Volume 1, No. 1 Fall 2004

Software Quality

OSQA'S QUARTERLY MAGAZINE FOR SOFTWARE PROFESSIONALS

Ensuring Enterprise Application Reliability

a manager's guide to risk-based testing

Motivating Extreme Development

a goal-based approach to best practices

Critical Path

the productivity equation (part I)

and more...

PREMIER ISSUE
Perspectives on Software Quality

OSQA

CARS™

Compuware Application Reliability Solution™

THE POWER TO Test, Validate, Deliver

Solve application reliability problems with Compuware's one-stop quality solution. CARS brings together the tools and expertise you need to establish quality assurance discipline. Manage risk with methodologies that align quality assurance activities to business requirements. Make informed deployment decisions. Implement best practices and confidently deliver applications your business can rely on.

THE POWER IS RIGHT HERE.

THE LEADER IN IT VALUE.

COMPUWARE®

www.compuware.com

This Issue

Perspectives on Software Quality

Software Quality Fall 2004

Letter from the President

Since its inception three years ago, we have planned to start a regular quality magazine for the Ottawa Software Quality Association. It took some time to get here—to secure the funding and to find people committed to making it happen. So, we're very pleased to present the inaugural issue of Software Quality, OSQA's magazine.

The OSQA executive believes in a need for a magazine focused on software quality experiences in Canada. Software Quality provides a forum for organizations in Canada to exchange information in a structured manner. It focuses on software quality rather than attempting to cover all aspects of software engineering. Software Quality also provides a means for vendors to access the local market.

OSQA's Software Quality magazine will provide practical solutions to the needs of quality engineers, engineering managers, and project managers. In some cases, articles will augment the regular monthly meetings, special seminars, and conferences that OSQA organizes, and in others they will be original.

We are looking for experience reports with practices, processes, and tools, innovations in quality improvement, reviews of quality literature, and opinion pieces. Not that we eschew theoretical or research contributions, but if material cannot be of immediate value to practitioners then it is probably out of scope.

Articles in Software Quality are peer reviewed by a dedicated editorial board. The members of the current board are listed in this issue, and deserve a big thank you for their hard work. Thank you also to everyone who contributed to this first issue. We plan to publish every quarter. If you wish to submit an article, please contact Teresa Wilde at twilde@osqa.org.

I would like to especially thank Matthew Hately from Macadamian Technologies in Ottawa for demonstrating leadership in taking this magazine from just an idea to an actual product, and Teresa Wilde, also from Macadamian, for her work as Associate Editor.

OSQA's Software Quality magazine is in good hands.

Dr. Khaled El Emam is renowned for his experimental work in software engineering. He has held positions at the Fraunhofer Institute and the National Research Council. He is also an adjunct professor at McGill University in Montreal and sits on a number of editorial boards of software engineering journals.

Ensuring Enterprise Application Reliability

a manager's guide to risk-based testing

Carolyn Barber, Director, Worldwide Compuware Application Reliability Solution
Elizabeth M. Maly, Director of Marketing, Professional Services and Quality Assurance
Compuware Corporation

The Software Quality Challenge

In today's business environment, IT is challenged to do more with less. Budget constraints and growing competitive pressures are forcing IT, especially the quality assurance team, to re-evaluate how they can successfully meet their business' requirements. This challenge has intensified as applications are deployed over the web to customers and business partners alike. When an application fails or performs poorly, a company risks losing not only productivity, but also significant revenue.

New and emerging technologies have mandated that IT organizations build "products" as well as "applications." Building a "product" means that many users, both inside and outside the enterprise, will use the software. The more pervasive an application becomes, the higher the total reliability and quality must be. The definition of quality expands in this environment to mean:

- Functional business fit
- Ease of use
- Speed of delivery into production
- Reliability in the production environment
- Minimal disruption for maintenance and support

Not achieving quality costs the business in lost productivity and poor use of resources. Research from the Gartner Group indicates that the average cost of unplanned downtime for a mission-critical application is $100,000 per hour.

Traditionally, IT has taken a siloed approach to the development life cycle. Business analysts developed the requirements, development coded, and whatever time was left would be devoted to testing, just prior to release into production. In today's complex environment, the siloed approach leaves many opportunities for risk. A significant risk occurs when software testing not considered until the end of the development process and then it is only given the resources "available at the time." As business processes become more complex, the software applications necessary to support those processes also become more complex. The software testing process must keep pace with this complexity and the heightened risks of delivery in order to be effective.

Research from Meta Group indicates that approximately 60%-70% of IT project failures result from poor requirements gathering, analysis, and management. Risks of poor communication and collaboration between departments result in applications that do not

meet business or operational expectations. Executives are only aware of these risks after an application has gone into production.

A recent issue of Internal Auditing magazine stated: "the biggest risks facing organizations are now technology-based." IT failures impact the enterprise value, its reputation and competitive position. When applications fail, revenue, reputation and customer confidence all suffer. Valuable resources are expended without a return on investment. Deadlines are missed and budget overruns go unnoticed until it is too late to take corrective action. Without the proper means to derive critical information, an enterprise is limited in its ability to mitigate risk.

The Growing Awareness of Software Reliability

There seems to be little disagreement between IT professionals that failing applications pose substantial business risks. As awareness of these risks grows, IT organizations are seeking strategies to minimize them. An important trend is that this awareness is not just on the part of the IT organization, but with business executives as well.

A 2003 study conducted by Forrester Research found that 82 percent of executives in North America and Europe associated application failure with material revenue loss. Indeed, this study also found that 44 percent of these executives had experienced an application failure during the past three years with 64 percent of this group directly experiencing material revenue loss due to application failure. When IT executives were recently asked which aspect of application quality has the greatest impact on their business, their most frequent response was 'meeting business requirements.' Yet the same IT executives, when asked what quality metrics they used to ensure their applications met quality targets, cited a metric that has nothing to do with business requirements. That metric was – 'Delivery of an application project on time and on budget.' It appears that meeting project deadlines and budget constraints may be easier to measure and report on than application performance in production.

This study also found a number of interesting facts regarding software quality including:

- Executives in general acknowledge significant consequences of application failure, and know those key applications where failure would have the greatest impact.
- Business executives have a less favorable view of application quality than IT executives.
- IT executives indicate that they intend for testing to minimize risk. However, application quality tracking does not consistently back this intent.
- Companies deploy flawed applications because of the lack of timely information about quality problems.
- Companies have invested in tools and processes but in many cases haven't seen compelling improvements.
- Executive least satisfied with application quality cited inconsistent implementation of quality methods throughout their organizations.

The Impact of Outsourcing on Software Quality

An increasingly compelling issue for today's IT organization is ensuring application reliability when development is outsourced to a third party – either locally or offshore. Companies outsource for several reasons including:

- To minimize time-to-market pressures.
- To supplement limited staff.
- To reduce development expenses.

Organizations must not look to outsourcing for an answer to the quality problem. Some organizations have taken a shortsighted approach and hastily sent their entire IT organizations to an offshore provider based solely on cost. The theory is that cost is causing the poor quality and customer satisfaction issues. In

reality, what IT organizations have failed to realize is the lack of maturity, communication, and collaboration in their organization are frequently the real culprits.

