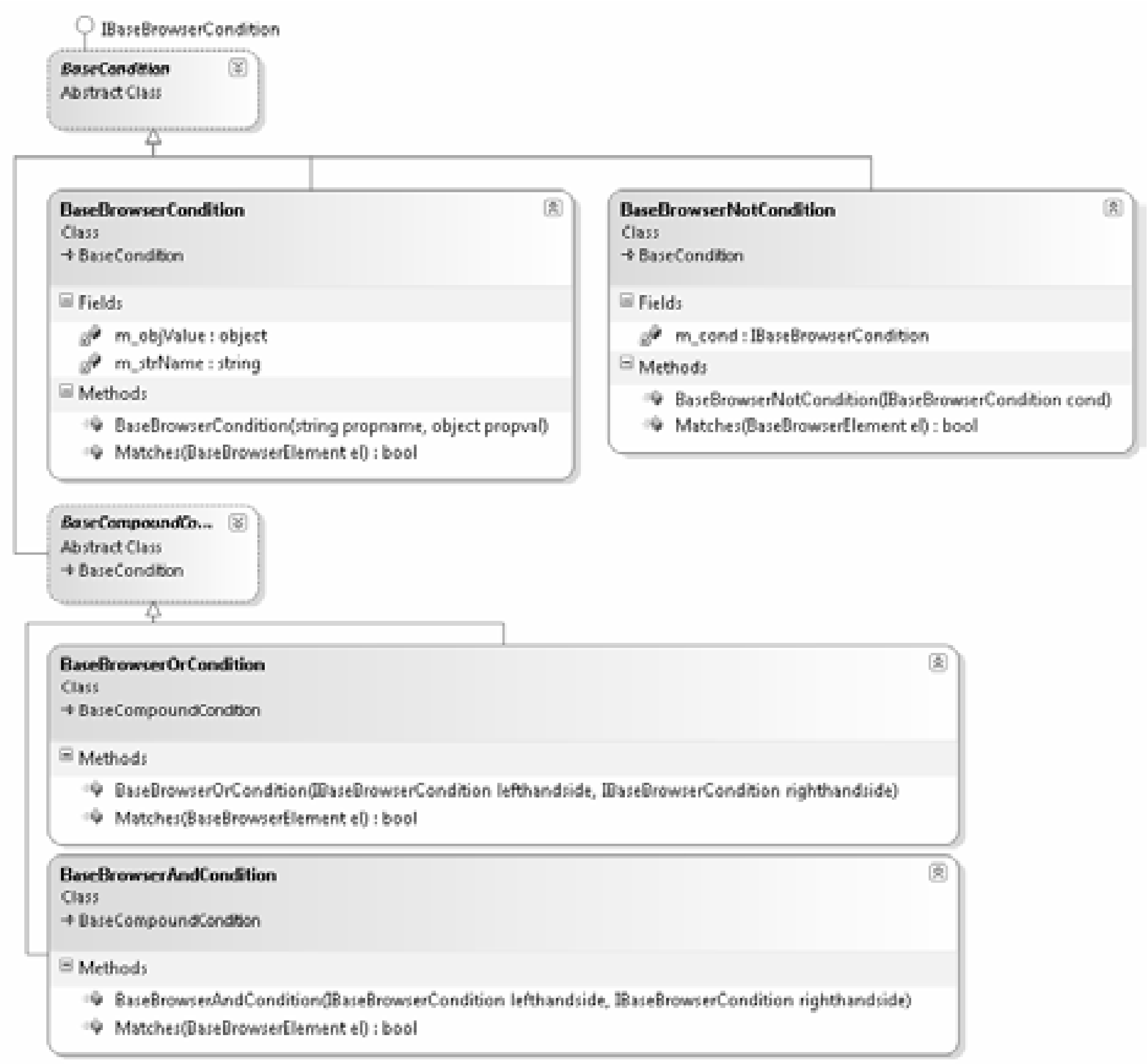
BrowserElementPropertyChangedWaiter
Class

Fields

- _ElWatching : BaseBrowserElement
- _PropertyName : string
- _PropertyValue : string
- _TimeOut : int
- _Waiter : IUIFrameworkBrowserElementChangedWaiter

Methods

- BrowserElementPropertyChangedWaiter(BaseBrowserElement el, string propertyName, string propertyValue)
- BrowserElementPropertyChangedWaiter(BaseBrowserElement el, string propertyName, string propertyValue, int timeOut)
- Initialize(BaseBrowserElement el, string propertyName, string propertyValue, int timeOut) : void
- Wait() : void


IBaseBrowserCondition
BaseCondition
Abstract Class
BaseBrowserCondition
Class
→ BaseCondition
Fields
m_objValue : object
m_strName : string
Methods
BaseBrowserCondition(string propname, object propval)
Matches(BaseBrowserElement el) : bool
BaseBrowserNotCondition
Class
→ BaseCondition
Fields
m_cond : IBaseBrowserCondition
Methods
BaseBrowserNotCondition(IBaseBrowserCondition cond)
Matches(BaseBrowserElement el) : bool
BaseCompoundCo...
Abstract Class
→ BaseCondition
BaseBrowserOrCondition
Class
→ BaseCompoundCondition
Methods
BaseBrowserOrCondition(IBaseBrowserCondition lefthandside, IBaseBrowserCondition righthandside)
Matches(BaseBrowserElement el) : bool
BaseBrowserAndCondition
Class
→ BaseCompoundCondition
Methods
BaseBrowserAndCondition(IBaseBrowserCondition lefthandside, IBaseBrowserCondition righthandside)
Matches(BaseBrowserElement el) : bool

Appendix B

```
public interface IBaseBrowserCondition
{
    IBaseBrowserCondition AndWith(IBaseBrowserCondition cond);
    IBaseBrowserCondition OrWith(IBaseBrowserCondition cond);
    IBaseBrowserCondition Negate();

    bool Matches(BaseBrowserElement el);
}

public interface IUIharnessBrowserDocument
{
    Object NativeObject { get;}
    IUIharnessHTMLElement activeElement { get;}
    List<IUIharnessHTMLElement> getElementsByTagName(string strTagName);
    IUIharnessHTMLElement getElementById(string strId);
}

public interface IUIharnessHTMLElement
{
    Object NativeObject { get;}

    void Click();

    List<IUIharnessHTMLElement> Children { get;set;}
    string TagName { get;}
    string id { get;}
    string ClassName { get; set;}
    string Name { get; set;}
    string Text { get; set; }
    IUIharnessBrowserDocument Document { get;}
    IUIharnessHTMLElement parent { get;}
    string InnerText { get; }
    string InnerHTML { get; }
    string OuterHTML { get; }
    object GetAttribute(string strAttrName);
    Rectangle BoundingRectangle {get;}
    Rectangle BoundingClientRectangle { get;}
}

public interface IUIharnessBrowserElementChangedWaiter
{
    void Wait();
}

//******* The following interfaces are place holders
public interface IUIharnessHTMLTextArea : IUIharnessHTMLElement
{
}

public interface IUIharnessHTMLInputElement : IUIharnessHTMLElement
{
}

public interface IUIharnessHTMLButtonElement : IUIharnessHTMLElement
{
}
//*****************************************************

public interface IUIharnessHTMLSelectElement : IUIharnessHTMLElement
{
    void SelectItem(string strItem);
    int SelectedItem { get;set;}
}

public interface IUIharnessBrowser
{
   bool Attach(System.IntPtr hwnd);
   Object NativeObject { get;}

    void WaitForDocumentToComplete();
    void WaitForDocumentToComplete(int timeout);

    IUIharnessBrowserDocument Document { get;}

    bool Navigate(string URL);

    bool NavigateAndAuthenticate(string URL, string domain, string UserName, string Password);

    void Refresh();

    IUIharnessHTMLElement ElementFromId(string strElementId);
    IUIharnessBrowserDocument DocumentFromElement(string strElementId);
    List<IUIharnessBrowserDocument> Documents { get;set;}

    bool Visible { get;set;}

    IWindow MainWindow { get;}
}
```

The Tao[1] of Software Defect Testing and Estimation

Scott Martin

The Regence Group

sdmarti@regence.com

James Eisenhauer

The Regence Group

jreisen@regence.com

About Scott:

Scott D. Martin currently works for The Regence Group as a Performance Analyst for the Regence Information Technology Services (RITS) division. Mr. Martin has over 16 years of multi-level collaborative experience across a broad range of industries and employers which includes two Fortune 100 companies (Raytheon and Hewlett Packard), and two large international companies (Japanese-based Kyocera and German-based Infineon Technologies).

Over his career, Mr. Martin has won repeated recognition for his work constructing a variety of forecasting and trending models, mapping process workflows and quantifying product contribution margins, improving supplier quality and enhancing corporate workforce performance through orchestrating broad behavior-based performance reforms.

Scott has earned an MBA from Portland State University, a BS in Business from the University of Oregon and a BA in Management from George Fox University.

About James:

James R. Eisenhauer currently works for The Regence Group as a Software Process and Quality Analyst. Prior to Regence, James was a Sr. Software Architect at Lockheed Martin Information Technology, and was the driving force behind a progressive software solution approach to IT Service Delivery and Governance.

During his tenure at Lockheed Martin, James was the lead consultant on many application architecture and software process engagements with major clients that included; Nike Inc, Goodyear Tire & Rubber, Symetra Financial, Department of Homeland Security, and the United Negro College Fund.

He holds a Master's of Business Administration (MBA) from the University of Tampa, Six Sigma Green Belt, ITIL Certification, ISO Auditor and a certificate of Software Engineering from the Oregon Graduate Institute School of Science and Engineering.

[1] Defined via Google search (i.e. "define: Tao") www.Summerjoy.com defines Tao as "The all that is" and www.Buddhanet.net. defines Tao as "The way".

Abstract

Webster defines the *act of estimating* as "**1**. A rough or preliminary calculation, as of work to be done. **2**. An opinion: judgment." While this is all that is defined in theory, in practice, when millions of project dollars are at stake, estimating requires the focused efforts of its end users (stakeholders) and a great many subject matter experts (SMEs).

This paper will explore the collaborative process as it unfolded for two Regence analysts tasked with building a model that estimates the completion date for the software testing cycle. Neither analyst had a clear view of the path forward, but through collaboration began to: define the problem, identify the limitations (of both the techniques employed and the data available) and to construct a defendable model that the senior leadership team could confidently present to its board of directors. The many problems encountered and limitations to implementing ideal solutions are chronicled along with the authors' experience-based recommendations for readers facing similar challenges in their own profession.

Much has been written on the subject of software development and testing by noteworthy authors. As such, it is not the authors' intent to expound or refute the validity of their work. Rather, the intent is to inform the reader of the understated complexity involved in building software testing models, to share some of the limitations we discovered are inherent in all models, and to encourage a collaborative approach among the in-house talent of their organization to produce a useful estimating model.

Copyright

1 Introduction

It was a mild summer afternoon when the rote tasks assigned to me as the new performance analyst were nearing completion. Without warning, I (Scott) was whisked away to my director's office and joined by one of her peers (a vice president). "We have a project for you," he stated as a host of curious staff groupies encircled me. "Predict with some level of accuracy how many defects will be found and corrected in our current software development and testing process and when that effort will end." I recited the project's scope for clarity, adding, "sure" and looking confident. "Work with whomever you need," he added, "and try to have something for me by Monday to review." It was a Thursday. I returned to my desk confident in my new purpose, having constructed predictive models before (both in graduate school and within the high-volume high-tech manufacturing industry), but this was software development, and already I sensed that it was different…

2 The Problem

Uninitiated in the ways of software development, my first task was to understand and define the development and testing steps involved in "manufacturing" software prior to production. Once the steps were defined, including the distributions of each testing step, estimating a completion date would be a matter of simple addition. To quantify the current performance of the division's testing process I planned to use techniques and descriptors borrowed from the high-technology manufacturing sector like Little's law[2] and flow factor[3], respectively.

As I began to familiarize myself with software development projects, I learned that no two software development projects are alike. Each project is different from the last and "defects" occur randomly with varying complexity throughout the process. In this case, the software development project began over a year ago, with some design features (known as functional specifications) entering the Integration Technical Testing (ITT) sequence 8 months later. The project employed over 100 Subject Matter Experts (SMEs), with over 40 integration and performance testers running more than 50,000 acceptance test cases. Like the software development projects before it, very little information was available on past projects that related in a useful way to the new project, but the need to coordinate project delivery with sales and government filing requirements was a constant. So there it is, the "gift" given to the new guy to solve. I was stuck between the absence of useful historical data with which to construct a testing timeline (think "rock") and the need to set target delivery dates for management (think "hard place").

Finding no software superhero cape and tights in my bag of tricks, I sought a collaborative alliance with anyone who had more knowledge than I on the topic of software development and testing. After enduring many a theoretical rant from well-meaning, but nonetheless would-be collaborators, I finally met a capable insider with a big picture view who saw the problem as a classic one. Jim Eisenhauer works as a Software Process and Quality Analyst and explained that this problem is well documented in books such as the "Mythical Man

[2] Little's Law is defined as "Dynamic Cycle Time (DCT) = Work in process (WIP) / "Go Rate" (GR)

[3] Flow factor is defined as "Cycle Time (CT) / Raw Process Time (RPT)

Month" and remains an ongoing issue among all software development and testing projects. I began reading the book, and Jim and I began a collaborative effort to construct a useful software testing defect prediction model.

3 Collaborative Evolution

My original intent required only minimal collaboration with others to define the software testing steps of the associated features, known as "functional specifications"[4] (FS), and to assign historically related times to each process step. However, the need to involve others increased significantly with the absence of useful historic data. In addition, development projects with over 1,400,000 lines of code (1,400 KLOC) and more than 50 functional specifications, such as ours, are complex and need to be analyzed in the most efficient manner possible. Jim's involvement added the researching resources and efficiency to forge a new estimation method for such a complex task.

We began our effort by inquiring if any predictive models were being used that we could adopt, leverage, or build upon. We discovered that the Application Development (AD) group used a predictive model based on the COnstructive COst MOdel (COCOMO) methodology to predict their testing duration, so we met with them to learn what we could. In reviewing their model we learned that they applied known industry averages, such as defect density per thousand (K) Lines Of Code (KLOC), and relied heavily on internal expert opinion for the model inputs.

After analyzing the results of their model, with its use of benchmark and qualitative inputs, we discerned that the completion dates and progression rate it produced were very different from what was planned and what we knew to be true respectively. For example, the AD group's model used a percentage of the known development time to estimate the testing duration. Specifically, they assigned an additional 20% of the time it took to develop the software to ITT and another 20% to user acceptance testing (UAT), but those figures produced an end date that was surpassed months ago.

Though the model was inadequate for our needs, we didn't leave empty handed. In meeting with them we observed that they were diligently measuring and recording the lines of code produced and that knowledge later proved very useful.

Next we adopted a somewhat broader view and assumed that if we couldn't construct the process steps and populate them with historic data, then perhaps we could construct a ratio of the defects discovered for each completed FS and apply it to the remaining FS to be tested.

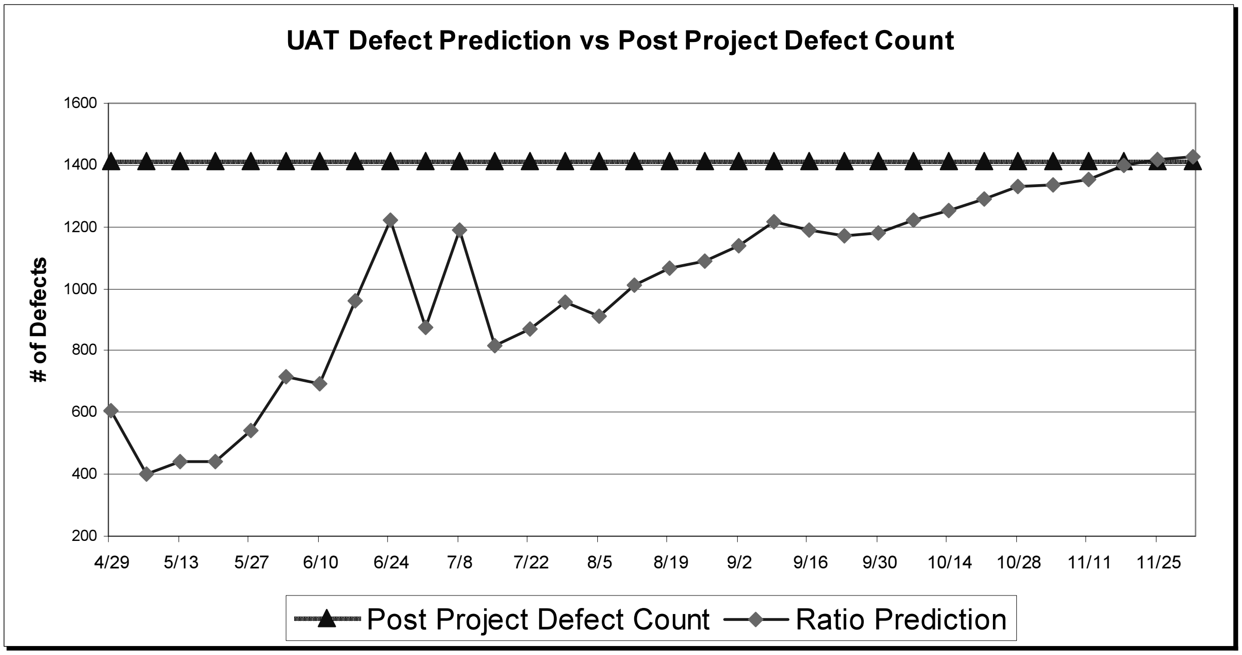
In essence we would be solving for a testing end date of "x" from a ratio of the defects found per completed FS[5]. To accomplish this we elicited the aid of the Testing Management group, which is responsible for conducting ITT and UAT testing. The testing group was invaluable in providing the testing status of each FS (because each varies in complexity) and the number of defects encountered thus far. With the data they had available we were able to apply ratios to the remaining FS (assuming the same distribution of FS complexity) to determine how many defects remained and approximately how long it would take to resolve each.

[4] Functional Specifications are a design category that describes the intended software feature to be built.

[5] Calculated as "Forecasted end date = (Number of Defects Discovered / Completed FS) * Remaining FS).

Unknown to us at the time was the fact that we had only found approximately 22% of the total defects to be discovered by project end. As such we observed that the ratios produced by this approach varied significantly due to the limited testing history (small sample size). While the ratios do attain some accuracy near the very end of the project, they are highly suspect at the beginning of the project. Post project analysis later revealed the inaccuracies of this method. (See Chart 1.)

Chart 1.



Post development testing efforts, we learned, are like FS development projects. They deviate. Often, defects found in testing make repeat trips through different parts of the testing sequence after a "fix" is implemented (very similar to "rework" in manufacturing). Knowing where each rework effort begins within each testing sequence (what point in ITT or UAT) is key to predicting future testing times, but that iterative information, like the prior development histories, was not available.

It soon became clear that piecing together a predictive model by summing the process parts or grouping similar product features (functional specifications) was also not viable because first-time build efforts lack a detailed historical reference.

We also recognized that, despite the size-based complexity of the build project, at the core we essentially had three problems associated with the testing phase that we needed to solve:

1. How many defects remain?
2. When will they be found?
3. How long will it take to resolve each defect?

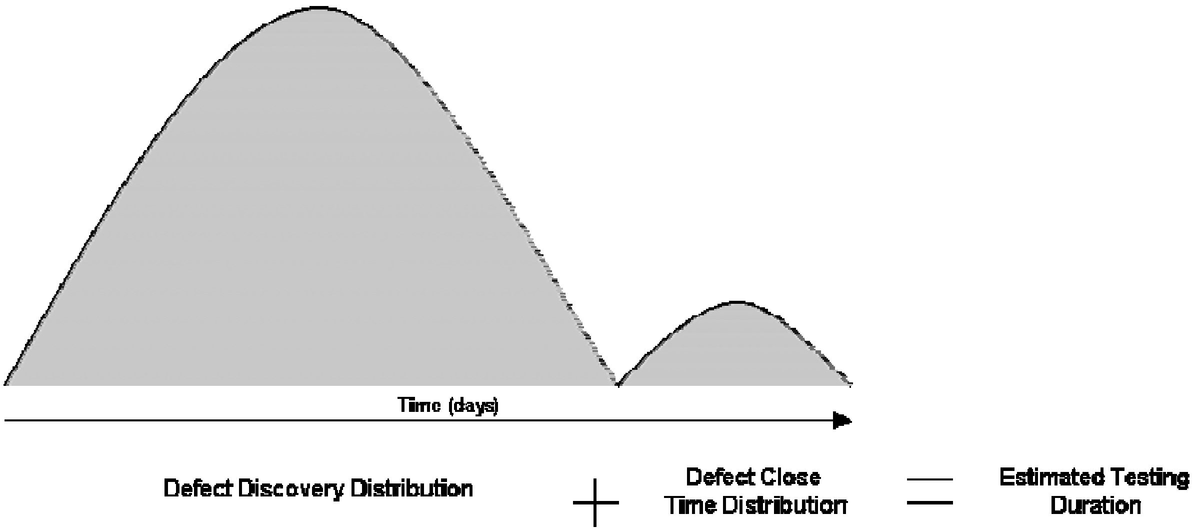
We addressed each problem in a unique way. The total number of defects was estimated using a KLOC-based ratio (the value of which we learned during our visit with the AD group). From that estimate of the total defects to be found, we subtracted the defects we had already found in order to obtain the number that likely remained.

The average defect discovery rate was an average of the sum of defects discovered each week and provided some insight into how rapidly we would find the defects we anticipated.

The average defect resolution rate was an average of the sum of defect resolutions, and was also tallied each week. This number would provide the final time period we would attach to the last known defect to estimate the testing completion date.

Although our predictive model was using a somewhat arbitrary number for the total number of anticipated defects, the discovery rate and resolution periods were based on actual performance, and we had a method. In short, adding the defect discovery period (total defects anticipated divided by the average defects discovered each week) to the average defect resolution period, will find the completion date. (See Illustration 1. below)

Illustration 1.


Time (days)
Defect Discovery Distribution
+
Defect Close Time Distribution
=
Estimated Testing Duration

Although this was a good start, the model had a number of limitations. One limitation was that the model was static. It used a fixed defect discovery and resolution rate that could not account for efficiency improvements (as the development and testing teams became more familiar with the type of defects they would discover). In other words, it assumed the same level of performance from the current level to the end of the project, when in fact there would be improvement gains along the way. In addition, for any average to be useful it must relate to the same set of variables within its population, apples to apples, oranges to oranges. In our case all testing resources needed to stay constant, with no loss of key skills or reassignment of individuals from one week to the next. We acknowledged this was not the case in practice and accepted that we could do little about it without developing a more complex econometric model, which was a time-intensive endeavor beyond what both of us could commit to with our existing work loads.

However, we could make the model more dynamic by using a polynomial equation (an algebraic equation with more than one term). After a few weeks of additional data, to give the equation more data points to work from, we tested several equations before settling on a third order polynomial (an algebraic equation with three variable terms). This equation[6] provided a reasonable inflection point rather than a flat (linear) regression trajectory for both ITT and UAT testing. (See Charts 2. and 3. below)

Chart 2.

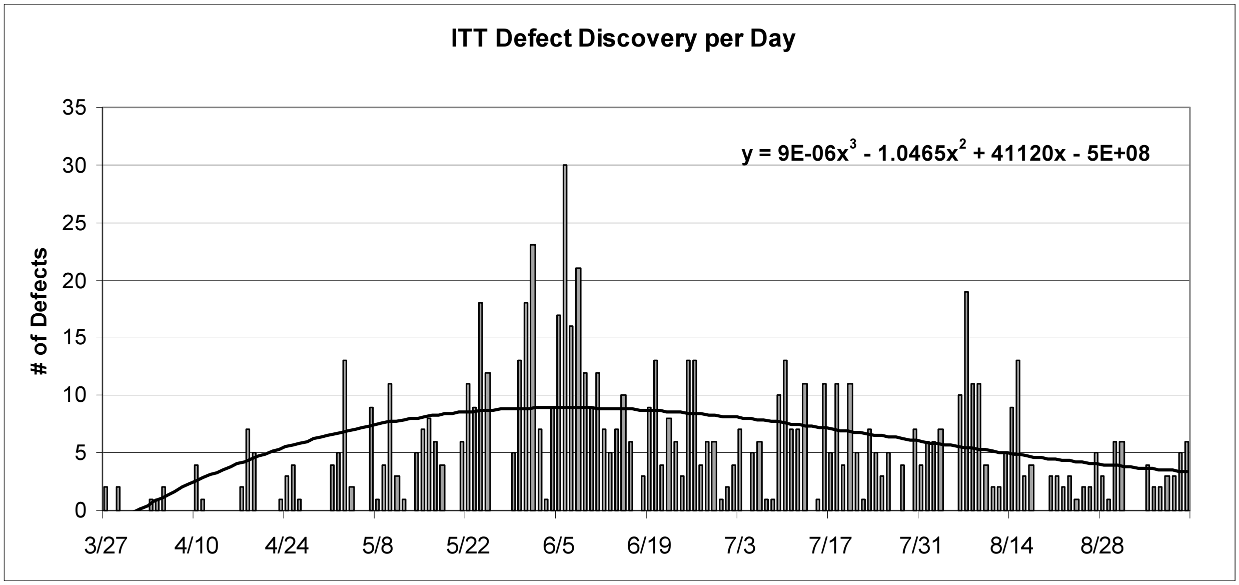
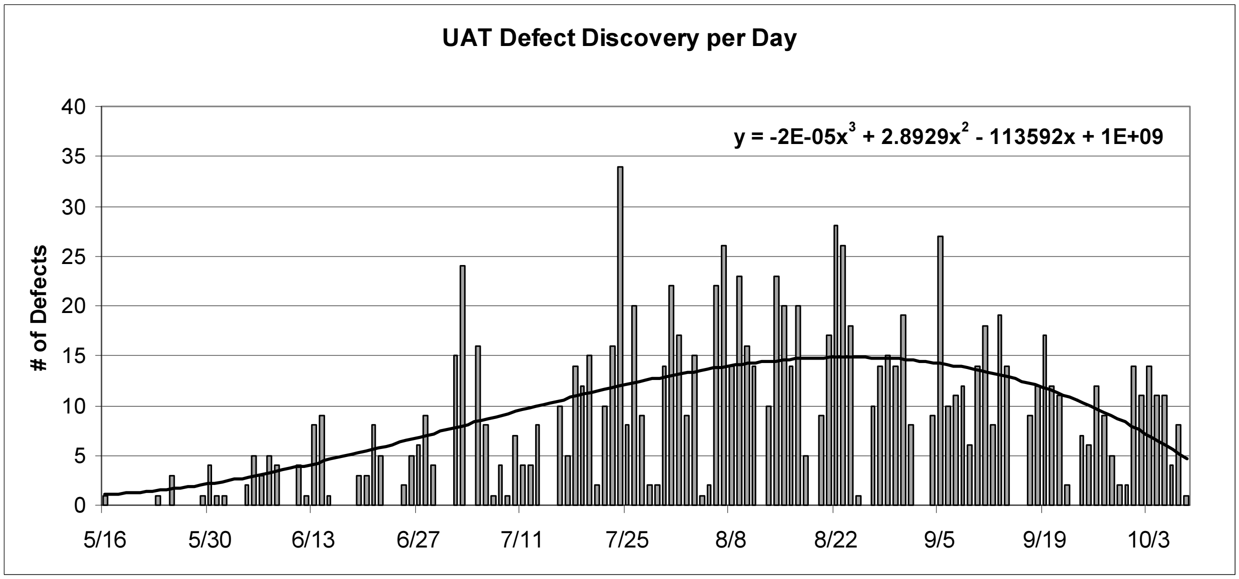


Chart 3.



[6] Obtained from MS Excel. Right-click on the data series, select "add trend line", "polynomial", scroll to 3rd order.

With the addition of the polynomial equation, the forecasted completion date extended (post inflection point), as the average weekly discovery rate declined, and the resolution period extended as more persistent defects lingered. This approach, while more accurate in theory, compounded the difficulty in obtaining a project completion date from week to week as the defect discovery and resolution rates changed. This problem could be solved by using integral calculus to find the length of the area under the polynomial curve. However, getting Excel to perform this function would be a difficult script to write. Other applications, such as VB or C++ are much better suited for this, but the time demands to pursue this effort were as complex as the econometric modeling approach.

After several weeks of additional data gathering to refine our numbers, division management pressed for results. We made several informal presentations throughout the estimation effort, but now needed to provide a defendable completion date via a user friendly tool (GUI). We reviewed our methodology, simplified the front-end tool and presented the polynomial model to management.

Management was pleased with our approach and accepted our insights given the lack of detailed historical data and both the time and resource constraints we were under in building a more responsive model. However, in the absence of a model capable of responding to changes in FS complexity, testing resources and retest scenarios, the CIO (a PhD. in mathematics) insisted that we incorporate Monte Carlo Analysis[7] (MCA) into the model. Of course! Why didn’t we think of that?!

This was a logical request given the sum of our constraints and would provide a statistical measure of accuracy to support the model’s conclusion if done correctly. As a graduate student, I used MCA to predict the outcome of various investment strategies and knew that it was well-suited for our application.

MCA essentially exposes an equation or model to random variation within a known reference distribution (most often gathered from historical data). In our case the reference population had a mean and standard deviation derived from just four months of testing. MCA then uses the reference population to modulate the results of our prediction model to define a greater range of possible outcomes.

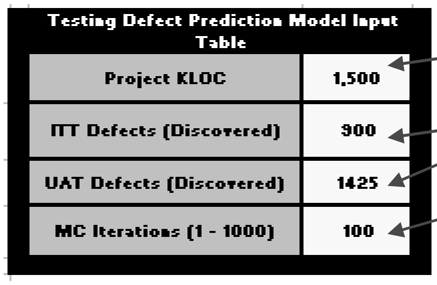
Finding a quick way to incorporate MCA into the estimating model put us in collaboration with the finance group who were actively using a commercial application of this statistical tool. What we discovered, however, was that their version required qualitatively weighted entries around anticipated outcomes (again based on internal expertise rather than statistical data we now had available). So, in the absence of a commercially viable MCA tool that we could simply “plug and play”, but with a good working knowledge of statistics, we collaborated with a talented VBA programmer and created it ourselves.

[7] Wikipedia defines Monte Carlo Analysis as “a class of computational algorithms that rely on repeated random sampling to compute their results. Monte Carlo methods are often used when simulating physical and mathematical systems. Because of their reliance on repeated computation and random or pseudo-random numbers, Monte Carlo methods are most suited to calculation by a computer. Monte Carlo methods tend to be used when it is infeasible or impossible to compute an exact result with a deterministic algorithm”

4 The Model

Mechanically speaking, the MCA portion of the model works by creating a distribution of individual, or iterative, results within the statistical possibility of the reference population. While the mean and standard deviation values of the reference populations for ITT and UAT testing are hard-coded into the model, the user must input: the KLOC value, the number of defects discovered in the testing process thus far, and the number of MCA iterations to run. Limiting the number of inputs was requested by management to encourage use of the tool and to improve output consistency. We controlled the back-end calculations. (See Table 1.)

Table 1.



Testing Defect Prediction Model Input Table	
Project KLOC	1,500
ITT Defects (Discovered)	900
UAT Defects (Discovered)	1425
MC Iterations (1 - 1000)	100

Ratios are applied to the KLOC figure to predict a total number of defects to be found for the project.

ITT and UAT defect discoveries were updated each week (these numbers are for illustration purposes only)

The model can run up to 1000 iterations of Monte Carlo Analysis, but the results change very little from running 100 iterations.

When the user selects “run” the model begins the first iteration by subtracting a random, but statistically viable, number of defect discoveries from the “Remaining Defects” column and continues to subtract a statistically viable number until the defect balance goes negative. At that point the model records the number of subtraction events (days) and begins the next iteration. (See Tables 2. and 3. below)

Table 2.

MCA of UAT Defect Testing				
Capture Date	Binary Outcome	UAT Mean (Historical)		x
		UAT Sigma (Historical)		x
		# of Monte Iterations		10
		UAT Test Comp @ 95% C.I. (Earliest)		x
		UAT Test Comp @ 95% C.I. (Latest)		x
		Days Remaining	Randomized Defect Disc. Rate	Remaining Defects
0	1	1	4.151565498	1183.3
0	1	2	8.06270953	1175.3
0	1	3	0.505096696	1174.8
…	…	…	…	…
0	1	144	8.124268956	24.6
0	1	145	17.34453569	7.3
146	0	146	16.67944873	-9.4

Table 2. This is an example of how each iteration was tabulated in a separate Excel tab away from the user interface.

The programmer enters the UAT Mean and Std Dev. Values here.

The iterations are displayed from the user interface.

The upper and lower 95% confidence intervals (an output) from the model’s previous use are displayed here for quick reference.

The “Randomized Defect Disc. Rate” column subtracts a random number of defects (within the parameters of the UAT Mean and Std Dev.) from the previous remaining defect balance.