Whether enhancing a development time line or augmenting limited resources, outsourcing increases the need to perform a greater amount of testing to ensure application reliability. Many organizations have indicated that when development is out of their control, it is imperative to expand testing to compensate. For example, a large manufacturing company has adopted outsourcing for all of its development. And a health care company outsourced all of its regulatory compliance development to meet the deadline. For both of these companies, having a comprehensive testing plan relieves concern about development quality when the development is outside the control of IT.

The Impact of Web Services on Software Quality

As new technologies such as mainstream web services are implemented, quality becomes an even greater issue. While new or experimental technologies are in the "skunk works" phase, the goal of research is to determine whether the technology can be of use in the real world. During research time, appropriate for prototypes, software engineering best practices are not at the forefront. Management decisions to skip skunk works projects result in failure and quality nightmares.

Because of their structure and dynamism, web services expose and exacerbate architectural flaws and organizational weaknesses. For example, a monolithic application may work as expected 90 percent of the time, which is an accepted level of quality for most applications. However, in a web services scenario of several loosely coupled components that work as expected 90 percent of the time, the system as a whole is only functioning at a level of 0.9 raised to the number of web services in the loosely coupled system. For example, if there are three components, 0.9 cubed (0.9 x 0.9 x 0.9) will yield a level of 0.729. Hence, the overall system will be operating at a level slightly better than 72 percent, rather than at 90 percent, as the individual quality of the web services may indicate.

Limitations in scalable architectures will also be exposed as web services are joined with others to form a "system." Web services that have not been tested thoroughly for capacity will result in the lowest common denominator of failure.

Prior to new technologies entering production, issues surrounding new technology processes, tools, vendors, skill sets, performance and other issues need to be addressed and solved. All of the issues surrounding quality have, for the most part, been pushed aside by application development organizations until the technology is relatively close to being used in production by early adopters. This practice significantly increases the cost of quality. Planning for quality from project inception is the key to project success.

Who is Responsible for Quality?

Given the significant pressure that IT organizations are under to control and reduce development costs, the issue of who should be responsible for testing frequently comes up. The fact of the matter is that the entire team — development, quality assurance, and production/operations — are responsible for quality and need to be addressing quality throughout the development lifecycle.

Some organizations will perform testing using just development resources – which can introduce unique challenges. Quite simply, the development team is the wrong resource to be responsible for quality assurance. While development resources excel at writing code and should be responsible for conducting code diagnostics and unit testing, they traditionally do not prefer, or function well with, overall application testing. When testing, developers check to see if their functionality works as designed versus testers, who fully test the

application to ensure that requirements are met. Both developers and testers are critical to the software development lifecycle, but it is commonly accepted that the two skill sets are not interchangeable.

Unfortunately, in house testing is not typically assigned the same level of importance as development. And, there are organizations where people inexperienced in testing, or those who do not have an aptitude for testing, conduct application testing. The reality is that testing is most effective when done by experts who have an understanding for the software testing process and are equipped with the appropriate tools.

Many companies do not employ a well defined, repeatable, testing process. Because of this, some organizations are not able to realize the efficiencies that can be gained when testing processes are repeatable, and when test assets are re-usable. They do not have the insurance that every testing project will follow the same steps as the previous project. Conversely, those organizations that do follow a consistent testing process experience the benefit of having a consistent approach to evaluating application quality. Further benefits can be realized if tools are used throughout the testing process to automate testing steps, such as executing tests, tracking defects, or managing requirements. But without a process that ensures each person uses tools the same way each time, potential productivity gain is not usually realized.

Therefore, without testing professionals who know how to test, a proven methodology to ensure repeatability, and the right tools to improve productivity, IT organizations have no reasonable guarantee that they are deploying reliable applications that will meet their business objectives. This combination affects application quality, time-to-market and profitability.

What is Risk-based Testing?

Ensuring that applications meet business objectives and are ready for deployment can be a daunting task. After all, how does an organization know when an application is ready for production? How does it know that enough testing has been done? Do you stop testing just when you run out of time?

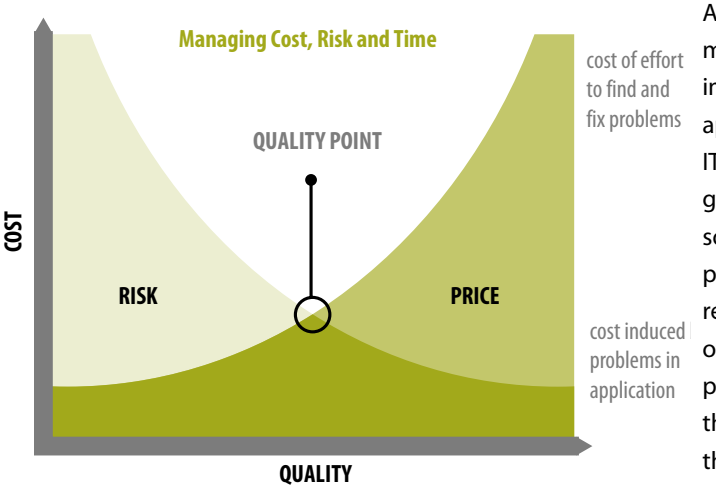


Figure 1: Benefits of Finding and Fixing Problems Early

Adopting a risk-based methodology greatly improves the quality of applications, allowing IT organizations in general, and the software QA manager in particular, to confidently report on the readiness of an application for production along with the potential impact on the environment.

So, how does risk-based testing work? Risk-based testing is based on a cost model that depicts the logical point between investing more time and effort to eliminate a defect and the cost of leaving the defect in place.

Figure 1 displays this cost model. As shown in the model, the testing cost increases exponentially as the software approaches zero issues. This is true because the resources remain the same but not as many defects are being uncovered. On the other hand, risk increases exponentially as the amount of testing decreases. The quality point can only be determined by using processes with associated measurements and a risk

analysis approach based on measurements taken. To the left of the quality point (break-even point), the risk is usually much higher as the application will be released with unresolved issues. Risk is lower to the right of the quality point because more issues are resolved, but resulting costs are higher.

Organizations will typically have a large number of enterprise applications — and most likely each of those applications will have a unique quality point. Determining what that point is requires an understanding of both the application and the business. Effective risk-based testing provides organizations with the confidence that applications will meet business and operational requirements.

QualityPoint – A Proven Approach to Risk-based Testing

To help organizations identify and manage the business risks that have become an integral part of the application development lifecycle, Compuware Corporation has developed a patent-pending risk-based methodology called the QualityPoint testing methodology. This methodology provides a way to ensure that quality assurance activities align with the most critical, highest priority business requirements. It links measures of an application's readiness for production back to those requirements through defined metrics. Implementation of the methodology creates a repeatable testing process that can be re-used to drive additional defects and the cost of poor quality out of the application.

Traditionally, most of the quality assurance activity involved with software is more testing than quality assurance. There is a difference between the two. Testing is the final stage in an overall quality assurance process. The purpose of testing software, or anything for that matter, is to validate that all of the other production processes worked correctly and the product meets the needs for which it was designed.