The “Binary Outcome” column changes states when the remaining defects go negative, which triggers the program to capture the total days remaining.

This MCA iteration produced an ending date 146 days from the start of UAT Testing.

Table 3.

UAT Monte Carlo Iteration Data	
Iterations 1-1000	Defect Det. Days Remaining
99	157
100	165
101	

Table 3. This is an example of how iterations 99 and 100 were recorded (apart from the user interface). With the data now in column format, the necessary statistical calculations can be performed with ease.

When all the iterations are complete the resultant distribution mimics the statistical properties of the reference population and can be used to make statistical inferences about a completion date as long as the mean and standard deviation of the reference population is valid.

At management's request, we also kept the model's output as simple as its inputs. Three key outputs include: a project completion date, percent to forecast values for ITT and UAT and a graph of key statistical markers for both ITT and UAT completion. The completion date is the sum of the upper 95% confidence interval values for the UAT defect discovery and resolution periods (from the date UAT testing began). The percent-to-forecast values are ratios of the defects discovered to the total predicted. Finally, the graph of statistical markers for ITT and UAT provides additional insights into the resultant MCA distribution.

Predicted Completion Date

Conditionally formatted ITT and UAT % to completion windows

Iteration Countdown window(s) (only active during use)

The "Run" button

ITT and UAT graphical representation of the defects discovered to the forecast.

The ITT and UAT defect forecast windows

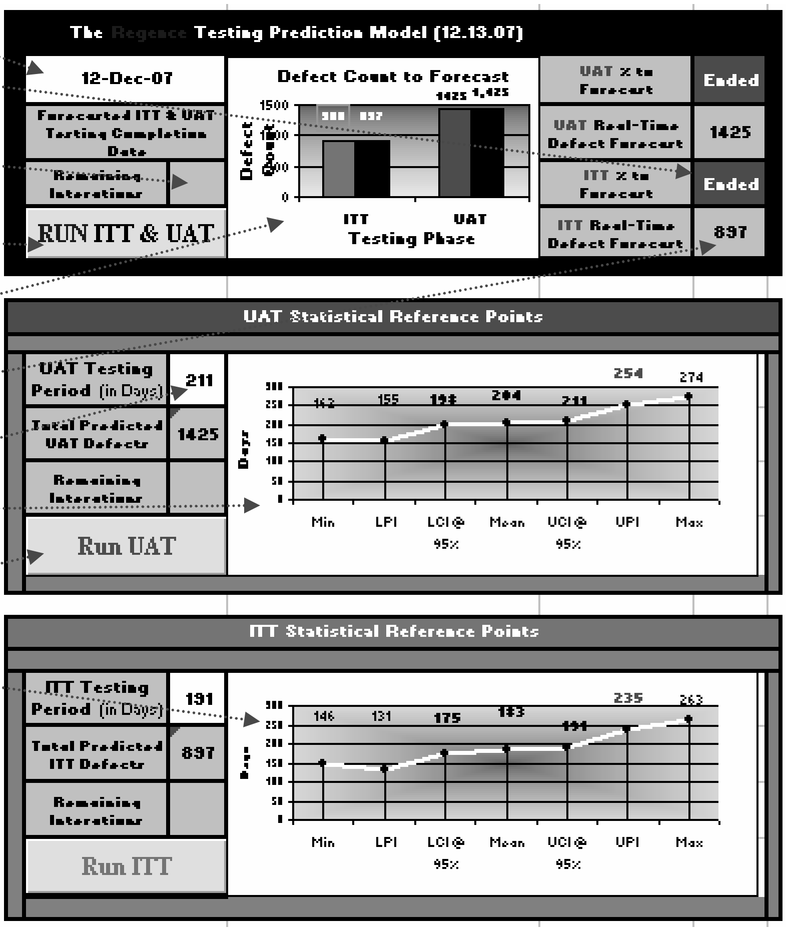
The predicted UAT testing period.

Graphs of Key statistical markers for UAT.

Independent UAT and ITT "Run" Buttons for testing.

Graphs of Key statistical markers for ITT.

Note: all the statistical calculations and MCA related tables ran on a separate tab to minimize the size of the interface and to simplify its use.



The Regence Testing Prediction Model (12.13.07)

12-Dec-07

Forecasted ITT & UAT Testing Completion Date

Remaining Interations

UAT % to Forecast: Ended

UAT Real-Time Defect Forecast: 1425

ITT % to Forecast: Ended

ITT Real-Time Defect Forecast: 897

UAT Statistical Reference Points

UAT Testing Period (in Days): 211

Total Predicted UAT Defects: 1425

Remaining Interations

ITT Statistical Reference Points

ITT Testing Period (in Days): 191

Total Predicted ITT Defects: 897

Remaining Interations

Today, with this software project complete, a review of the forecasts generated with 20% of the total defects detected (approximately when the polynomial equation was added to the model) showed reasonable accuracy. Specifically, as the model approached to within six months of the project end, it yielded 90% accuracy at times (with reduced deviation as the project neared completion). Given the skewed distribution[8] of the ITT and UAT defect discovery and resolution rates, and the "non-dynamic" methods used in the model, we were quite surprised by the overall accuracy.

Another factor that influenced accuracy was the number of total defects, calculated from a KLOC ratio each week, but this was mainly attributed to the simplicity of the approach and the lack of sufficient data to obtain a more accurate number.

Despite the model's many limitations, we were pleased with the forecasted completion date it provided. The methods outlined in this paper could be duplicated for any project, but to enhance its ability to scale to other projects would require many improvements. An econometric approach, if feasible, would allow the model to adapt to different types of software projects. A few of those project variables would include:

- Size & Skill of Testing and Development teams
- Project Risk Level
- Application, Project complexity, Domain, Programming Language
- New Development vs. Maintenance Release

However this method requires lots of data about your software development lifecycle. Designing processes and implementing systems that will enable us to capture this type of data will be our next challenge.

5 Conclusion

In completing our estimation model we collaborated with a great many talented people that tested our assertions and shaped our thinking. Throughout our journey we ran into many of the problems that have plagued software estimation for many years: size and complexity measurements, multiple levels of software development maturity, and the fact that no two software projects are the same. We did not solve any of these problems, but through our collaboration found methods to work around them. In the end, we were able to provide a tool that provided significant confidence to executive management, making way for final project budget approvals and successful system implementation.

6 Resources

(1) Norman E. Fenton, Member, IEEE Computer Society, and Martin Neil, Member, IEEE Computer Society, "A Critique of Software Defect Prediction Models", IEE Transactions on Software Engineering Sep/Oct 1999.

(2) Brad Clark, Dave Zubro, "How Good is the Software: A review of Defect Prediction Techniques", 2001 Carnegie Mellon University

(3) Frederick P. Brooks, Jr. "The Mythical Man-Month", 1975, 1995, and 1999

(4) David W. Gerbing, "Relevant Business Statistics Using Excel", 1999.

[8] See charts 2 and 3 on page 8 of this report. The skewed distribution is also thought to have influenced the LPI figures in the graphs above, producing an MCA result lower than the Min value.

Dealing With the Most Influential Factors that Cause Customer Dissatisfaction, An Organization-wide Effort in Product Crash Reduction/Elimination

Abstract

Product crash is an abnormal termination of a software program, which is usually caused by a software issue, although a hardware failure can also be the reason. When a computer "crashes," it locks up (freezes), and the user cannot obtain any response from the keyboard or mouse. Product crashes represent a relatively small percentage of problems compared to the total number of product issues (e.g., ~ 20%). However, they have a disproportionate influence on product users. When crashes occur, they are very annoying to users as they often result in the loss of data and require immediate resolution before normal operations can resume.

This paper describes the story of how an organization addressed their product crash issues. It will discuss the success factors and best practices that were used in the context of the following sequence of key activities:

- Executive commitment and leadership
- Core team and organization involvement in the implementation
- Identifying error-prone products with Failure Analysis and Root Cause Analysis
- The metrics used to verify improvement
- The progress tracking and corrective action process
- The product crash reduction process
- The results
- Lessons learned and future improvements

The first year results confirmed the effectiveness of the effort in reducing the total number of crashes in the organization's products. The organization strongly believes that continuous process improvement and consistent implementation of the product crash reduction process will enable the organization to achieve its improvement objectives in customer satisfaction and engineering productivity by minimizing the cost of rework.

The experiences and lessons learned from this story can be useful for other companies with similar needs in customer satisfaction as well as product and process quality improvement.

Bio

Dr. Duvan Luong is an Engineer Director/Architect at Cadence Design Systems. Dr. Luong's background is in the software development process and best practices with emphasis in testing. Dr. Luong's other interests are in the areas of organizational change, process improvement, customer satisfaction, effective consulting and problems solving, and operational excellence including metrics and statistical controls. Dr. Luong has had 29 years of experience in the Software Development Industry with AT&T and Bell Labs, IBM, Synopsys, Sun Microsystems, Hewlett Packard, and Cadence Design Systems.

Introduction

For most successful software companies, effectively addressing the key factors of customer dissatisfaction and the high cost of rework are the top must-have items in the company's business strategy. It is widely known that product defects have a very high correlation with the above two factors. If a company can effectively handle its product defects, it has addressed the major contributors to both its customer satisfaction and the cost of rework issues.

As customers use their purchased software products they will likely experience many product "issues." Depending on the type of issue they encounter, the customer will have a varying amount of negative experience, and hence their satisfaction for the product is adversely affected. Crashes are one type of serious product issue. One definition of a "crash" is an abnormal termination of a computer program, usually caused by a software issue, causing a computer to lock up or freeze and preventing it from responding to other parts of the system such as the keyboard or mouse. While these are a relatively small percentage of the total issues (e.g., ~ 20%) they have a disproportionate influence on customer satisfaction. When crashes occur, they are very annoying to users as they often result in the loss of data and require immediate resolution before normal operations can resume. Crashes also result in wasted effort required for re-starting customer operations and for rework by product developers to implement fixes to the crashes.

The Story

This is a story of a software development organization of several hundred members who reduced the number of current and potential crash issues in their product code base of more than 15 million lines of source code. The organization achieved its crashes reduction objectives several years ago and is currently continuing to expand their improvement efforts to include other areas of product reliability.

When the product team had a meeting with a key customer some time ago to discuss product quality issues, to the team's surprise, the customer pulled-out records showing how frequently the product crashed in their operational environment. By the end of the meeting the message was very clear – "The company tools crashed too often. When they crashed, the crashes severely impacted customer operations. The customer was not very happy with this situation". The team that developed the product believed it could do many things to improve the customer experience with respect to crashes... The team then embarked on a journey to eliminate crashes.

As the first year of the improvement program came to an end, the team was starting to see some initial results from their improvement actions: 100% Unit Testing was achieved for the latest release for the first time (previously Unit Testing was not emphasized as part of the standard development practice). All product code was analyzed with a Static Analysis tool that identified many key issues that were promptly fixed. Root-cause analysis results started to show patterns of common issues so the team could formulate lasting effective resolutions for them. Total unresolved crashes came down by 70%. Best of all, customers started to have a better operational-reliability experience with the team's products.

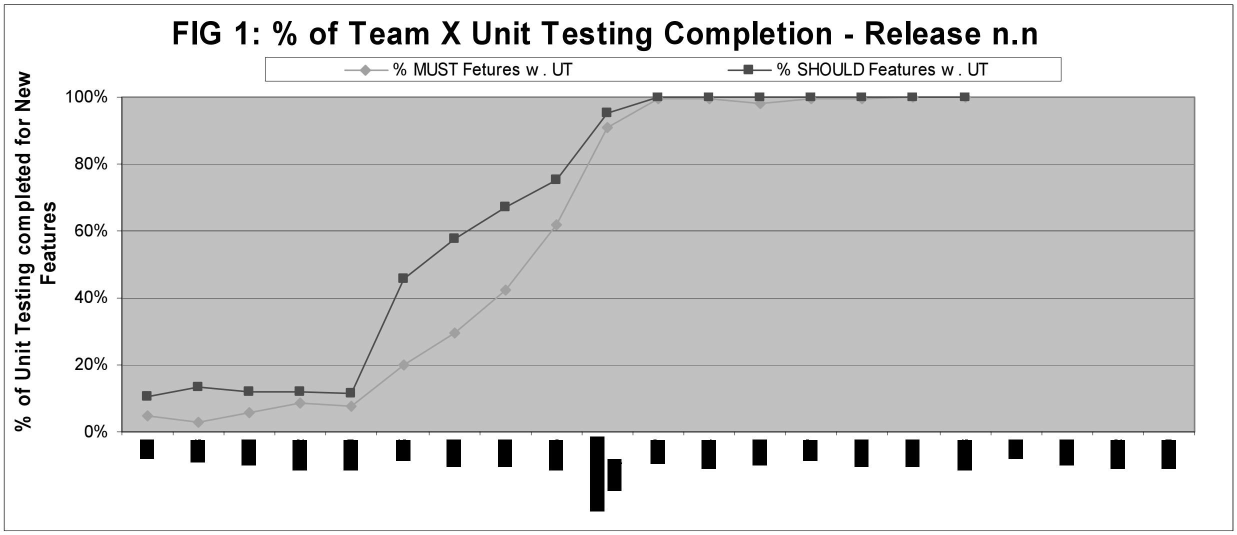


FIG1: % of Team X Unit Testing Completion - Release n.n

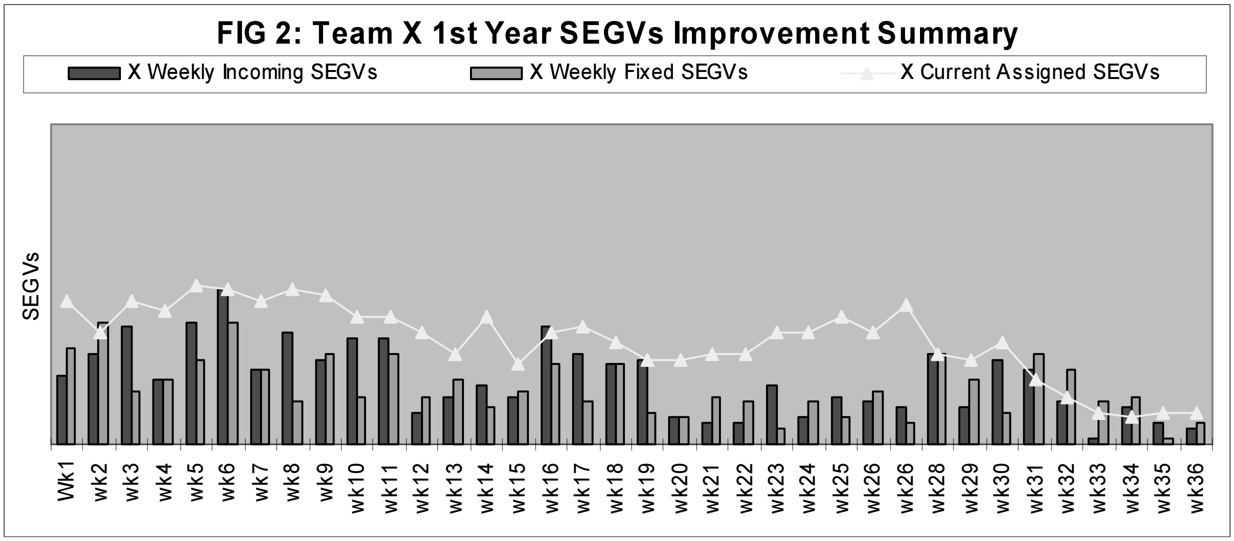


FIG 2: Team X 1st Year SEGVs (Crashes) Improvement Summary

As the team continued to make progress over the next several years the effort was shifted from crash reduction to resolving other kinds of product issues, moving from defect detection to prevention and from working on prior unresolved issues to managing new incoming issues. The team started the improvement effort with the belief that "Doing the same thing and hoping for different and better results is unrealistic." The team now believes they have a new motto to share: "Consistent focus on making improvements will produce better results." At the time this story started this particular software team was ranked at an undesirable position compared with other teams in the company in terms of product quality; however, the team gradually improved its position. As of today, it has already surpassed the aggressive quality improvement targets set by the company and is on the way to make even more improvements with its strong focus and momentum. The team has become the quality benchmark for many other teams in the company.