Quality assurance is about preventing problems from arising, and testing identifies what problems exist in the current product. A comprehensive set of quality-oriented processes and tools is needed to deliver anything of quality. QualityPoint represents the initial step in gaining control of the software delivery process. A robust software testing process supported with the right tools and expertise are the first steps to delivering applications that meets the business requirements.

QualityPoint was developed to provide a repeatable process that best balances cost and degree of testing. The origin of the QualityPoint name comes from the cost model that depicts the logical point for investing more time and effort to eliminate a defect and the cost of leaving the defect in place.

QualityPoint helps you to accomplish your quality goals by:

- Streamlining the testing life cycle
- Employing a risk-based process that accelerates the delivery process
- Delivering a repeatable process with reusable assets
- Supplying a centralized web-based repository
- Standardizing deliverables
- Deploying a systematic approach for evaluating test cases
- Setting testing priorities based on business drivers and their potential risks
- Defining metrics for determining when an application is ready for production

Compuware's internal Testing and Integration Center currently utilizes this proven methodology. The Testing and Integration Center is responsible for testing the integration between Compuware products and has been rated as satisfying SEI-CMM Level 3, a rating that was earned in March 2004.

QualityPoint Key Process Areas

The QualityPoint risk-based testing framework provides a way to ensure that the testing effort aligns with the most critical, highest priority business requirements. It links measures of an application's readiness for production back to those requirements through defined testing metrics. The QualityPoint framework identifies seven key process areas:

- Test planning
- Test case development
- Test environment preparation
- Test execution
- Test results analysis
- Management reporting
- Quality management

These process areas are supported by procedures and templates, which assist in creating a repeatable testing process that results in more reliable application quality. In QualityPoint, methodology can be viewed graphically in Figure 2.

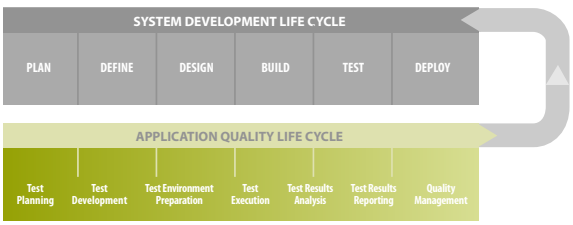


Figure 2: The Application Quality Life Cycle

The following section outlines the specific elements of each of the QualityPoint key process areas.

1. Test Planning

The test planning key process area is focused on two areas – test strategy and test planning. A test strategy should be created at the same time that application requirements are defined. A clearly defined test strategy enables the quality assurance team to provide a high level definition of the tests, locate and configure test-related hardware and software products, and coordinate the human resources required to complete the testing. The test plan provides the details that will be implemented to meet the objectives outlined in the test strategy.

Both the test strategy and test plan should be reviewed and approved by the test team, the application development team, all user groups and management. Once the test strategy has been completed, the test plan can be created. The test plan is a living document that evolves as application functions become more clearly defined.

The test planning process assists in establishing a process for developing efficient and effective test plans. The results from this KPA are a test plan for an initial project and a test plan model, including process flow, process narrative, and templates that walk you through the development of a test plan.

2. Test Case Development

The test case development key process area provides a method to create test cases and a way to easily reuse test cases. Since test case development is a part of an overall testing process, it is assumed that a testing methodology has been defined, test goals have been created, and test planning has been completed.
After the test plan has been defined, the test cases that will be used to test the application are created. The test plan dictates the type and number of test cases that will be needed. A test case identifies the specific input values that will be sent to the application, the procedures for applying those inputs, and the expected application values for the procedure being tested.

3. Test Environment Preparation

The test environment preparation key process area provides a framework for the preparation of the testing environment itself. This preparation includes a description of the environment, arranging operational support, and installation/restoration procedures. Only after these items are prepared can the test environment

be considered ready to accept input of test data, application installation, and the initiation of the actual test procedure.

This process is in place to ensure that tests are executed in an environment that closely mirrors the production environment where the application will be deployed. This is a critical point, because testing will not help determine application readiness if the environments are not similar.

There are many approaches to establishing and restoring test environments. The approach that is right for one application may not be best for another. However, there are many factors to consider. A few are listed here as a reference:

- Technical complexity of the application environment
- Number of projected test cycles
- Time and resources necessary to restore a test environment
- Number of test case data dependencies

4. Test Execution

Test Execution key process area applies test cases identified by the test plan and documents the results. This phase will address how the test gets applied to the application. This step of the process can range from very chaotic to very controlled and schedule driven. The problems experienced in test execution are usually attributed to not properly performing steps from earlier KPAs.

Additionally, several test execution cycles may be necessary to complete all the types of tests required for the application. For example, a test execution may be required for the functional testing of the application, and a separate test execution cycle may be required for the stress/volume testing of the same application. A complete and thorough test plan will identify this need and many of the test cases can be used for multiple test cycles. The secret to a controlled test case execution is comprehensive planning. Without an adequate test plan in place to control the entire test process, problems may be created for subsequent test cycles.

5. Test Results Analysis

The test results analysis key process area addresses the evaluation of test results and comparison of these results against the application acceptance criteria from the test plan. Analyzing test results can be a complex process and an unexpected result from test execution does not necessarily indicate a problem with the application being tested. There are four basic categories into which the test results can fall:

- Valid – No further action is required. Test case results require review to determine that the intent of the case matched the actual results.

- Application defect – During the course of the analysis, some defects are detected and regarded as being contained within the application. This may be referred to as an issue, problem, enhancement, etc. Utilize the test discrepancy logging procedures to accurately record and communicate any of the defects.

- Test case/test data defect – The detected error could be the result of an invalid test procedure or invalid test data. The data file that is used may be incorrect for the particular test.

- Application context defect – This defect may be as simple as the test started on the wrong screen. It could be that the wrong operating system was used.

When reviewing the test execution log, the focus is on the identification of errors and the entire flow of the test execution process. Follow the results analysis checklist and create additional documentation to record initial conclusions.

In addition, remember that test cases that are designed to get a negative reaction — and do — have passed,

because they did what they were expected to do (for example, a test case that checks to make sure that the appropriate errors occur if a duplicate social security number is input). This general rule can be applied to all test cases: if the results are what were expected, the test case has passed.

6. Management Reporting

The management reporting key process area is used to communicate the conclusions and recommendations of the test team to management, the software team, and the user community. It is important to not omit this stage of the test process. The management team should be able to make informed release decisions based on the number, severity and types of defects reported and cleared, the number of tests executed, and pass/fail rates. This information is critical for application quality professionals to understand the status of the application.

7. Quality Management

The quality management key process area encapsulates the concept that a testing project must be closed out. This close out includes necessary preparation for the next test project on this or any other application.

Included in quality management are activities such as cleaning up test cases, archiving old data, discarding old test cases, transitioning manual tests to automated tests, and conducting a review of the project and other items. For example, it may be appropriate to adjust testing priorities based on the level of defects in a function, and to determine which test definitions should become Regression tests in the test repository.

How QualityPoint Impacts Software Quality

QualityPoint maps to the system development lifecycle at key points to ensure that testing becomes an integral part of the life cycle. There are two key criteria to be considered to ensure the successful deployment of applications. The first is the utilization of software-testing processes that incorporate the verification and validation activities required to ensure that the software is correct when it is implemented. The second is having an application testing project management process and a software testing project team infrastructure that effectively identifies, manages, and controls the testing activities, as well as provides effective communications regarding application readiness throughout the entire testing effort. QualityPoint has a direct impact on these two key criteria for successful software project implementations.

It is important to note that the starting point is the clear documentation of business requirements that state what is important for a viable application and how those requirements will be substantiated by the testing effort. The definition of business and technical requirements — including application reliability and performance criteria — should be performed at the beginning of the development lifecycle.

Business requirements must be testable and traceable throughout product development stages. A testing process that does not explicitly support the business requirements can result in not meeting the organization's expectations.

Technical requirements describe the environment or computer system architecture in which the application must perform and integrate as well as the performance requirements of the system itself. Technical requirements, like business requirements, must also be testable and traceable throughout product development stages, and similar implications can result if the testing process does not explicitly support and link to the technical requirements. Resources spent working on test requirements that do not fulfill business or technical requirements resulting in an ineffective use of resources.

There are various types of tests. Each test type addresses a specific testing requirement. Test types include unit,

integration, functional, system, performance, beta, acceptance, and regression tests. Understanding the types of testing that are needed for each application is essential for several reasons:

- Ensuring that the types of testing support the business and technical requirements and is pertinent to the application under test; for example, there may not be reason to include unit testing if the application is a "vanilla" implementation of a third-party vendor software application.

- Ensuring that the activities for each test type and associated phase are included within the master test schedule; often, time is not adequately allotted to complete testing for each test type.

- Helping identify and plan for the environments and resources that are necessary to prepare for and execute each test type. Resources required may be significantly different to support different test type requirements.

Metrics are used in almost all disciplines as a basis of determining the effectiveness of the process. Within software correctness, metrics are used to objectively evaluate progress towards meeting the business and technical requirements and goals. Without objectivity, it is difficult to assess how much progress has been made, how much is left to do, and what areas are potential problems or risks that need to be addressed.

Software Quality and the Application Lifecycle

Deploying reliable applications requires the implementation of a repeatable process and a rigorous commitment to addressing quality throughout the entire development lifecycle. Earlier in the paper it was stressed that quality is the responsibility of the development team, the quality assurance team and the production/operations team. Furthermore, a rigorous approach to testing is required to achieve the benefits of software quality. The types of testing required includes:

- Code diagnostics designed to test for coding errors and performance bottlenecks.
- Unit testing designed to test a single application component, program, or subroutine.
- Code and test coverage analysis designed to ensure that an entire program has been tested.
- Integration testing designed to test groups of modules to ensure that data and control is passed properly between modules.
- Functional testing designed to test an application from the user's perspective.
- System Testing designed to test the application in its entirety and is designed to validate the functional requirements.
- Performance testing designed to test application and system characteristics including response time, resource consumption, throughput, and efficiency. Stress testing is a subset of performance testing.
- Beta testing designed to uncover errors that usually the end user is only able to find.
- Acceptance testing designed to replicate, as closely as possible, the final deployment of the application under test.
- Regression testing designed to ensure that no adverse changes are introduced to the application during maintenance changes, upgrades, or other changes.

Compuware Application Reliability Solution

Compuware delivers the QualityPoint methodology in the Compuware Application Reliability Solution (CARS™). CARS is a comprehensive solution that helps IT organizations deploy quality applications with confidence through the use of certified quality assurance experts, the QualityPoint testing methodology and mature technology. Professionals whose expertise is quality assurance and the management of business risk deliver the CARS solution. The methodology is a complete quality management process, which identifies seven key process areas that

are supported by procedures and templates. CARS also includes the Application Quality Workbench™ which provides a common view of quality, rich diagnostics, and a managed workflow to enforce process discipline.

Conclusion

Application failure continues to be a significant issue for IT organizations and business executives. Risk based testing provides a means for identifying areas of risk exposure. It ensures applications meet business objectives and prioritizes the most critical areas so that they are tested first. Risk based testing is an efficient, non-biased system for deciding what exactly has to be tested and breaks down the silos within the organization. By implementing a risk-based approach to addressing this challenge, IT organizations and mitigate the risk of deploying unreliable applications. ■

About the Authors

Elizabeth Maly is the Director of Marketing for Professional Services at Compuware Corporation. Prior to joining Compuware, Elizabeth held senior management positions with Ameritech Cellular, Rockwell and IBM. She holds degrees in Math and Chemistry from Smith College.

During her 15 years at Compuware, **Carolyn Barber** she has held management positions within Compuware's professional services organization and is now the Worldwide Sales Director for the Compuware Application Reliability Solution (CARS). Prior to joining Compuware, Carolyn held various management positions in manufacturing operations management at FMC and Federal Mogul.

Teaching the Old Dog

a lesson in code review from the open source community

Stephane Lussier
Macadamian Technologies

"Who does this guy think he is? My code WORKS. We're on a deadline here!" Luc, the grand old man of my software development team, stood framed in the doorway of my office with clenched fists. A vein just below his graying hairline pulsed wildly.

"Sit down before you have an aneurysm," I said.

"What's wrong?"

"Alexandre. Julliard. Rejected. My. Patch."

"Ah."

This sort of thing had happened a lot lately. It was 1998 and corporations were discovering Linux, including Macadamian Technologies' largest client. In the couple of weeks since my team had become the largest contributor to the Open Source Wine project at the request of our client, my developers had learned that Alexandre was very particular about the patches—the code meant to replace existing code—submitted to the source tree.

Wine is not an emulator. That's what the self-referencing acronym Wine stands for. Instead, Wine is an Open Source implementation of the Windows API on top of X-Windows and Unix. It's a layer of compatibility that allows native Windows programs to run on an X-Windows or Unix machine. With our contribution filling in some necessary gaps, our client planned to include Wine with their product—slightly modified—letting them put out a Linux version in nine months, at a reasonable cost.

"Can't we replace him as committer?"

"Not really." I kept an even tone with my colleague, a close eye on that pulsing vein, and my finger on the "9" of 911. "He's been working on Wine for four years, since '94. This is just a new way of doing things. We have to put up with it for this project."

"And we can dump this single committer nonsense after that?"

"We never have to do it again. I promise."

Luc slumped back to his cubicle to fix his patch to Alexandre's standard, muttering something about how it was going to be a long cycle.

Developers are a strange bunch of people. They don't like change. Actually, that's not so strange. Lots of people don't like change. That's so common they have a saying for it—you can't teach an old dog new tricks.

Now, my team wasn't exactly a bunch of old dogs. In fact, it's company policy that we, as an outsourcing partner, adapt to the way our client's development team works. So we were used to encountering new ways of doing things.

But we hadn't worked on any Open Source projects before. I hadn't expected an Open Source project to be quite so rigidly controlled: I'd imagined the exact opposite. Unadulterated chaos. I'd pictured myself having to wade through reams of crap code trying to identify useful features, like separating a bowl of party mix into its core components.

Instead, I — and my team — had encountered a concrete system as mature as any I'd ever seen.