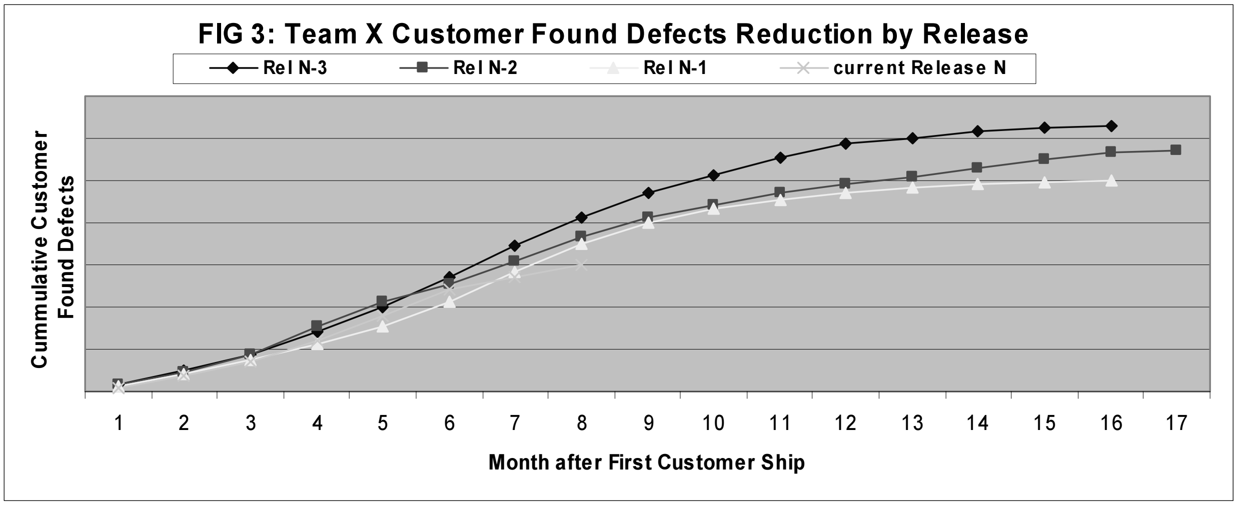


FIG 3: Team X Customer Found Defects Reduction by Release

The Success Factors

The following are the main factors that made this crash reduction effort possible:

- Leadership
- Reduction Strategy
- Involvement
- Implementing rigorous reduction actions

LEADERSHIP:

This was the most crucial factor that made the effort successful. The organization's General Manager (GM) was fully engaged in sponsoring the effort to eliminate crashes. The GM started the effort by developing and communicating the organization's vision of "Zero customer crashes" to all members of the organization. The GM identified and actively recruited the people with the right skills and enthusiasm to lead the effort. For example, a Senior Engineering Director was selected and recruited to lead the effort. The GM closely followed the progress of the crash reduction effort – giving praise and recognition to teams and individuals when significant progress was made. He raised the visibility of the effort and provided coaching and support when there was regression in performance. There were weekly discussions between the GM and his direct staff to maintain a continuous focus on the customer crash reduction effort.

REDUCTION STRATEGY:

To ensure the success of the effort, the crash reduction program was divided into two phases:

Phase 1: focused on the immediate need for reducing the current backlog of unresolved known crashes. The objectives for this phase were to bring the crash management process under control and to improve the responsiveness to crash issues.

Phase 2: focused on the longer term reduction and, if possible, the elimination of new incoming crash issues. The objectives of this phase were to identify the root causes of the crashes in order to eliminate current problems and prevent similar issues from happening in the future. An additional benefit of this effort will be as the rate of incoming crashes decreases, the team's responsiveness to crash issues and the cost of rework will also improve.

INVOLVEMENT:

Almost everyone in the organization was involved in the crash reduction effort. A taskforce was formed and named the "SEGV taskforce" (SEGV, segmentation violation, is a technical name for the most severe type of crash). This was led by a Senior Engineering Director and was chartered with the overall responsibility for the crash reduction effort. The Taskforce, in turn, worked with all the appropriate people in the Engineering group to put the crash management process under control. Engineers, Product Managers, Product Directors, Program Managers, Release Managers and Enterprise Quality Consultants were all involved.

IMPLEMENTING RIGOROUS REDUCTION ACTIONS:

For Phase 1, the focus was on setting up a project tracking and oversight infrastructure to manage crash resolutions and backlog reduction. The taskforce identified, tracked and published the information about the known crashes to all involved Engineering personnel every week. The taskforce informed the owner(s) of the code that caused the crashes and worked with them to ensure closure of the issues. Crashes that had not been resolved for more than 30 days got special attention from Senior Management.

For phase 2, the focus was on enhancing the software development lifecycle with new formal technical best practices that would reduce/eliminate new incoming crash issues. The main actions were:

- Formalize engineer testing: Each of the new/enhanced features needed to successfully complete the Unit Level Testing by the Beta test entry time. The unit test case(s) had to be documented and successfully tested. Unit Test cases and their associated test results were archived for potential integration into the regression test suite and also for later audit.

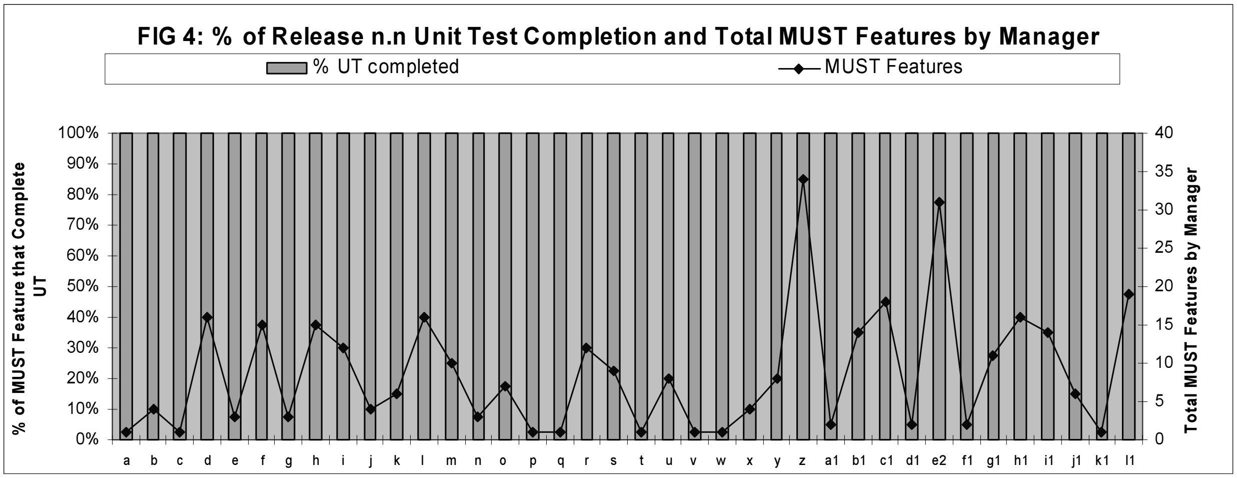


FIG 4: % of Unit Test Completion and Total MUST Features by Manager –Release nun

- Use a Static Analysis tool (QAC++ in this case) to identify potential severe issues in the code and fix them. The team worked with the QAC++ vendor Architect to identify the essential subset of the tool checking rules that identified code issues that could cause crashes and other serious code issues. This class of code issues is called severe issues. The team comprehensively implemented QAC++ in two ways. At the individual Engineer level, the tool was set-up to improve the unit code quality before it was checked-into the main code stream. At the release level, the tool was set-up to ensure compliance to the quality standards by the main code stream before it was released to customers.

- Perform Root-Cause Analysis of known crashes to identify the common problems, their common causes, how they were fixed, and what can be done to prevent these common crash issues from happening again.

- Incorporate the learning about crash prevention into future release planning so sufficient resources and time can be allocated to the necessary crash improvement activities.

The Best Practices Used

The following best practices were used by the team to eliminate potential crashes in the product:

- Identifying product crashes
- Effective use of Static Analysis tools to identify and fix potential product crashes
- Identifying the crash-prone area for resolution/reduction

IDENTIFYING PRODUCT CRASHES:

A query on the company defects database for bugs from the previous year containing “crash”, “core-dump”, and “SEGV” issues resulted in a large number of hits, of which ~20% of hits were associated with customer found problems. If we use the software industry estimate of 80 hours of effort required to find, validate, fix, and revalidate a customer found problem and about 15 hours to do same for an internally found problem, then the effort to find and fix these crash problems can account for a quite sizeable effort. This is a significant cost for the company. A similar query for just the few months before the crash reduction program started resulted in a similar proportion of hits. It appears that there wasn’t much improvement in the crash issues situation just by giving the organization more time. This was the main reason for the organization’s GM starting the whole crash reduction effort.

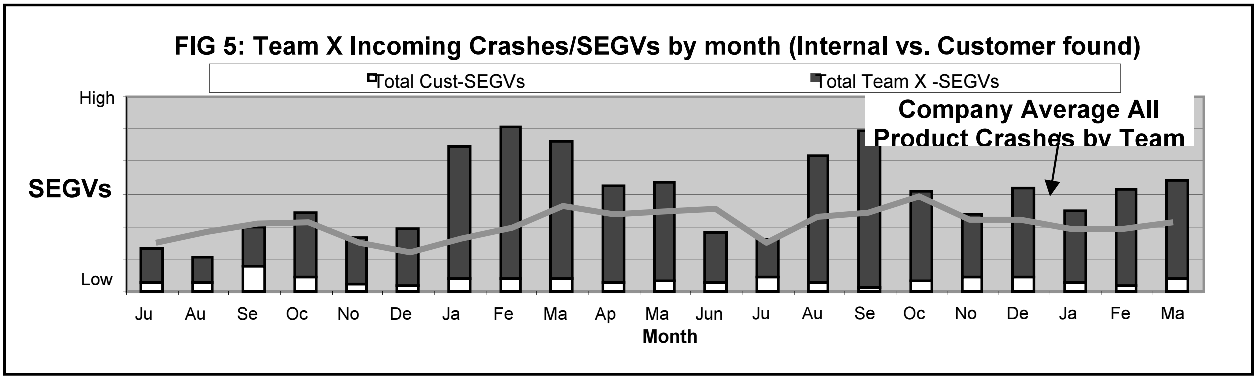


FIG 5: Team X Incoming Crashes/SEGVs by Month (Internal Versus Customer Found)

The regular (weekly) tracking of crash related metrics gave the organization the invaluable focus on resolution of the issues that eventually provided the expected improvements in crash reduction.

EFFECTIVE USE OF STATIC ANALYSIS TOOLS:

Beside product crashes, there are other serious issues that can impact customer satisfaction as well. In general, we can group these “bad” issues (crashes and other types of serious issues) together under the classification of “severe” issues. It will be fantastic if we can eliminate these severe issues from our product so we can channel the effort currently going toward fixing them into more interesting work such as new feature development. It would be even more fantastic if we can find a cheap but simple and effective way to do so. New developments in code quality analysis technology made possible the ability to find “severe” issues in the product code. Static Analysis tools are the class of application software used to analyze product code and identify potential severe issues (more than just crashes). To effectively use static analysis tool to identify severe code issues, we will need to know what the severe problems in our products are, what the impacts of those problems on product quality are and, potentially, what we can do to eliminate them.

Severe code issues can be defined as the issues related to the abnormal termination of the code execution, the corruption of the data used, the violation of memory allocation, segment violation (SEGV), and incorrect logic implementation in the products. These issues can severely impact the correct behavior of our products and cause a negative customer experience and perception about our product quality. Typical severe issues in C++ applications and their quality impact are described in Table 1.

The current crop of leading static analysis tools on the market come with a large set of capabilities that sometimes can overwhelm even experienced software developers. It is very important in planning for the use of static analysis tools to choose a small set of tool features that are specifically aimed at the discovery of the above class of severe issues. Checking for only this subset of issues by the tool can reduce the reporting of “false positives. False positives are reports from the tool of an issue that turns out to not be an issue after all. Reducing false positives helps the tool to be better accepted by the code developers by reducing the amount of output that they need to review.

IDENTIFYING THE CRASH-PRONE AREA FOR RESOLUTION/REDUCTION:

We all want to resolve every issue causing a product crash, however, in reality, we can only resolve the issues that our resources and/or business schedule allows. We must find a way to prioritize crashes to maximize return of investment (ROI) for our crash resolution efforts. There are many ways to do this. Typical prioritization approaches that can be used to identify areas for focused improvement are: crash-prone area identification, common crash-type classification, and crash escaped analysis. Figure 6 is an example of how product crashes are tracked by product. In this example, we can see products a, b, c, and d will need more attention to resolve product crashes.

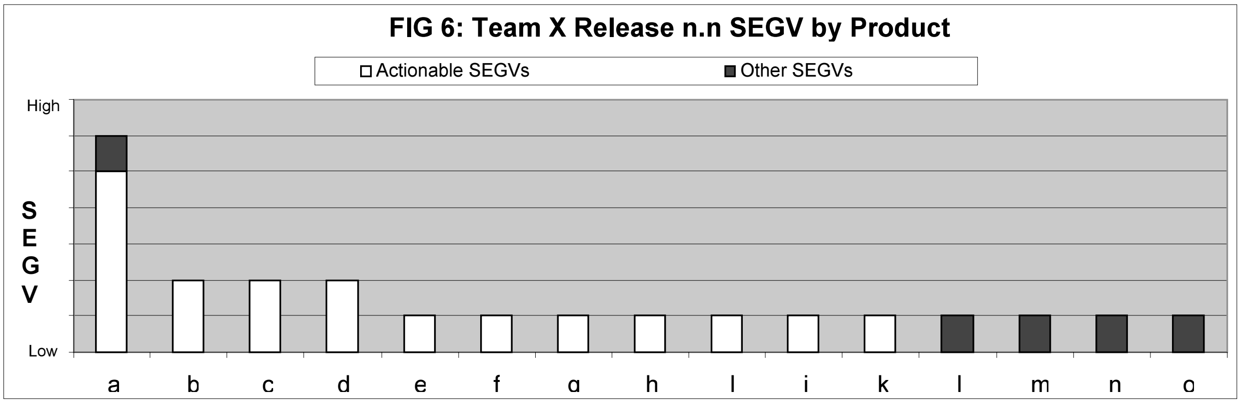


Figure 6: **Team X Release n.n SEGV by Product**

Figure 7 shows an example of crash issue classification for crash-prone product Y. In this case, the majority of the crashes fell into the error-checking category (error-checking was not built into the product code). Potential resolution for this common class of crashes is to implement the features in the product code to check for errors, such as division by zero or out-of-bound conditions, before code operations are performed.

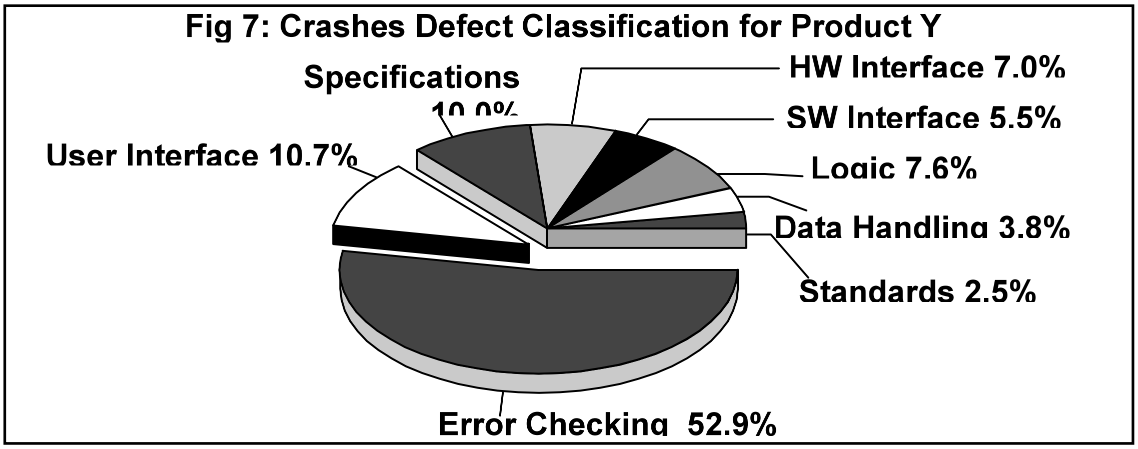


Figure 7: Crash Defect Classifications for product Y

Figures 8a through 8c are examples of crash-escaped rate analysis. Each numeric value in these charts shows the percentage of crashes found by the noted function.

In Figure 8a, a majority of crashes escaped R&D testing and were found by System testing (Operations in this case means System Testing). One potential improvement in this case is more focus on the R&D check/testing of products before turning the products over for System testing.

In Figure 8b, the case is reversed. The crashes that escaped the detection during the early phases of product development (i.e. Requirements and Architect/Design) were found by R&D. In this case, additional document and code review coverage would prevent many of the crashes that reach the R&D testing phase.

In Figure 8c, a large number of crashes escaped the internal testing effort. In this case, perhaps more flow testing or customer-like usage testing could reduce the amount of crashes that escape to customer.

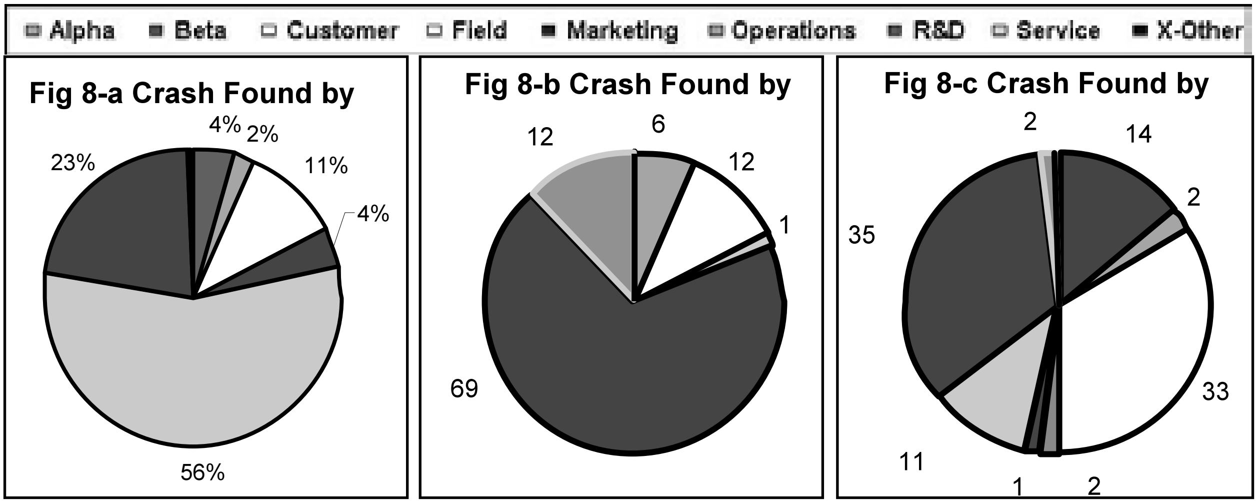


Figure 8: Examples of Crash-escaped Rate Analysis

Summary

Product crashes are annoying and can cause customer dissatisfaction and also a large amount of expensive rework. We want to eliminate and/or prevent product crashes as much as possible. This article described the process one company used to reduce the number of product crashes in an existing product line. This included: querying for product crashes among existing defect records, creating various metrics based on the identified crash reports to identify the crash-prone areas, and making the appropriate process and product improvements to prevent future crashes. Trend charts can be used to track and monitor the progress of prevention and reduction efforts.

Best practices that any team can use to ensure product crashes reduction are: leadership, participation by all members of the team, use of failure analysis and root-cause analysis to identify error areas for focused effort, and use of static analysis tools for identifying potential code issues.

TABLE 1: C++ “severe” issues

- **Abnormal Termination** (termination by C++ run-time environment)
    - Uncaught 'throw' expression in destructor -> if another exception is already being handled, and this destructor is called during stack unwinding, C++ run-time will stop - clean immediate termination
    - Throw of an exception of a type which is not listed in the exception specification -> C++ run-time will stop - clean immediate termination (for ISO C++ compliant compilers - doesn't include GCC compiler, unless -fenforce-eh-specs option is used)
    - Rethrow expression is outside a catch block -> C++ run-time will stop
    - Throw in main will cause the program to terminate -> C++ run-time will stop

- **Data Corruption**:
    - Missing required return expression -> Garbage returned from function
    - The default value is different from the overridden function -> Inconsistency between the default parameter value in base and derived classes
    - Unary minus operator is being applied to an unsigned type -> The value will remain as unsigned (has the two's-complement value)
    - An implicit conversion from an array of derived to pointer to base -> if sizeof (base) < sizeof (derived) then incorrect index to the array would be used -- because of extra member variables in the derived class). If the derived class has no new additional members then it is OK (this issue can also cause SEGV)
    - Passing a class object as an ellipsis argument -> for Ellipsis arguments (i.e. third argument to printf function) the value is copied bit-wise on the stack, and no type conversions are applied (this issue can also cause SEGV)
    - Builtin object or argument is modified and accessed between sequence points -> the order of evaluation is un-specified for operands in an expression. It is not guaranteed to be left to right so the compiler can optimize the code
    - The right hand side operand of the shift operator is negative or is too large -> undefined behavior, meaning the resulting value cannot be pre-determined
    - A jump past initialization of objects -> use of un-initialized data
    - Function has a non-void return type but no return statements -> Garbage returned from function
    - Uninitialized member used as initialiser -> use of un-initialized data

- **Memory violation** (cause memory leaks):
    - Object used with inconsistent allocation methods in different translation units -> Link-time issue - inconsistent use of plain and array allocation of the same object in the project
    - There is no corresponding operator delete for this operator new -> Use of custom 'operator new' with default 'delete'
    - This object was used as pointer to non-array object earlier -> inconsistent use of plain and array allocation in the same source file
    - This object was used as pointer to array earlier -> inconsistent use of plain and array allocation in the same source file (reverse of previous case)
    - This object was used as a pointer to old C-style allocated memory -> inconsistent use of C and C++ allocation in the same source file (malloc and new) -- malloc doesn't call constructor and free doesn't call destructor unlike new and delete

- **SEGV** (segmentation violation - termination by the OS)
    - The memory referred to has been deallocated -> termination by the OS
    - Index is out of bounds -> termination by the OS

- **Wrong Logic implementation**:
    - The operand of 'sizeof' has side effects that will not be performed at run-time -> operand of sizeof is evaluated at compile-time, so any side effect will not be performed, i.e. a function call or modification to variable

    - Exception is derived from an exception caught above -> this catch-clause will never be executed - dead code
    - This catch-all exception handler should be the last -> the following catch-clause will never be executed - dead code.

References

Review and Inspection

- Michael Fagan - Inspections - http://www-isd.fnal.gov/ftr/www/faganInsp.html
- Tom Gilb - Software Inspection**,** Addison-Wesley Longman Publishing Co., 1993
- Karl E. Wiegers - Peer Reviews in Software: A Practical Guide*,* Addison-Wesley, 2002

Continuous improvement, Metrics, Root-cause-analysis:

- Watts S Humphrey:
    - Managing the software process, Addison-Wesley Longman Publishing Co, 1989
    - Introduction to the personal software process, Addison-Wesley Longman Publishing Co, 1997

- Robert R. Grady:
    - Software Metrics: Establishing a Company-Wide Program, Englewood Cliffs, NJ: Prentice Hall, 1987
    - Practical Software Metrics For Project Management And Process Improvement, Prentice Hall, 1992
    - Successful Software Process Improvement, Prentice-Hall, Inc. 1997

Learning Organization

- Peter Senge - The Fifth Discipline: the Art and Practice of the Learning Organization – Currency, 1990

Static Analysis tools:

- http://weblog.infoworld.com/article/06/01/26/73919_05FEcode_3.html (Between klockwork and coverity)
- http://www.ep.liu.se/ea/trcis/2008/003/trcis08003.pdf (All the tools)
- http://weblogs.java.net/blog/cos/archive/2006/10/static_analyzer.html (klockwork and coverity)
- http://pubs.drdc.gc.ca/PDFS/unc69/p528977.pdf (Almost all the static analyzer tools available. Has some info about Polyspace too.)

Ensuring Software Quality for Large Maintenance Releases

Jean Hartmann
Test Architect
Developer Division Engineering

Microsoft Corp.
Redmond, WA 98052

Tel: 425 705 9062
Email: jeanhar@microsoft.com

Abstract

Like many divisions within Microsoft, Developer Division with its two flagship products - Visual Studio and Visual Studio Team System - allocates a significant portion of its time and resources to ensure the highest possible quality of its maintenance releases. Product teams within the division face the challenge of having to execute increasing numbers of automated tests against a growing code base in ever shorter development test cycles.

In a previous PNSQC paper[1], we described an approach that focused on leveraging so-called selective revalidation techniques using code coverage data, which provided a significant reduction in the number of regression tests that needed to be rerun for a given code change and better prioritized test suites. Thus, the focus of this paper is to describe and discuss how we deployed and evaluated these techniques as part of a recent maintenance test cycle.

Our guiding principles for a successful deployment/evaluation of this technology were:

∉ **Redefine the test exit criteria** to force a change in *test focus* from validating the entire product to validating only the modified portions of that product (aka code churn)

∉ **Enhance the test process and tool infrastructure** to support the collection, analysis and reporting of relevant data, so that test teams could gauge progress and declare success respective to those new test exit criteria

∉ **Enable testers to gain a deeper understanding of the code changes** and their impact on the product as well as requiring them to better quantify the (re)testing effort

This paper outlines the testing process associated with our maintenance releases. We reflect on some of the challenges that we faced in enhancing this process and tool infrastructure. Finally, to demonstrate the potential of these selective revalidation techniques, we use examples and data from this recent maintenance test cycle.

[1] J. Hartmann, "*Applying Selective Revalidation Techniques at Microsoft*", PNSQC 2007, pp. 255-65.

1. Introduction

Maintenance work on Microsoft products is characterized by scope and frequency. Within a division this work is either conducted by the product team during new product development work or by a dedicated Servicing Team that focuses on maintenance. For example, a team working on a smaller product, such as the C# compiler, would conduct its own maintenance, while the Windows organization would have a dedicated servicing team for maintenance for operating systems such as Windows XP/Server 2003.

Maintenance covers a wide range of activities in terms of required code changes and testing effort. In this paper, the emphasis is on large-scale maintenance releases, known as Service Packs (SPs), where the changes are much more substantial and less frequent in nature compared with smaller-scale patching efforts. These releases require collaboration and coordination from many teams within an organization. They touch many features in different ways with resulting code changes ranging from new binaries being added to existing code being re-factored.

The nature of such large-scale maintenance releases has the potential for introducing not only bugs within the individual Visual Studio products, for example, the various language compilers, but more importantly bugs resulting from the integration of these individual products into the overall Visual Studio product due to numerous dependencies. Thus, the motivation and goal for improving our regression testing strategy is to thoroughly exercise the individual products as well as eliminate system integration issues without having to rerun all our test suites repeatedly.

In a previous paper [1], a methodology for performing selective revalidation was discussed. In that paper, we focused on how to effectively and efficiently select a subset of regression tests based only on the code that has changed or churned. Here, we explain how this methodology benefited large maintenance releases within Developer Division.

2. Selective Revalidation Techniques – Recap

While selective revalidation techniques were discussed in detail in a previous paper [1], it is important to briefly recap the key facts concerning these techniques and highlight those aspects that contribute to the discussion in this paper.

Selective revalidation techniques are a class of testing techniques that guide testers in the selection of suitable subsets of regression tests or assist in prioritizing tests in response to code changes or churn. The techniques described in the previous paper and applied here use code coverage data which was collected at a previous product release milestone, plus linear programming algorithms known as 'greedy algorithms' [2], to determine a minimal set of tests to rerun for the upcoming maintenance milestone.

The techniques address two issues, both with the same end goal of streamlining the regression testing process. One technique is known as *regression test selection*, the other as *test set optimization (or test prioritization)*. The key distinction is that the former utilizes code churn data in addition to code coverage data to determine a minimal solution that focuses on tests traversing modified code. The latter simply identifies a minimal and thus prioritized set of tests from the test suite without regard for where code changes have occurred.

The previous paper [1] also describes the various scenarios in which these selective revalidation techniques could be used. This paper is focusing on *post-verification testing*. In post-verification testing, code changes have already been checked-in and tested by the individual product teams. Now test passes are conducted on a larger scale to validate overall product functionality and in this case, to specifically satisfy a given exit criterion - code coverage – for the SP maintenance release.

3. Enhance the Test Process

While the previous paper [1] focused on describing the test methodology itself, this paper intends to focus on the next steps that we are taking to leverage the methodology and reap the benefits of resulting test efficiencies and effectiveness.

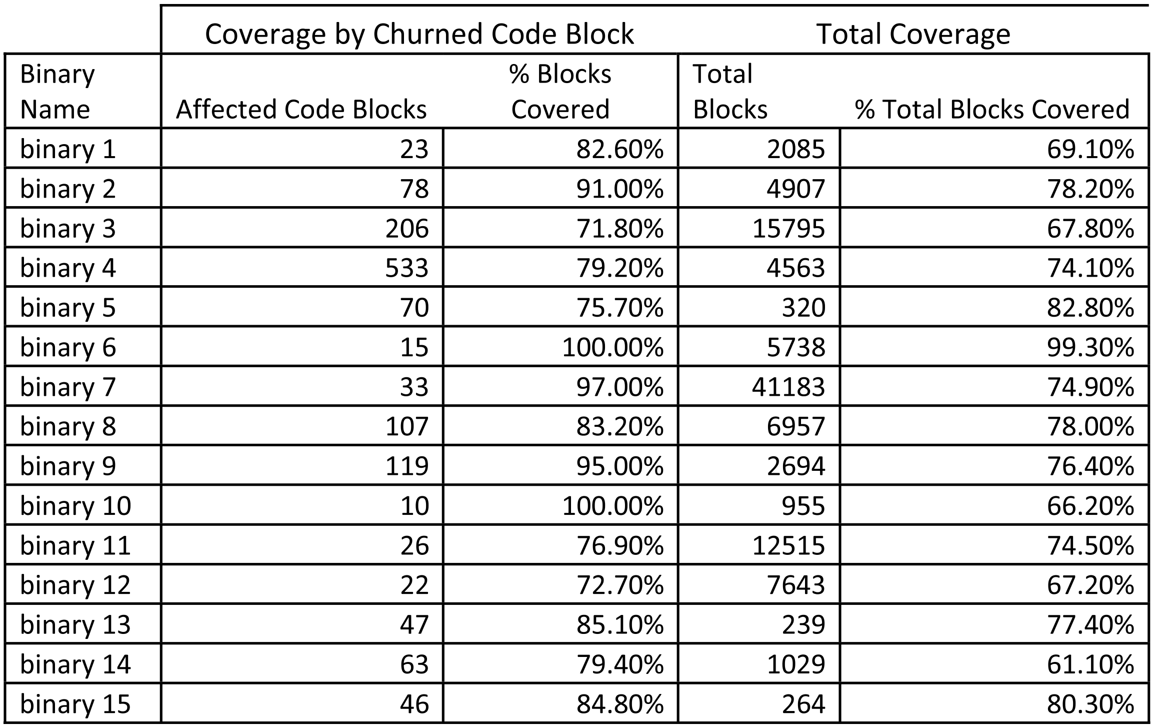
3.1 Redefine the Exit Criterion

Maintenance releases are subject to the same set of exit criteria as major product releases, which requires teams to satisfy a code coverage criterion by reaching 70% block coverage[2] for each executable that is part of the product[3]. This quality bar applies to both existing and new executables for a maintenance release. In the past, such a criterion led to adequate testing of the new code (executables) being introduced, but did not necessarily ensure that code modified in the existing code was adequately retested, if that code had reached the 70% bar for the previous product release. Regression suites were being rerun in their entirety with inadequate attention being paid to how many of those tests actually traversed the modified code segments. There was no means of assessing test thoroughness.

Therefore, the first enhancement that we made to the existing test process was to redefine this exit criterion subtly by stating that the quality bar was now to be applied to the *coverage of churned code*. This shift in test focus has the benefit that testers were forced to examine and better understand the code changes as well as quantify and plan the test effort required for those changes. As a result, testing a change was not simply a binary statement of whether or not the change had been tested, but a quantifiable measure of how thoroughly the change had been tested – a big step forward.

[2] A basic block is a sequence of binary instructions without any jumps or jump targets. In other words, a basic block always (in the absence of exceptions) executes as one atomic unit (if it is entered at all). Because several source lines can be in the same basic block, for efficiency reasons at execution time it makes more sense to keep track of basic blocks rather than individual lines

[3] This coverage adequacy criterion and its specific value of 70% is a corporate-wide standard.