The main form of communication between developers on the Wine project was their email mailing list. With development spread out over the globe, it made for efficient communication and links developers together. Despite members being so widespread, there was still a feeling of teamwork.

The problem came in when it was time to submit a patch to the mailing list…

The problem came in when it was time to submit a patch to the mailing list. All patches were submitted to the entire mailing list, which had the great benefit of keeping everyone informed of what was going on in other areas of the project. But the flip side of the code being so public was that… well, the code was PUBLIC.

We'd tried having code review meetings before. Besides making the team fat from all the donuts, we encountered some problems. We found that the developer who presents the code, showing the features and defining the logic behind development, tends to control the pace and flow of the meeting. Whenever someone brought up a suggestion, instead of the coder taking it under advisement and learning from it, they would get defensive and justify their logic. Usually at great length.

Also, the reviewers didn't have the kind of time they needed to get into the details of the code, to analyze how it works. The code just went by too fast.

So code review meetings hadn't been popular. As the cycle progressed, it was just too easy to dump the practice as taking a lot of time for not a lot of benefit.

But on Wine, code review was an irreversible part of the process. There was no way of getting over it, around it, or under it. Every person on the Wine mailing list had a chance to read every patch before it was committed to the source tree. Developers got religious about checking other people's code, mostly because no one wanted someone else to submit a patch with a bug in it. Here was a process—code review—that commercial software developers had made into something painful, but the Open Source Community was doing because they believed in it. Something professional developers made every effort to get out of, that Open Source developers where doing even though it was optional.

If you ever have a desire to take up a time-consuming and thankless task, consider becoming the "committer" on a single committer software project. A committer decides what patches go into the source control. So even if no one else reviews code submitted, the committer does. The committer also adds new features (You want to bet everyone looks at his patches!) and removes ones that need to go. And he enforces the coding conventions.

Alexandre Julliard is the "committer" on the Wine project. He's the person—the only person—who adds patches to the source tree. And my development team felt like he was rejecting their patches for the heck of

it. "Doesn't follow coding conventions"—rejected. "Not the way to go. Change this at the user interface level instead of so close to the engine"—rejected. Oh sure, after the developers had fixed their issues, the patch would go in. But that wasn't the point.

When I thought about it, what Alexandre was doing made perfect sense. It bordered on genius. Of course you want to come down hard on people when they first join the team. If you train people to follow the coding conventions right away, they learn to do it on their own.

A couple months later, sitting in my office with my first coffee of the day, I realized something was wrong. Well, not wrong, exactly. Something was very right, in fact: No one had complained about Alexandre—or any of the other developers on the Wine mailing list—in a week.

When Luc arrived, I gave him a few minutes to pour himself a cup of ambition before heading to his cubicle.

He offered me a seat in his guest chair, after removing from it a couple of thick bibles on software architecture and a sheaf of use cases.

"How are things going on Wine?" I asked.

"Aren't you on the mailing list?"

I shrugged. "Yeah. But how are things going?"

"They're alright."

I checked the forehead vein. No pulse to be seen. "And Alexandre?"

"Hey, I got my last patch accepted on the first pass." I swear his chest puffed out a couple of inches.

"Uhm. Cool?"

"Sure. And you know something?" His voice fell to a cubicle whisper. "The other developers aren't so bad. I've learned a few things. Picked up some tricks. Just the other day I saw Marianne's code for List View support. She did this cool thing with sorting the columns that will work in my Tree View patch."

"Good trick."

"They learn stuff from me, naturally. I just showed Brett how to fix his Z ordering problem. We're a team now, so I thought I'd help the kid out." He sipped his coffee. "Also found two errors which he has to fix before Alexandre will admit the code."

"So it's not so bad, then."

"Hey, don't get me wrong, it was hard to start, and I still don't like some of their coding conventions." He shrugged. "But the process has some merit. It's been an interesting experiment."

The kickoff meeting for our next project was going about the same as usual. Marie, Theo, Salvador, and I munched Boston Crèmes and talked about the usual stuff: Architecture, communicating with the client, status reports. But unlike usual, Luc didn't have anything to say. In fact, he had a distant, far-off gleam in his eye.

"Luc, what do you think?" I asked.

He shrugged. He had everyone's attention—with his experience, there wasn't a person at the table who hadn't gone to him with a difficult coding problem.

"I think we produced some of our best code ever on the Wine project."

There were nods around the table. I'd noticed the same thing.

"Didn't take us any longer either," said Marie, one of the newer developers. "Not much longer anyway."

Theo agreed. "Holding all the code to the same standard should make it easier for someone to come along later and make sense of it."

"You learn to be more careful when you're coding—find the errors before someone else does." Marie said.

And before they get into the source control, I added mentally.

"I don't know." Salvador tapped his finger on the board table. "Sometimes, I just didn't find the comments on Wine helpful."

Luc's eyes sparkled with amusement. "Like when a new developer says something useless, like 'There's a problem with this patch'? That doesn't help you find the bug."

Marie piped up. "So let's say that when you report a bug, you have to state the specific problem, where to find it in the code, and a suggestion for a solution, if you have one."

"But just make the suggestion, don't implement the fix yourself." Salvador said.

There was a general murmur of agreement. I spoke up. "The guy who submits the code should be responsible for fixing it." Otherwise, how will you learn from your mistakes?

"There might be something to this single committer stuff. We could do a mailing list for this project." Luc suggested. "Someone could be the committer."

"Are you volunteering?" I tried not to smile, remembering that pulsing vein in his forehead.

"Suppose so."

Salvador piped up. "We could use our code review checklist."

I had another contribution. "I don't really want to see Luc spending all his time just reviewing other people's code." He was one of my best developers, after all. Some of the others could learn a lot from him, but I still needed him to do his share and not devote all his time to being a mentor. A couple of faces around the table fell, thinking I was vetoing the project. "What if you sent each patch to someone else, but not necessarily Luc? Of course it should go to the entire mailing list, but assign special responsibility to one other team member. They have to review it. Everyone has different areas of expertise, so if you choose the best person to review that particular patch, no one person will get overloaded."

Everyone has different areas of expertise, so if you choose the best person to review that particular patch, no one person will get overloaded.

General nods, and a look of relief from Luc. He hadn't anticipated that problem, but could see that he'd dodged a bullet.

"Maybe we should try it." Luc said.

I sipped the last of my coffee. Time for a refill. "Maybe we should."

"Blah blah blah committer blah blah." My ears perked up at the sound of Luc's voice. It isn't my habit to eavesdrop, but the door to the meeting of Luc's new team was open. I slowed my steps.

"The experiment was a success," said Luc. "It really worked. Sure, there were a few things we had to work

out along the way, things like ideal size for a patch..."

"How big should they be?" asked Martin.

"They have to take less than an hour to review. Fifteen minutes is the best." I smiled to myself. I'd known that the developers had really bought in to the single committer model when they apologized for submitting a big patch. "And we finally decided that one committer had too much to do and made Maria a committer as well."

I moved on. With the developers evangelizing the process, code review was going to spread through the company without help from me.