	Coverage by Churned Code Block		Total Coverage	
Binary Name	Affected Code Blocks	% Blocks Covered	Total Blocks	% Total Blocks Covered
binary 1	23	82.60%	2085	69.10%
binary 2	78	91.00%	4907	78.20%
binary 3	206	71.80%	15795	67.80%
binary 4	533	79.20%	4563	74.10%
binary 5	70	75.70%	320	82.80%
binary 6	15	100.00%	5738	99.30%
binary 7	33	97.00%	41183	74.90%
binary 8	107	83.20%	6957	78.00%
binary 9	119	95.00%	2694	76.40%
binary 10	10	100.00%	955	66.20%
binary 11	26	76.90%	12515	74.50%
binary 12	22	72.70%	7643	67.20%
binary 13	47	85.10%	239	77.40%
binary 14	63	79.40%	1029	61.10%
binary 15	46	84.80%	264	80.30%

Figure 1: Code Coverage Statistics for Selected Binaries

Figure 1 above shows the coverage metrics collected by a Developer Division team as they adopted this redefined criterion. Under the old coverage criterion of 70% block coverage per binary, several binaries, such as binaries 10 and 14, would have required additional test effort to meet the quality bar. This would have resulted in additional tests being created without knowledge of whether those tests actually traversed the churned code and ensured the quality of that churned code. By applying the new criterion to those same binaries, we can clearly see that the changes have not only met the quality bar, but have exceeded it. Availability of such data enables test managers to better monitor test progress and its completion with respect to code changes - they are no longer simply relying on the assurances of their staff.

3.2 Improve Tool Infrastructure

Due to the scale of our current testing processes, adoption of new test processes and technology is only viable when supported by reliable tools. This support must be defined and deployed in such way that it has minimum impact on the existing test infrastructure and effectively complements it. Figure 2 illustrates the new and existing tools and their flow; their usage will be explained in more detail in the next section.

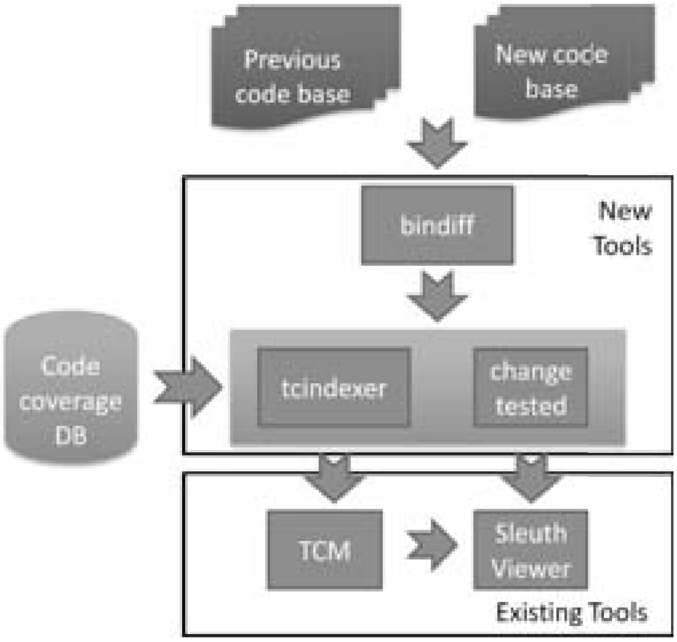


Figure 2: Tools and Flow

In prior testing cycles, the environment consisted of the code coverage database, the *Test Case Management (TCM) tool* and the *Sleuth Code Coverage Viewer* tool. In order to measure code coverage during the maintenance test pass, the code base was compiled and the resulting binaries instrumented for block coverage. The tests stored in the TCM were executed and the testers ensured that they had met the quality bar by viewing the resulting coverage data in the Sleuth code coverage viewer tool. These processes leveraged the Magellan toolset [3], a suite of tools for instrumenting compiled code binaries and then collecting, analyzing and viewing the resulting code coverage data.

The new tool flow adds a binary differencing (*bindiff*) tool that compares each binary in the previous (baseline) and current product builds and generates an XML file detailing the differences between builds on code block basis. This file is then consumed by two tools, *TCIndexer* and *ChangeTested*, together with code coverage data associated with the previous and current builds, respectively. This tool is also part of the Magellan toolset.

TCIndexer, the selective revalidation tool, leverages the code churn data for the two builds and the code coverage data from the baseline build to compute a regression subset or prioritized test set that can be rerun to exercise the current build and collect new code coverage data. *TCIndexer's* XML-based results are delivered with each new instrumented product build during a test pass and can be consumed and filtered by the various test teams for their relevant test cases as the solution contains tests (contributions) from other test teams. Its output file can also be consumed by the TCM tool, so that the process of rerunning could be fully automated, if the teams wanted to do so. *TCindexer* is built on top of yet another tool in the Magellan toolset.

While *TCIndexer* provides guidance for test selection, the *ChangeTested* tool can be regarded as feedback for the teams on how much testing remains before the churned code has been adequately tested. Again, the tool leverages the code churn data for the two builds and code coverage data. However, this coverage data is reflected in a newly created and then forward-merged coverage database that coincides with the start of the maintenance test pass. Metaphorically speaking, you are at base camp respective coverage of the churned code, looking up the mountain that you need to conquer and getting feedback with each new product build (step) as to how you are progressing with testing the churn.

The output files produced by *ChangeTested* include a coverage report for the churned code, supplemented with details of the churn. In fact, the table shown in Figure 1 is derived from such a coverage report. These reports are not only consumed by the individual teams, but are rolled up to an organization-wide website displaying all exit criteria relevant to the maintenance release. Figure 4 shows the code coverage reporting page. The other file being output is a *Sleuth* coverage viewer filter that testers can use, whenever they view the coverage data superimposed on the code. The filter file focuses and guides testers as it directs the viewer to display only those code functions in which code blocks have churned. This helps testers to more quickly understand the churned code, its impact and additional testing that may be required.

4. Testing a Service Pack Release

Figure 3 outlines the enhanced regression testing process flow for a recent service pack release based on the redefined exit criterion and improved tool infrastructure described above. A step-by-step explanation of the new process is given.

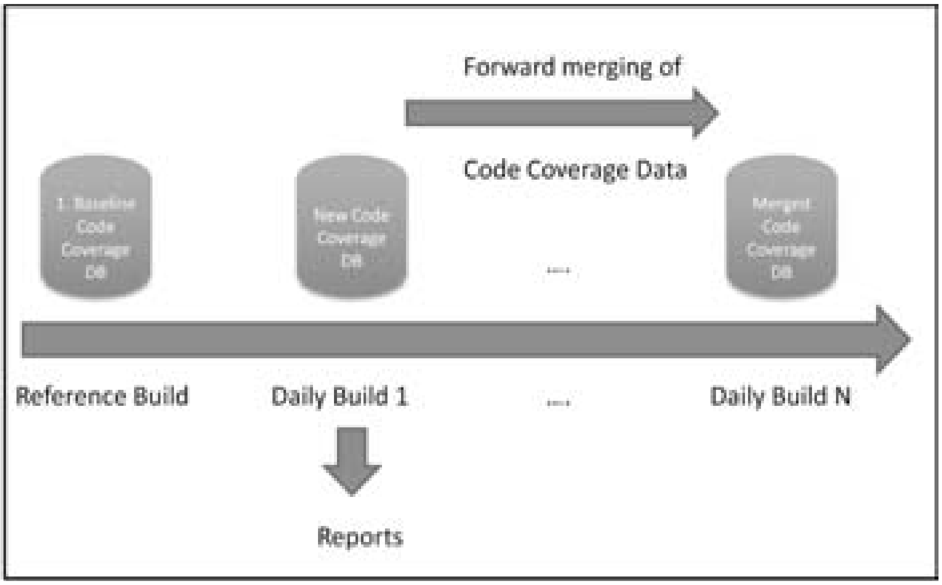


Figure 3: New Regression Testing Process Flow

Before test teams could apply this technology, process and tools, they had to collectively run their test suites against an instrumented product (reference) build for each major product release. This collected code coverage data, which then became the baseline code coverage database for subsequent maintenance releases. This baseline met the code coverage quality bar of 70% block coverage.

As the date for a service pack release grew closer, a new code coverage database was created alongside stable, instrumented daily product builds. This database was used to capture code coverage data from each team as they contributed to the testing of the service pack. The intent was that each team conducted a test pass using their regression tests (with or without guidance from *TCIndexer*) and then supplemented them, if necessary, with new tests. During and upon completion of the test pass, teams measured their progress and success against the exit criterion using *ChangeTested*.

As new product builds were released and teams executed their tests against those builds, a central team with Developer Division ensured that all coverage data was aggregated for the entire duration of the test pass. The final, merged code coverage database along with the product binaries was archived. This data now has the potential to be leveraged at a later date as baseline data for additional test passes where the above process would be repeated. The result was that test teams would reduce their regression testing workloads, the closer they got to the final release date for that service pack.

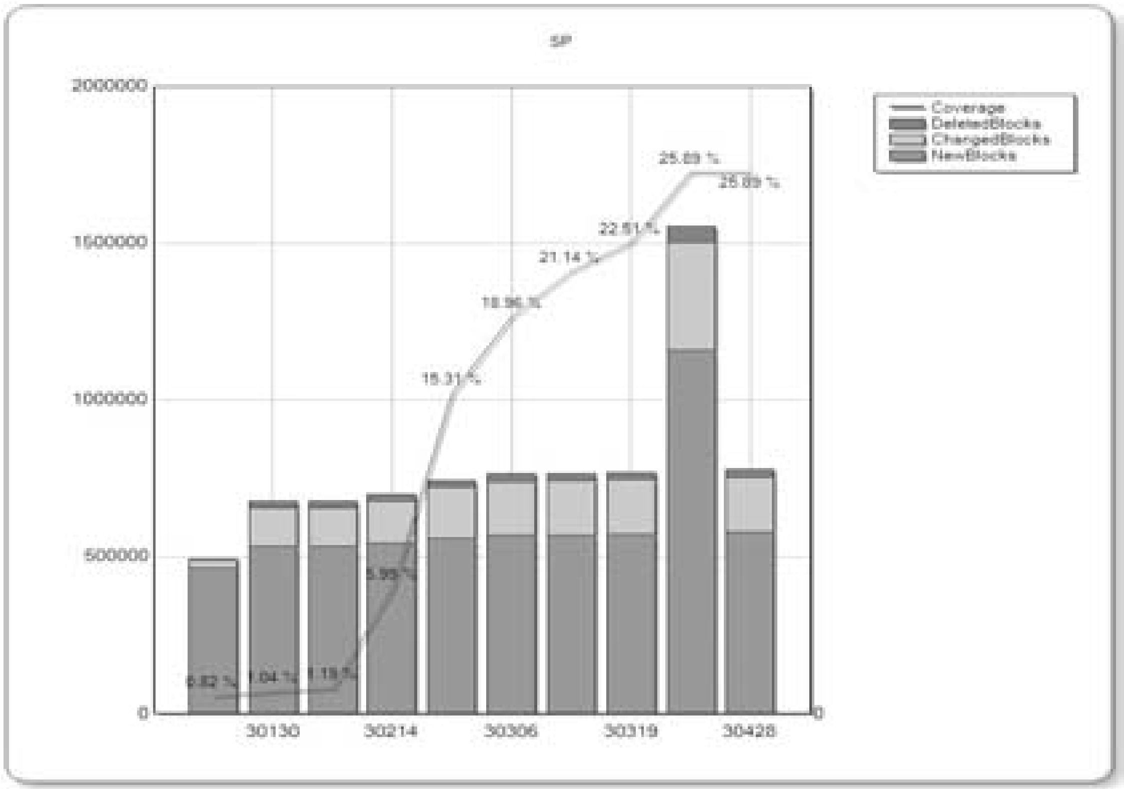


Figure 4: Code Coverage Exit Criteria Reporting

During this entire process, the output from *ChangeTested* (in the form of reports) showing % coverage of the churned code were updated and delivered to our ship room to give an indication of how close we were to meeting the coverage quality bar.

Figure 4 shows the type of data being collected during the initial stages of a test pass. The x-axis indicates the successive builds of the same product over time., while the y-axis indicates the number of code blocks being churned for each new product build compared with the baseline code binaries. The stacked bar chart indicates the contributions being made by the different types of churn – deleting, adding or modifying code blocks! This chart clearly shows the impact and timeframe of code integrations from different team branches into the main product build branch. Moreover, we superimposed the coverage data upon the code churn data via a line chart to provide a unified view.

5. Benefits and Challenges

While we only recently started to deploy this approach for service pack testing, the test teams saw two immediate benefits regarding their test focus. They were able to quickly identify test holes corresponding to churned code and create additional scenario tests to exercise that code. They also appreciated the diminishing workload associated with code coverage test passes required at each release milestone, because they only needed to validate the churn that had occurred since that last milestone. Examples of this were discussed in Section 3.1. Unfortunately, this work is so recent that we are unable to make any kind of statement regarding the impact of this approach on defects.

The biggest challenge was technical in nature and has the potential to seriously impact this new process. When code coverage data is initially collected in a reference database, a persistent mapping results between a test case and the binary code blocks it traverses. If changes are made to the compiler used to compile source code into a binary, then it may cause different binary code to be emitted. Thus, even if no changes were made to the source code for that binary, compilation using that latest version of the compiler would cause the binary differencing tool to flag incorrect differences between the previous and new versions of the binary. *TCIndexer* would then consume those differences and attempt to compute an incorrect subset of tests for rerun, In addition, *ChangeTested* would show incorrect churn that would need to be covered[4]. In summary, compiler changes can lead to a chain reaction resulting in incorrect data being presented to test teams.

This challenge was mitigated to a certain extent by recompiling the baseline build binaries with the same compiler version used to produce the latest product binaries. That enabled us to at least monitor progress regarding coverage of the churned code via *ChangeTested*. However, it left us with fewer options regarding selective revalidation as the binary differences that were being determined could not be used to help determine a subset of tests for rerun.

[4] *Note: This situation is relatively unusual as many Microsoft product teams maintain the same compiler version for building their products over long time periods in which case, the above approach works well.*

In summary, two approaches are being discussed:

∉ Base the above methodology on source- rather than binary-level differencing to overcome the compiler issue. We can then fully leverage selective revalidation and retain the redefined exit criteria.

∉ Use *test set optimization/prioritization* to determine a subset of regression tests, effectively ignoring any code changes. This would result in larger numbers of regression tests being rerun compared with subsets selected based on code churn and we would need to revert back to the existing exit criteria of % coverage per binary.

6. Conclusions and Future Work

Regression testing remains a very important topic for companies such as Microsoft who are developing and maintaining large software platforms and applications over long time periods. Testing technologies, such as selective revalidation, have the potential to deliver huge savings in maintenance costs for the company. We are now starting to deploy these technologies, albeit with a few challenges, and gather the necessary data to make that case. This paper chronicles how we are starting to deploy selective revalidation technologies and adapting our tools as well as processes to leverage them effectively.

In future, we intend to apply the test exit criterion to the validation of the individual products before they are integrated into the Visual Studio product (suite) *as well as after*. This will contribute to the quality of our individual products as well as the overall product. We are therefore working on replicating the above methodology, processes and tools for each product test team. The result is two instances where the test exit criterion needs to be satisfied. The test teams would thoroughly test their products prior to final integration and then rerun a representative (sub)set of their tests against the overall product suite. The latter would also have the beneficial effect on contributing/improving the coverage numbers of other product teams.

While the processes and tools outlined in this paper are internal to Microsoft, the concepts described here are easily repeatable and implementable using commercial-off-the-shelf tools including code coverage analyzers, test execution harnesses and management systems, differencing tools, etc.

7. Acknowledgements

I would like to thank Mak Agashe and David Yee, Directors of Test in Developer Division Engineering for their ongoing support. I also want to express my gratitude to my Microsoft colleagues, particularly Carey Brown, Apurva Sinha and Ephraim Kam for their significant contributions, support and discussions concerning this work as well as to the Developer Division code coverage champions for their feedback as we continue to

deploy and refine these technologies throughout the division. Thanks also go to my reviewers whose invaluable feedback and comments are greatly appreciated.

8. References

1. J. Hartmann, "*Applying Selective Revalidation Techniques at Microsoft*", PNSQC 2007, pp. 255-65.
2. V. Chvatal, "*A Greedy Heuristic for the Set-Covering Problem*", Math. Operations Research, pp.233-5, Aug. 1979.
3. A. Srivastava and J. Thiagarajan, "*Effectively Prioritizing Test in Development Environment*", International Symposium in Software Testing and Analysis (ISSTA), pp. 97-106, 2002.

Outnumbered: Ensuring Quality with a Low Test-to-Dev Ratio

Brian Rogers **brian.rogers@microsoft.com**

Abstract

How many testers does a software project need? Ask a test manager and you will likely hear "as many as we can find!" Unfortunately, it is often difficult to staff a testing organization with as many resources as is optimal. Does this mean a project must suffer poor quality if it falls short of the magic "1:1" test-dev ratio? Absolutely not!

Projects with smaller test teams are forced into certain behaviors out of necessity. As it turns out, these behaviors can be great assets in the quest for high quality. Some of these behaviors include customer-focused planning, aggressive trimming of unneeded functionality, and balancing of traditionally tester-only tasks across the team. While these are almost requirements in smaller organizations, teams of any size can learn and benefit from these activities as well.

A key factor for success given a small test team is the recognition that quality is everyone's responsibility. Testers and developers must be partners, not rivals, in ensuring quality.

Author Bio

Brian Rogers has worked as a software development engineer in test at Microsoft for five years. For most of his career, he has focused on distributed testing of enterprise messaging systems, including a three year assignment on the first version of Windows Communication Foundation. He is a strong proponent of engineering excellence within the test discipline, and has designed and developed static analysis tools for detecting defects in test artifacts. Brian has a Bachelor of Science in Computer Engineering from the University of Washington.

Introduction

The tester-to-developer ratio is a subject of much debate within software circles. In modern software projects, most test managers seem to have settled on 1:1 as the ideal ratio. Why is this so? Ask and you may hear one or more of the following reasons:

- "We need at least as many testers as developers to effectively 'police' the team."
- "If one developer is assigned to code a feature, then we obviously need one tester to test that feature."
- "A project with fewer testers delivers lower quality."
- "That's how we've always done it."

As a consequence, when a project suffers from a dearth of testers it may as well be doomed from the start, at least according to conventional wisdom. But what real evidence is there that 1:1 is correct? Products with large test teams still have bugs, still ship late, and still miss requirements.

I assert that the behaviors of the team can make a much bigger impact in the quality and overall success of a project no matter how many testers and developers there are. Furthermore, the behaviors which are inherent in smaller test teams can provide insight into some general best practices for all test teams.

Background

As a professional software tester for over five years, I have been involved in both large projects and smaller projects. In my experience, most of the large projects had a tester/developer ratio which held steady at around 1:1. This was true for the most recent large project I worked on directly, with over 100 engineers participating. To contrast, in the current smaller project on which I am employed, the tester/developer ratio has consistently been 1:2, with three testers and six developers.

While it may shock the traditionalists, the small project has maintained exceedingly high standards for quality. In addition, this small project has been described by participants and stakeholders alike as highly successful from a test standpoint. How did we get here? I will say that it was not through luck but through a very deliberate set of behaviors, some of which run counter to long-established practices of software testing and development.

As a caveat, I will note that the following recommendations and experiences are intended to apply to home, business, and enterprise software projects. I make no claim that this information is relevant to life-critical systems or any other strictly regulated development projects.

Quality

So far, the term "quality" has been thrown around without a formal explanation. What does high quality mean for software? As with all abstract concepts of this nature, the exact definition is open to interpretation and may vary depending on the person or project. To paraphrase the ISO definition, quality is the degree to which a set of inherent characteristic fulfills requirements.[1]

This is a fair starting point, but we can definitely expand on it. High quality software products have acceptable behavior and performance as perceived by the primary users of the software. The product allows these users to perform a set of tasks in a way that meets their expectations. Invariably there are disagreements between the product team and the end users about what "acceptable performance" means and whether feature X should have been implemented or not, and so on. Still, high quality software delights the customer more than it disappoints with respect to the above aspects.

Who "owns" quality?

In many projects, the test team is given the dubious honor of being the sole guardian of quality. It is likely that the perceived importance of tester/developer ratios and the horror of being "outnumbered" stem from this practice. Certainly, testers tend to be more quality-focused, but it is unfair to let this responsibility rest on any one discipline. As has been stated by many others (including the American Society for Quality), "Quality is everyone's job." It is only through a team effort that software projects can deliver high quality results.

The role of test in quality

The above points notwithstanding, there are certain activities intended to drive quality which are uniquely associated with test teams. These include the following:

- Writing test plans
- Executing manual and automated tests
- Analyzing and characterizing aberrant behavior of the software under test
- Filing defect reports
- Verifying bug fixes to catch regressions

This is a list which could have been lifted from Software Testing 101, so of course the test team (and the test team alone) does all of these tasks – right? Well, yes and no, as we will discuss later.

To contrast, I will present a set of activities which also have a large impact on quality, but many test teams do *not* participate in:

- Customer engagement
- Fixing code defects
- Writing and reviewing product documentation

What causes testers to pass on such important activities? Certainly there are organizational pressures in many teams which prevent testers from looking at these work items; after all, they are not associated with the traditional role of a tester. Perhaps more significantly, time pressure prevents otherwise motivated people from considering new activities. Testers have so little time to do the work they are "supposed to do" that they cannot even think of taking on additional tasks.

Learning from small test teams

Given the oft-demonstrated phenomenon of the test team time crunch, it is instructive to look at a case study of a smaller test team. It stands to reason that a small team will be more sensitive to such time constraints and will develop behaviors to deal with them effectively. Indeed, lack of resources is a good forcing function for encouraging the right set of activities at the right time, with as little wasted effort as possible.

In the following sections, I will share some information about my current project and present three approaches that my small team has used to great success in ensuring a high quality product. I cannot speak for every small test team, thus some of these experiences may be unique to my organization. If nothing else, I hope that managers and individuals in teams of any size can use the approaches that my team has taken to identify areas of improvement for their projects.

About the project

My current project is a framework primarily targeted at developers, so automated testing is a large part of the quality assurance process. As testers, we must write small pieces of software to validate various components in the product. In previous similar projects, the test team was more or less solely responsible for planning, developing, executing, and analyzing the results of these test cases. Understandably, this is a difficult task, and the software systems involved in testing the product will often approach the complexity of the software systems being tested. This is one of the reasons why all projects of this kind made great efforts to maintain a 1:1 tester/developer ratio.

Faced with the realities of our staffing situation, a 1:2 tester/developer ratio that saw no signs of changing anytime soon, we knew early on that we could not follow the methodologies of past projects and expect good results. One of the first decisions we made was pivotal to setting the tone for future innovations and successes of our test team: all developers must write unit tests to validate the functionality of the components for which they are responsible. This practice led to the adoption of a number of related behaviors which I summarize as **finding innovative ways to balance quality-oriented tasks across the team**.

With developers largely in charge of low-level and core validation tasks, the test team needed to clarify its mission. By **using a customer-focused approach to project planning**, we were able to come up with test tasks and strategies which everyone agreed were essential to shipping a high quality product.

As the product evolved and new features were proposed, we were faced with the challenge of adding work items for a test team already operating at near 100% utilization. With our strong customer focus, we became quite good at pushing for answers to a very important question, "Why do our users need this?" If there was a clear justification for the added functionality, we would of course find a way to take on the additional work. If not, we often found that the right thing to do was to **push for less functionality in order to keep the project on target**.

There were certainly other practices that we employed along the way, but I believe these three were emblematic of our most successful behaviors.

Behavior #1

Find innovative ways to balance quality-oriented tasks across the team

As mentioned previously, the decision to put developers in the business of testing their work was the spark that started a wildfire of best practices that I credit for our continued success. This example of sharing previously "test-only" tasks with developers did much more than give us confidence in the core of the system and early warning into product defects (although these were certainly huge benefits). Since developers knew they had to maintain the product code as well as their test code through any future changes that would be made to the code base, the product components ended up being loosely coupled (and thus testable in isolation) with more carefully thought out design.

Having developers write tests is not a new concept; participants in agile development processes have been following these practices for years. Even in the larger projects I have participated in, developers have done some amount of testing. So what exactly did we do differently?

For one, our developers took testing more seriously than in any previous project I have seen. We jointly developed a "Developer Test Contract" which formalized the responsibilities and expectations of developer tests. The contract called for a brief test plan to be written as part of the developer design documents (consequently opening them up yet to another previously test-only task). Developers and testers reviewed these unit tests with the amount of scrutiny that was usually reserved for production code, going so far as to reject code submissions due to a lack of tests.

We on the test team also integrated the developer tests into our own processes. We assumed a base level of coverage from these tests and opted to not duplicate what we knew was already tested. Instead, we took a scenario-oriented approach for our own tests, focusing on the higher level usage patterns of the customer and targeting our tests appropriately.

This stands in stark contrast to past projects, where it was normal for testers to write what essentially amounted to unit tests. Even in cases where developers had written some of these tests already, testers were not always aware of what was covered and there was a general feeling of distrust in these artifacts. Looking back, I blame this on a lack of oversight; if testers had been encouraged to explicitly review and sign off on these tests, they would not have felt obligated to fill in these perceived gaps.

Balancing of tasks is not a one way street. With the increased understanding of the product that comes from being involved in code and test reviews, testers on my team are often able to pinpoint and fix actual product defects. On many of my previous projects, this practice was so unusual as to be nonexistent. However, on my current project we take "quality is everyone's job" more literally.

The practice of testers writing production code is understandably controversial. In a product group where testers are thought of as the police and the development team are assumed to be criminals it would make sense that testers were discouraged from contributing to such a conflict of interest. However, if approached carefully, this practice can benefit everyone. First of all, we must stop thinking of the police analogy when we think of the tester-developer relationship; high quality software is built as a result of a partnership between these roles. Additionally, there should be no special treatment given to a tester who has submitted production code for approval. The code must pass all the usual independent quality checks and be given as rigorous a review as if it had been submitted by any other developer on the team.

Not every tester will be able to contribute to development work, and this is perfectly fine. Those who can contribute should be encouraged to do so. In short, the focus should be on choosing the right person for the task, regardless of primary role.

Behavior #2

Use a customer-focused approach to project planning

When the test team is focused less on the dev-centric notion of "classes and methods" and more on the end-user-centric notion of the "product experience," some interesting changes take place. As I touched on in the previous section, customer scenarios become the primary pivot for planning test work.

Presumably, all software exists to fulfill some customer need. These needs are often articulated like problem statements, the solution for which is outlined in a use case or scenario. For example, in the case of a multimedia software product, the statement may be, "Mom needs to create a DVD with a slide show of her family vacation photos." The scenario associated with this would have a sequence of steps Mom could follow to achieve the goal, e.g. "She opens the Slide Show Wizard and is prompted with the Select Pictures dialog," "She selects one or more pictures and presses the Next button," "Mom waits as a progress dialog is displayed while the pictures are encoded," and so on.

Software architects and developers are then supposed to think about the individual pieces that need to be created to actually build the software project fulfilling this scenario. In this example, there might be several back-end components that the user will never know about, such as a picture encoder, a set of interfaces to implement progress notification, and low level components to interact with the DVD device; whether the user sees it or not, of course, it must be designed and implemented.

Thinking back to my previous projects, the test teams focused almost entirely on these types of back-end components. This style of testing still uncovered defects but it was not always clear what the actual user impact was. This information would always have to be "reverse engineered" in essence by attempting to think of a realistic use case in which the bug would occur. In some cases, the tests were exercising purely negative paths which required passing bad data or performing other actions far outside mainline scenarios. Negative testing is still a valuable exercise, of course; the problem was that it was being done in preference to the more valuable task of proving the product works for its intended purposes.

Due to the intense focus on the low level aspects of the system, many testers did not fully understand the ways customers actually used the product. This prevented many otherwise knowledgeable testers from being able to participate in customer engagement activities.

On my current project, we recognized the pitfalls of the previous approaches and looked to scenario testing as a way to focus our limited resources in the right places. Instead of splitting up test tasks along software module boundaries, we created "scenario buckets" which grouped use cases into higher level system areas. Instead of pure negative cases, we focused on plausible errors or failures which a user could encounter and evaluated the system's reliability and consistency in these cases (this was part of the "Recovery" scenario bucket).

One of the many challenges of planning test work is prioritization. In the mathematically infinite space of testing, priorities can help decide what is essential and what is dispensable. Customer-focused planning is especially valuable here, as it makes it much easier to make this decision. In simple terms, the test cases which get first priority are those covering scenarios our customers exercise most often. The target platforms and conditions we choose for our test passes match our customers' systems as much as possible.

While this sounds obvious, the larger projects in which I have been involved have not always followed this guidance. Since features were often tested in isolation rather than as aspects of a larger scenario, it was hard to get a sense for how important a smaller case was in terms of the bigger picture. Even so, a larger team may be able to absorb the cost of a prioritization error by essentially throwing more people at the problem; if it turns out a last minute high priority test needs to be written and executed, a few testers can be taken aside for a week to complete this work. No such luxury exists on a small team, so it is critical to have the right plan the first time.

Behavior #3

Push for less functionality in order to keep the project on target

As professional testers, we are a unique group. We love to experiment with the products we use. Many of my colleagues spend considerable time setting up their computers in unique ways, even going as far as changing their keyboard layouts in the hopes that they will be more productive or efficient. Indeed, we cherish the flexibility and freedom of choice provided by the software we use.

What we must understand, however, is that typical software users do not want choices; they simply want to get their work done. By allowing features of questionable utility to slip into our products, we cause two problems, one for ourselves as testers and one for our users. As testers, we must verify that these features work, which means we must extend the schedule or drop other tasks to fit this into the release. Furthermore, our customers must deal with the additional complexity introduced by the new feature; this may come in the form of an additional dialog to click through, a new settings page, or an extra menu item, all of which have the potential to confuse or distract the user and reduce their overall satisfaction.

Members of a small test team cannot afford to have more features to test than are absolutely necessary to fulfill customer requirements. Testers should not be afraid to request outright removal of features which do not map to any known use case. I can point to a recent example of this on my team. In our product install utility we had originally planned to include a screen where some configuration settings could be modified. However, we had no actual scenarios where a user would want to change the default settings. Armed with this data, and the argument that the feature added a mostly superfluous, somewhat confusing, and mandatory additional screen to click through, we successfully lobbied to eliminate it from the product.

Adopting the behaviors of small test teams

My team has beaten the odds; we may be outnumbered but we are no less successful because of it. We have fought for changes to processes that we saw as unsuitable for a smaller team and won. We continue to deliver high quality releases to our customers. How can your team do the same?

First, the test team should encourage the developers to share in the joy of testing. Unit testing has become fairly popular in recent years and many developers are already doing it. Try to formalize the practice by establishing some guidelines and expectations for developer-produced tests. If possible, get testers involved with reviewing these tests. It is great education for both parties. Teams with non-programming testers can still benefit when developers write tests. A solid foundation of unit tests prevents defects from entering the product which saves testers time that would otherwise be spent having to investigate, file, and work around obvious bugs. Such “common sense” data can be very powerful in convincing skeptical team members that a new practice is worth a try.

Assuming a worst case scenario where the development team absolutely refuses to do unit testing, it should still be possible to apply the other techniques. Using customer-focused planning for test work and shifting the focus to scenarios is a test-centric change; given the usual recommendations that testers should “think like customers” it should make perfect sense to the rest of the team. Of course, pushing for smart feature cuts is an option available to any team; testers simply need to get into this mindset and embrace the “less is more” style of product planning.

Above all, it is important that any changes be made in gradual steps. A bad experience with a new practice can make future process improvement proposals much harder to sell, no matter how sorely they are needed.

Conclusion

I encourage all test teams, no matter the size, to consider the points I have raised and start thinking like a small team. Whether you work with two testers or 200, you can benefit from these behaviors. My experience has shown that the combination of sharing quality-related tasks, customer-focused planning, and trimming of features will lead to higher quality software.

Fight for your rights when you are outnumbered – push for best practices in quality assurance.

References

1. No author. "Glossary." 2007. Whittington & Associates, LLC. Retrieved 15 Aug. 2008 <http://www.whittingtonassociates.com/v2/glossary.shtml>.

Virtual Lab Automation: Best Practices and Common Pitfalls

Abstract

Virtualization is a ground-breaking technology that promises quantifiable benefits for application development and QA organizations: faster lab deployment, less manual set-up work, greater resource flexibility and utilization, and easier reproduction of defects.

However, adopting virtualization in a development or QA organization isn't without issues. Often it's not obvious whether to build out a custom virtualization framework or make a strategic bet to implement a full virtual lab management solution, complete with automation and a pool of centralized hardware.

This paper discusses the software quality challenges commonly faced by application development teams and how virtual lab automation can lead to a more strategic approach to QA practices. It describes the best practices for virtual lab automation adoption and also highlights the common pitfalls organizations face during implementation. Finally this paper outlines the steps to evaluate a virtualization solution for your QA organization and provides further resources to help you get started.

Ian Knox
iank@skytap.com
Skytap, Inc.

Ian is currently Director of Product Management for Skytap and is responsible for product management, market positioning, demand generation and go-to-market strategy. Ian joined Skytap from Microsoft Corporation where he was group product manager for Microsoft Visual Studio. In this role, Ian led the product management team responsible for building a new $400M enterprise tools business in the Application Lifecycle Management (ALM) category. Prior to Microsoft, Ian was a Principal Consultant at PricewaterhouseCoopers where for 7 years he worked on global software delivery projects for Fortune 500 clients.