Guess the old dog can learn one or two new tricks. ■

Stephane Lussier is a development manager at Macadamian Technologies, a software consulting firm in Ottawa, Ontario, Canada. He manages desktop application and driver development projects for customers like Nortel, Groove, Corel and GraphOn. Stephane holds a Baccalaureate in Computer Science from the University of Sherbrooke, Quebec, Canada.

Email Stephane **at stephane@macadamian.com.**

The Productivity Equation (Part I)

Frédéric Boulanger

Critical Path is a regular column dedicated to encouraging best practices in software development. In each issue of Software Quality, we will bring you a discussion of the issues and ideas that affect software managers every day.

The Critical Path

On the scale of 0 to 5, rate your worst team member.

How about -5?

In most cases, it takes a team to deliver software. And the productivity of any development team relates directly to the productivity of the individuals on that team. In the best situations, each person has a symbiotic relationship with the rest of the team members, contributing positive work, attitude, and ideas to the common effort.

But situations are rarely ideal.

The best team members do exactly what I've described above. They are natural +5s.

So what about the worst team members? Why are they -5s, and not merely zeros?

A zero is a team member who doesn't contribute. But they don't hurt either. Zeros are largely theoretical, because a person who did nothing wouldn't last in the working world. 99% of people either move the project forward or set it back.

People on the low end of the scale aren't zeros because a zero wouldn't affect the team productivity equation at all. And bad team members are worse than non-productive, they are counter-productive.
Identifying -5s

Simply put, counter-productive team members drag the development effort down. They produce low quality code and they take more time than usual to deliver.

Worse still, -5s require support from more productive team members. They need help to design and implement their code, which reduces the productivity of +5s. New hires always need support to integrate into the team and to understand the role of their code in the larger scheme of the project, but -5s never get to the place where they can function on their own. They never stop creating below standard code that is at best careless and awkward, and at worst breaks the build.

Questions to Identify a -5

- Does he break the build twice as often as your other team members?
- Do other team members complain that they spend too much time helping him or debugging his code?
- Does he constantly interrupt you or your team with questions when he could find the answers online or in the training documents?

If you answered yes to any of these questions, you probably have a -5 messing up your productivity equation.

Avoiding -5s

Two little words: Hiring process.

I can't say enough about having stringent quality standards in your hiring process. Your company's future depends on it. If you don't hire smart, you compromise every project you will ever do, costing yourself time and money, and demotivating the entire team, who will have to deal with the consequences of a counter-productive employee.

It isn't enough to merely look for the spark in a potential employee's eyes, although that needs to be there, too.

Finding the right employees has to be a repeatable, systematic process.

The interview process must verify that the candidate:

- Understands what they've worked on in the past;
- Has a coping strategy for what they don't know and makes an effort to learn; and
- Will fit with the team.

The interview process must verify that the team:

- Will accept the new member; and
- Will benefit from the addition of this new member.

Creating an objective interview process that takes the emotion out of the decision will improve your team's productivity equation.

Here are a couple of great books that will help you:

96 Great Interview Questions to Ask Before You Hire
– Paul Falcone

High Impact Hiring
– Dr. Del J. Still

Next time in the Critical Path, I'm going to talk more about team composition and how to ensure you hire +5s. ■

Frédéric Boulanger is the President of Macadamian Technologies, a software engineering and consulting firm that specializes in helping technology firms meet tough deadlines and release products faster. The Critical Path is also delivered monthly (free) to an email inbox near you.

Email Frédéric at **frederic@macadamian.com.**

Subscribe to The Critical Path at:
www.macadamian.com/columnmgmtsubscribe.html

Motivating Extreme Development

Dana Smith

In the past three years, I had great success taking the Extreme Programming approach for three of my projects. In this article, I'll share some of the experiences my team had; hopefully, those of you who are trying Extreme Programming (or are thinking about it) will benefit from what we learned.

Using XP is hard. Contrary to popular belief, it requires a high level of discipline and a deep understanding of the intent of the practices. Having had our success, my team and I dissected what went right and in doing so, we came to an understanding of Extreme Programming (XP) and the motives behind the practices that comprise it.

We found that the twelve XP practices can be organized as a hierarchy of goals. Of primary importance is the root goal — the purpose the entire hierarchy is meant to support. As we came to understand it, XP is aimed at creating quality software. We also came to understand that the term 'quality' is remarkably difficult to define. Everyone had his own idea of what 'high quality' software is. That proved to be the revelation itself: Quality is subjective. The question then became "if quality is subjective, who determines if a piece of software is of high or low quality?" Knowing that our customers write our paychecks, we felt safe in the assertion that it's the opinion of our customers that matters.

Now while the customer ultimately funds our work, it's our managers who actually keep us around to do it; we decided we would do well to keep them happy, too. Plenty of other people in the organization are affected by the success (or lack of success) of a product: business and market analysts, sales personnel, and the managers above all these people—up to and including the CEO himself. We include all these people in the group that decides the quality of a piece of software.

So our definition of high quality software is software that meets the needs and expectations of the stakeholders. We refer to the needs and expectations as requirements.

The group of stakeholders is typically too large to be as intimately involved in the project as required. On our projects, we had a single person who acted as a proxy for the group of stakeholders, representing their position in the day-to-day affairs of the project. While the individual chosen as proxy is ideally an expert in the product domain, the most important thing is that on an ongoing basis they are able to bring the group of stakeholders to consensus on the product requirements. Throughout the rest of the article, whenever I refer to the stakeholders it's safe to substitute 'stakeholder proxy.' I want to keep the focus on the stakeholders clear, so I won't muddy the waters by appending 'proxy' every time, even though that would be more accurate.

The strategy employed in XP to create quality by meeting stakeholder requirements has two

components: Improve project visibility and Increase discovery time.

Improving Project Visibility

In my experience, the very worst thing you can do to stakeholders is surprise them. Stakeholders need to know as early as possible if their expectations are not going to be met so they can adjust any contingent plans.

Sadly, it's very difficult for the project team to be aware of all the expectations of all the stakeholders, or even to fully understand those they are aware of. Regardless of how much time is spent planning and documenting project requirements, things always get missed; requirements also tend to change over time and the appropriate people are often not informed. Improving project visibility is meant to mitigate these problems. By ensuring that stakeholders are fully aware of the state of the project, the team can rely on the stakeholders to inform them when progress is not meeting expectations.

There are three practices that contribute to improving project visibility: Planning Game, Frequent Releases and Constant Integration

The Planning Game

The Planning Game is a meeting that's held once just before work begins and then recurs periodically for the duration of the project. At this meeting, the engineering team works with the stakeholders to create and prioritize the list of tasks necessary to meet the requirements, and to generate estimates for how long each task will take. Associated with each estimate is a statement of risk. Priority is determined primarily by the value each task has for the stakeholders; tasks with higher value are done first. Priority is adjusted based on the risk: It's wise to do riskier tasks first since they are likely to cause changes in the schedule.

The outcome of the Planning Game is the project plan. New information comes to light over the course of a project, bringing about changes to the plan: The engineering team makes discoveries that change their estimates and task value changes based on a variety of factors. The value of having the plan is not to follow it, but to measure when you've deviated from it so that appropriate adjustments can be made.