Virtualization Forces a Rethink of QA Practices

A manager in charge of application QA needs to determine the best way to apply limited resources to achieve the goal of delivering a reliable, performant application. Given the nature of modern distributed environments, this job is becoming more and more difficult:

- **Increasing complexity.** Building a lab environment to support testing is a painstaking task for a typical distributed architecture. Implementing test environments that mirror production as closely as possible require that machine, network and application settings are carefully configured to ensure environmental issues are found before deploying to production.

- **Resource constraints.** Budgets are shrinking and procuring physical hardware, storage and network resources for test environments is costly and often hard to justify given low utilization. Applications often require testing on multiple operating system versions and language variants, browsers and devices, so achieving the optimum balance between adequate test coverage and acceptable risk is difficult to achieve.

- **Productivity bottlenecks.** The set-up and tear-down of labs is usually a time-consuming, manual process. This IT provisioning overhead is costly and reduces the time QA teams can spend testing an application.

- **Faster cycle times.** The broad adoption of agile development practices have put pressure on QA teams to reduce test cycles and work iteratively to deliver software.

- **Communication issues.** Developers often spend an inordinate amount of time trying to reproduce and debug defects reported by the QA team. If an application state is difficult to reproduce, it can mean hours of wasted time diagnosing an issue.

- **Globally distributed teams**. The growing trend of using offshore testing resources compounds the problem of sharing consistent environments across teams and facilitating effective team collaboration.

Given the rise of virtualization, application development managers and QA professionals are rethinking tooling, practices and skills to help solve these ongoing challenges. Many have already experimented with virtualization in their lab environment and are now realizing a virtual lab automation solution is necessary to overcome new challenges that a virtualized environment brings.

Virtualization Challenges

Many QA teams are using virtualization by building a proprietary framework using scripting, such as VMWare images with Perl scripts. This can provide many benefits including the ability to snapshot and restore images, faster machine deployment times and better utilization of hardware.

This approach has proved successful in many organizations, especially with ad-hoc or simple test frameworks. However, as many teams are learning, it can soon become a significant effort to develop and maintain scripts and a library of images. Additionally, it's not easy to deploy and manage multi-machine configurations in an isolated network without implementing virtual private networking. Usually there is no user interface to manage the test lab which limits the control users have over the lab environment. The cost of administration can quickly warrant implementing a more robust solution and many organizations are investigating virtual lab automation to solve the overhead costs of a custom solution.

The remainder of this paper suggests five best practices for adopting a virtual lab automation solution and some common pitfalls organizations should avoid.

Best Practice #1: Understanding Virtual Lab Automation (VLA) Capabilities and Limitations

Virtual Lab Automation (VLA) is the industry term that has been coined to describe a new breed of tools and test practices to automate labs using virtualization technology. A VLA solution can include some or all of the following capabilities:

Resource Pooling and Provisioning

Resource pooling enables processing power, storage and networking infrastructure to be shared between different teams and individuals, increasing utilization and availability of resources, and reducing costs. In conjunction with resource pooling, an orchestration and provisioning process allocates and releases resources as needed.

Multi-Machine Configurations

Virtual machine images are the containers that enable operating systems and applications to be isolated from physical resources. A group of virtual images that define a complete system, including network and storage characteristics, is defined as a configuration. For instance, a configuration could consist of multiple Windows Vista client machines, an Oracle database server and a WebSphere application server. Configurations can be easily created by combining virtual machines through a user interface. A configuration is a very useful concept for QA teams because it allows a whole system to be defined and isolated in a test environment. Virtual networking enables copies of the same environment to be run in parallel and the emulation of production environments during the test process.

Configuration Library

A configuration library allows a team to manage and organize virtual images and configurations. Standard builds and images can be created and made available to development and QA teams to save hours of set-up time and environment configuration. Additionally, the library is used to store new configurations that are cloned or created as part of a test.

Suspend, Snapshot and Restore

The ability to suspend a complete state of a multi-machine configuration and make a snapshot (a copy at a point in time) is a major benefit of virtualization. This is especially useful for application development teams because when a bug is found a configuration snapshot can be taken at the point of failure and a link to the configuration added to a defect report. Instead of a developer spending hours to reproduce the defect, he or she can restore the configuration and start debugging the issue within minutes.

Scheduling and Reservations

Many in-house virtual lab implementations have a fixed pool of resources for teams to share. Scheduling and reservation functionality allows the resolution of resource conflicts and test environments to be reserved ahead of time.

Reporting and Monitoring

Reporting modules allow users and administrators to manage usage and quotas and determine whether the system resources are being used optimally. Monitoring enables the system health to be diagnosed, including CPU utilization, storage performance and network usage.

Automation API

Automating a test lab almost always involves integrating tools and test processes. An automation API enables teams to automatically create test environments as part of the build process and initiate automated test runs once a new build has been deployed. An automation API is typically made available through a web services interface.

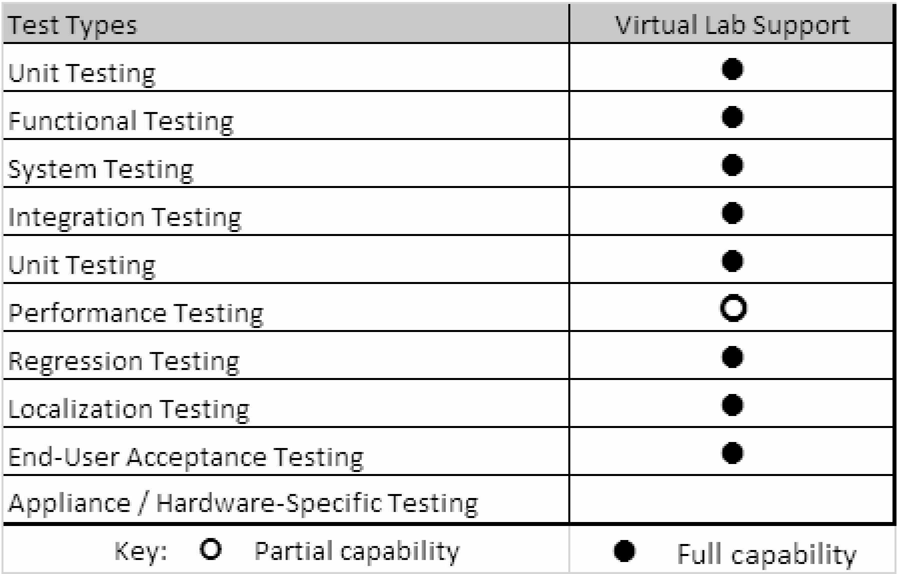
Administration and Security

Administration and security features often include user and quota management, project creation, permissions and authentication. Remote access to the system (for instance for an outsourced vendor) is usually managed through secure connections via encrypted protocols and virtual private networking.

These capabilities are undoubtedly attractive to the vast majority of QA teams. However, every team is different and determining whether some or all of these capabilities are beneficial is important to consider. Any QA manager who has implemented functional testing tools knows that there are some testing projects where the effort required to implement an automated testing framework can far outweigh the benefits. The same is true for virtual lab automation.

Common Pitfall: Misunderstanding test types that are suitable for a virtual lab

Almost all VLA solutions utilize standard infrastructure and support a wide range of test scenarios. Hardware and network characteristics can be easily configured through the web interface to specify number of processors, amount of memory and network settings of machines in a configuration.



Test Types	Virtual Lab Support
Unit Testing	●
Functional Testing	●
System Testing	●
Integration Testing	●
Unit Testing	●
Performance Testing	○
Regression Testing	●
Localization Testing	●
End-User Acceptance Testing	●
Appliance / Hardware-Specific Testing	
Key: ○ Partial capability	● Full capability

Fig 1. Test Types Supported

Typical customer test scenarios include unit testing, functional testing, system testing, integration testing and load testing of applications. However, there are a few use cases where a VLA solution is not recommended. These include tests which require specific hardware access (e.g. bios driver tests) and some types of performance and stress testing (e.g. a test of application performance on a specific hardware profile).

Best Practice #2: Determine the right implementation approach for your organization

There are essentially two approaches for adopting a virtual lab automation solution, an on-site package and a hosted virtual lab service, both with different advantages and disadvantages

There are a number of vendors now offering packages for Virtual Lab Automation, including VMware. Many of these solutions offer most or all of the capabilities discussed above. They have typically been adopted by large enterprise organizations where the expense, time and organizational changes required to build a centralized lab are worth the effort. These organizations have seen dramatic Return on Investment (ROI) from these new labs. Voke, an analyst firm, estimates a virtual lab automation solution can deliver a 25-50% reduction in hardware needs and an average time saving of 3 days to deploy a lab environment.

For QA organizations that are not part of a large enterprise organization, there are some considerations that may lead them to a hosted alternative (such as the solution offered by Skytap). First, implementing an automated test lab requires a large upfront investment in lab hardware and software. It also requires significant effort to implement, configure and train lab personnel. This type of expense is especially hard to justify for dynamic projects and if departmental QA budgets are under pressure.

Second, implementing an in-house virtual lab solution requires skilled IT resources to be assigned for administration and virtual image population and maintenance. Unless a lab reaches a critical mass to cover multiple development and test organizations, this administration overhead can be prohibitive.

Finally, even though an automated virtual lab improves resource utilization there are still going to be resource conflicts between teams unless an expensive pool of infrastructure is procured that covers peak demand. This means some test groups will still need to wait for resources, reducing their effectiveness and increasing delivery risk.

An alternative to an in-house virtual lab package is using a hosted lab or 'Virtual Lab-as-a-Service'. In the same way Salesforce.com offers a CRM package as a service, a virtual lab as a service solves many of the issues and risks associated with an in-house implementation. For example, there are no upfront investment costs and infrastructure can be scaled up and down according to testing needs. Furthermore, the administrative costs associated with running the lab are handled by the service provider. A hosted lab can be easily integrated back to on-site assets using a Virtual Private Networking (VPN) connection and typically a customer will only pay for the hourly usage of the lab machines when in use, eliminating the expense of test machines sitting idle.

Common Pitfall: Ignoring Indirect Costs

When determining the Total Cost of Ownership (TCO) for each approach, it's important not to forget to include the costs for internal IT support, as well as the more obvious hardware and software capital expenditures. Often, much of the ongoing costs for a lab involves IT administration. Also, indirect costs should be included such as time savings for development and test team members gained by using the capabilities of a VLA solution.

Best Practice #3: Automate Your Test Lab Operations

One of the major benefits of a VLA solution is the ability to integrate with a build process and testing framework to enable an automated workflow. Typically, this workflow is enabled through scripting or tooling support in the build server and/or automated testing tools. Almost all VLA solutions (both in-house and hosted) offer an API to enable integration.

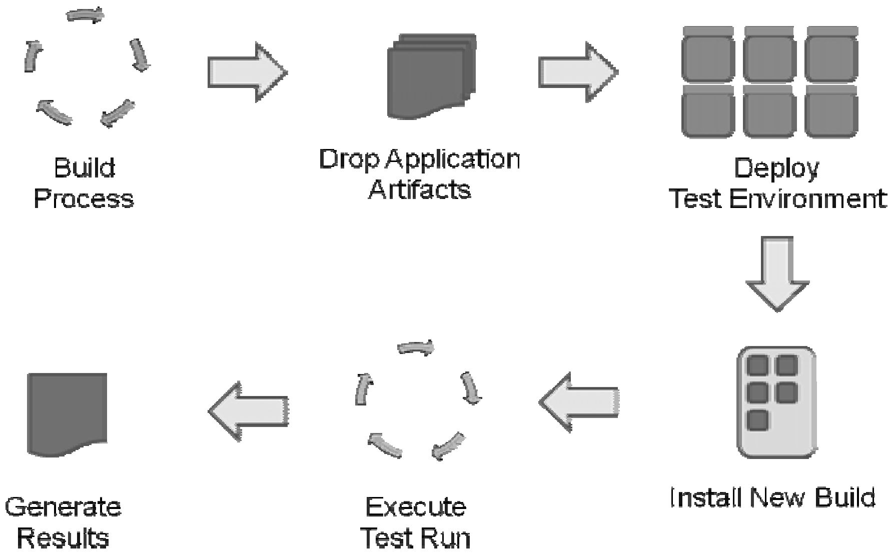


Figure 2, Typical Lab Automation Process

Investing in populating an asset and configuration library will enable standard environments to be deployed quickly. Many environment set-up tasks can be automated to avoid manual user intervention. For instance, as part of a nightly build, new virtual machines configurations can be automatically deployed and software builds and patches installed in preparation for a test run. In addition, using the virtual private networking functionality found in a typical VLA solution, a test environment can be deployed to mimic a production environment as part of the automation process.

Best Practice #4: Enable Team Collaboration with User Permissions and Projects

Every VLA solution offers the ability to specify user access levels and permissions. Typically an administrator will have access to the entire lab, team leads have the ability to create new projects and environments and individual testers work as part of a project and have access to only the resources they need.

Once user access control has been specified, administrators and team leads can enable or restrict access to resources through the virtual lab user interface. This is especially useful for projects where outsourced vendors are utilized. A project can be created with the environments to be tested and outsourced testing professionals can be given access only to the resources required for a given test run. If an organization is using a hosted lab, it's easy to ensure the test environment is isolated from the corporate network.

Now environments are stored in a configuration library, replication of defects becomes much easier as both development and test teams are sharing the same environments on the same virtual infrastructure.

Common Pitfall: Not Restricting Access to Master Configuration Images

There will be a set of master (or 'Gold') configuration images that are commonly used to create environments. These include standard desktop images, application builds and server images. It's important to ensure these images can only be accessed by a lab administrator and not accidentally modified by a developer or tester.

Best Practice #5: Obtain Team Buy-In With High Impact, Low Effort Changes

Once implemented, training your staff about the use of a VLA solution is the first step to encourage adoption. However, demonstrating how it will help make their jobs easier is equally important. By choosing a few high impact areas to focus on and securing some quick wins to improve productivity, you team is much more likely to adopt the new solution.

One of the most obvious areas to focus on is populating the configuration library. If you have chosen a hosted VLA solution, this will be pre-populated and you will only need to update base virtual machine images to match corporate images. This will enable IT operations and QA leads to very quickly assemble and deploy new environments, dramatically cutting down the 'time-to-test' and simplifying environment configuration.

Another high impact, low effort change is to ensure snapshots of virtual machine configurations are captured for new defects. This will enable testers to more easily communicate defects (some have called it a 'screenshot on steroids') and developers to isolate problems in a fraction of the time it previously took.

Common Pitfall: Inadequate Training of Outsourced Test Teams

Organizations will typically roll-out training programs to their staff, but may forget to do this for outsourced test teams, especially when new service providers are added after a VLA implementation. Conducting short training sessions using a Webinar solution will ensure outsourced teams are equally productive with a VLA solution.

Summary

As part of evaluating approaches for adopting virtualization in your development and testing organization, we recommend the following steps:

1. **Determine the Capabilities Your Organization Needs**
Build a requirements matrix and determine the types of testing and usage patterns typically seen in your organization. Consider operating platforms and hypervisor vendors used in your environment as well as the current skills of your IT operations and QA team.

2. **Explore the Potential Solutions**
Evaluate an in-house package and using a hosted virtual lab service.

3. **Evaluate Total Cost of Ownership (TCO)**
Build a Total Cost of Ownership model. Be sure to include software, hardware, implementation, and administration costs. Also try to estimate the indirect costs associated with each option.

4. **Conduct a Trial Project**
Conduct a proof-of-concept or trial project using your short-list of solutions.

5. **Implement Solution and Refine Your QA Process**
Once you've tested and implemented your solution, we recommend evaluating your current test practices, updating these to reflect the new virtual lab capabilities and investing in training your team before rolling out the solution broadly.

Proceedings Order Form
Pacific Northwest Software Quality Conference

Proceedings are available for the following years.
Circle year for the Proceedings that you would like to order.
2008 2007 2006 2005 2004 2003 2002

To order a copy of the Proceedings, please send a check in the amount of $35.00 each to:
PNSQC/Pacific Agenda
PO Box 10733
Portland, OR 97296-0733

Name____

Affiliate_______________________________________

Mailing Address_________________________________

City______________________State_____Zip__________

Phone_________________________

Using the Electronic Proceedings
Once again, PNSQC is proud to produce the Proceedings in PDF format. Most of the papers from the printed Proceedings are included as well as some slide presentations. We hope that you will find this conference reference material useful.

Download PDF – 2007, 2006, 2005, 2004, 2003, 2002, 2001, 2000, 1999

The 2008 Proceedings will be posted on our website in November 2008.

Copyright
You can print articles or slides for your own private use, but remember they are copyright to the original author. You cannot redistribute the Acrobat files or their printed copies. If you need extra copies of the printed or electronic Proceedings please contact Pacific Agenda at tmoore@europa.com.