By working with the stakeholders to create and maintain the plan, we improve their overall view of the project. We are more certain that we're actually doing what the stakeholders expect and that any decisions to change the plan involve their input. They also quickly become aware of any unexpected changes to the plan, and can react accordingly.

The most important outcome of the Planning Game is clarity in the role played by each individual, allowing them to be more effective. As a good example, I once ran a day-long planning session. As each task was read, the stakeholder proxy explained the details and importance of the task and the engineering team gave their estimate. To start with, Paul (our proxy) was appending his task descriptions with statements like "So that's like, what – two days?" After conferring amongst themselves, the developers would come back with an estimate that was totally different from

what Paul expected. Very quickly, he realized that he had no idea how long a task would take. The rest of the team benefited from the awareness that we really had no idea what was important to the stakeholders: The priorities assigned to many of the tasks came as a surprise to us.

Frequently Releasing Working Code to the Stakeholders

By frequently releasing the product to the stakeholders (every two to three weeks), we provide an excellent measure of progress. At each release, stakeholders can see increased functionality. Conveniently, as a consequence of the Planning Game, it's the most important functionality that's been added.

This has an implication for the way we define our tasks. Because we need to increase the functionality of the product in short bursts, we need to define our tasks in user terms; the tasks — or User Stories — need to define what the product does, not what it is. Making an analogy to describing a car illustrates this nicely. If we define what a car is we say things like "A car has doors and wheels with tires, a windshield" and so forth. To describe a car by what it does we say "I can open the door and get in. I can turn the steering wheel to move the front tires. I can push a button and the window rolls down." By defining our tasks in this second manner we ensure that every task we complete increases the functionality of the product in a useful way.

In support of frequent releases, part of the Planning Game is to take the user stories and put them into 'iterations.' Iterations are groups of stories that, in total, are estimated to take two or three weeks for the team to complete. At the end of each iteration, the product is released to the stakeholders. If a task is estimated to take longer than two or three weeks it must be broken down until the constituent parts are shorter than the iteration length.

When we don't take this approach, our progress is invisible to the stakeholders. From their perspective, very little of the product works until very near the end of the project. This seriously hampers their ability to determine if the product meets their requirements. As a consequence of the many changes that happen over time, aspects of the product will likely surprise them when they finally see it, and surprises are bad.

I once switched from managing a project where I had this high level of visibility to one where the tasks were traditionally defined: monolithic, months long, and with the pay-off only at the very end. To describe the feeling, it was like driving down the freeway at a good speed and having someone reach up from the back seat, cover my eyes, and start shouting "Faster! Faster! Faster!"

Constantly Integrating New Code

In order to have a release for the stakeholders at the end of every iteration, the engineering team needs a fine-grained view of progress within the iteration, otherwise, we'd never know if we were going to have something that we could release. To accomplish this, developers follow the practice of checking code in frequently - every hour or so - and maintaining a fully operational product at all times.

As a result of these rules, the product's functionality grows in an interesting way. The addition of text support to one of my projects is a good illustration. At first, we modified the code to create space for the text on the screen. The stakeholders (and the engineering team) could see the space visually. Next, we represented every text character with a rectangle. Again, everyone could see the rectangles where the characters were supposed to be. They could witness the progression toward the desired endpoint. After that, we replaced each box with an 'e'. The next step was to put in the actual characters. Finally, the spacing around the characters was adjusted properly.

You can see how at each successive step the application behaved in a way that was more useful, and closer to the ultimate intent. The feedback and sense of progress generated by working in this manner is invaluable. It means that testing can begin sooner, and it means that developers can rest assured that any work they're doing is compatible with the work of their teammates. Most importantly it allows us to determine whether or not we're on schedule.

This feedback can be quantified. When finished, each task has an estimated time and an actual time. It's easy to keep a running tally of the estimated and actual times for all tasks completed during an iteration. You can calculate the ratio of total estimated to total actual time at any point. This is known as the 'project velocity.' The velocity tells you how close you are to the schedule (based on the estimates). A velocity of 1.0 means that you are meeting the schedule exactly; a velocity below 1.0 indicates you are slower than planned and a velocity greater than 1.0 means that you're ahead of schedule.

The inverse of velocity, load, is the ratio of actual to estimated time. The end of an iteration is a convenient time to make and record this calculation, since it is a fixed point where no work is under way. The load from completed iterations is used as a multiplier on the estimates in upcoming iterations. This compensates for any errors in the estimates.

Load (and hence velocity) shouldn't ever be 1.0. It's easier and more accurate to have the developers estimate the work assuming no interruptions and then let the effect of the interruptions average out over time and show up in the load factor. Load factors are typically around 1.8.

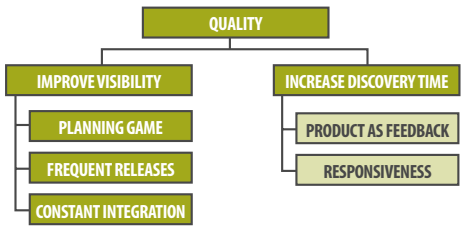
Increasing Discovery Time

A problem we commonly face is that stakeholders have a difficult time deciding what they want. In traditional projects, a big up front design is created in an attempt to force this decision. The intent is to improve predictability by making all the decisions early on. Once the uncertainty is out of the way, it's expected that the remaining work will progress as planned.

Always though, the design evolves and changes over the course of the project. The stakeholders and engineering team learn more about the product and make discoveries that demand a reaction. Typically, change and evolution meet resistance because they require changes to the plan, which in turn decreases predictability.

Unfortunately, this often means the stakeholders can't get exactly what they want; according to our definition, this is low quality. To combat this, a number of XP practices are designed to allow stakeholders to make decisions right at the last minute. Moreover, the practices urge stakeholders to use the extra time to handle the product and discover its strengths and shortcomings for themselves. The resulting changes to the plan are integrated at the regular planning meetings with the result that the product more closely meets the stakeholders' needs.

There are two components to the strategy for increasing discovery time: using the product as feedback and improving engineering responsiveness.



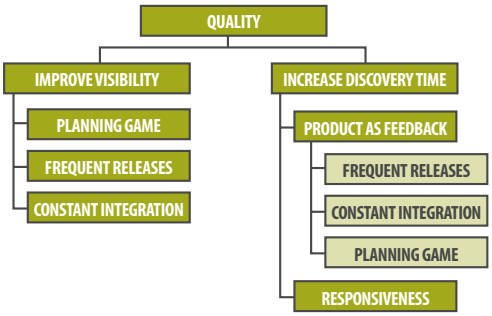
Using Development as Feedback for Discovery

In his February 20, 2004 ComputerWorld article "Iterative vs. waterfall software development: Why don't companies get it?" Bill Walton provided a scale to describe what a stakeholder is capable of doing

with a product when asked to provide feedback on it:

Imagine -> See -> Handle -> Use

The further to the right the product is, the higher the value of the feedback. In XP, three practices are aimed at eliciting feedback on a product that stakeholders can use. Actually, they are the same three practices I've already discussed: **Frequent Releases, Constant Integration** and **The Planning Game.** I'll discuss them again in this new context.



Frequently Release Working Code to the Stakeholders

For purposes of discovery, getting a working product into the hands of the stakeholders is vital. With something concrete to play with, they get a clear picture of how the product will suit their needs. As a consequence, they have a better idea of where to steer the product as it moves on from any given release.

Paul, the stakeholder proxy, once requested that small windows appear at certain times, to present users with options. After the stakeholder group used the application for a while, some concerns were raised about the value of these pop-up windows: Research showed that the extra options detracted from the usability of the product! We took the windows out. Without the extended amount of time that was allowed for feedback we would never have made this discovery and the product would have suffered as a result.

Constantly Integrate New Code

Integrating the changes from individual developers at the end of an iteration is a recipe for disaster. The longer people work independently, the more their work drifts apart and the harder it is to reconcile any differences when the time comes to integrate their efforts for a release.

By adding new code to the product frequently, we ensure that incompatibilities are rectified early. The direction of the every team member's code is corrected so often that significant drift can't happen and we are never in danger of encountering integration difficulties in the last few days of the iteration.

I have a good example, to give you a sense of how quickly isolated code can become prohibitively difficult to integrate. One of my projects served two purposes: It was a stand-alone application, used by customers, and it was also an engine for another product. At one point these two purposes generated conflicting requirements, so we created two copies of the code. Our intent was to merge them together again within a month. As it turned out, there was never a time when the benefit of merging the code was worth the cost. Eight months later, we still had two streams of code to manage, and no plan for when they should be joined.

The Planning Game

The stakeholders' involvement in planning gives them control over what gets done and in what order it happens. Consequently, they can correct or adapt the schedule at will to ensure that the work being done is actually progressing toward the desired result.

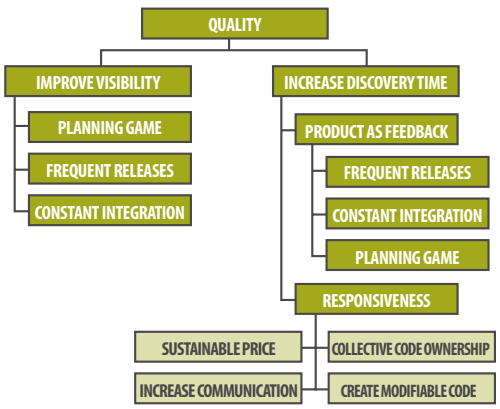
Because all the work on the schedule has an estimate and risk associated with it, stakeholders can very easily determine the cost of any changes they make. Adding, removing and relocating stories on the list clearly affects what can get done in a given time frame. With

this information in hand, stakeholders can predict the effect of any changes on the end result.

Nearing the release date, one of my products had a split personality. Due to an inadvertent and undetected split in the opinions of two key stakeholders, we were developing the product to serve two purposes; some of the features we put in were aimed at one goal, others at the second. A month before we shipped, the entire stakeholder group was given an iteration release and was surprised to see certain features. When they questioned these additions, the difference of opinion became evident. The beauty of the situation came in the resolution. Because we had planned our last two iterations properly as a collection of user stories, and because we had generated estimates for all the work, correcting the problem was simply a matter of re-organizing the stories so that the last two iterations contained everything everyone wanted.

Increasing Engineering Responsiveness

The goal of using the product as feedback is to give the stakeholders opportunity to determine what they would like to change in the product. The engineering team needs to be prepared to act on those changes as soon as they are declared. There are four things we do to provide the engineering team with the agility necessary to react to stakeholder requests. Two of them (sustainable pace and collective code ownership) are practices. The other two (increase communication and create modifiable code) are goals in-and-of themselves.



Maintain a Sustainable Pace

In his paper "Overtime Hours: The Rule of Fifty" John M. Nevison summarizes the results of a number of studies showing that regardless of the number of hours spent working in a week, the average person will accomplish a maximum of only fifty hours worth of work. The extra time you spend working actually costs you in the long run, typically because you need to correct errors you make when you're over extended. Worse, as the weeks wear on people don't even get fifty hours of productivity: In the extreme, the number of productive hours drops to about thirty. In other words, they would accomplish more without the overtime!

The intent of the 'sustainable pace' practice is to force engineering teams to achieve a high level of productivity over the long term rather than having periods of extremely high activity followed by spans of near inactivity. The rule is that staff should work at whatever pace they feel they can sustain for an indefinite period of time.

I'd like to offer a word of caution here, from my own experience. It's very easy to be overworked without realizing it. There was one colleague with whom I had a terrific, professional relationship. More than that, I considered her a friend (and still do). During one of product cycle, we were approaching a milestone and had a team meeting to discuss what needed to be done. By the end of the meeting, she was looming over table talking to me with a stern voice and a red face (and being in excess of six feet, this made her somewhat imposing!) My knuckles were white from clutching the arms of my chair. We both felt frustrated in our efforts to get the other to see our point of view. The next day, I ran into her in the hall. She asked, "What happened yesterday? I'm never like that, and neither are you!" In the ensuing conversation we both realized that our home lives were particularly stressful at the time and that our work was suffering as a result. Even though the stress was enough to cause an argument between friends, neither of us realized we were in trouble until after the fact.

Collective Code Ownership

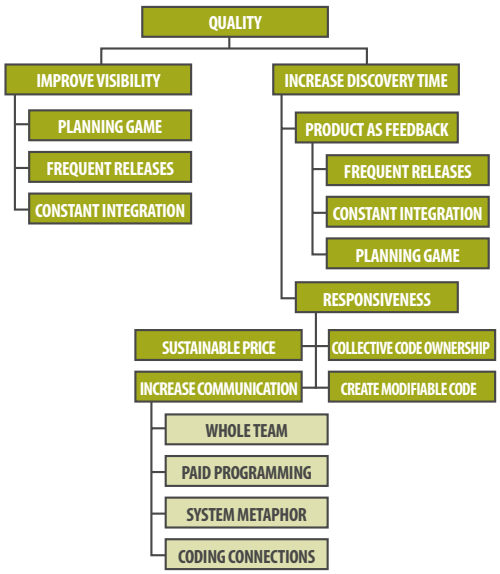
A 'truck number' is the number of people on your team that could be hit by a truck without affecting the team's ability to deliver the functionality currently on the table. A team that consists of a number of specialists, each owning an area of the code, has a truck number of one. If any of these people falls out of service for whatever reason, the project is in jeopardy.

A better approach is to have a team where each individual can take over any of the tasks currently in the queue. This way, no matter what happens to the individuals, the project's functionality is never compromised; the lack of people might slow you down, but it won't stop you dead.

This also has implications with respect to executing on the stakeholders' plan. The tasks in the queue are in priority order. If the team is made up of specialists, each person will have to pick and choose from the queue according to their expertise. On a team where there is collective code ownership, though, each person is able to take on the story at the top of the queue, thus ensuring that work is done in the order defined by the stakeholders.

Increase Communication

Ideally, the customer and the programmer are the same person. If this is the case, there cannot be any question of the requirements, or whether the product meets the needs of the customer. In reality, though, there are a number of layers between the people who build the product and those who need and use it. Maintaining the communication of customer needs through these various layers is absolutely vital if the product is to meet those needs in the end. There are four XP practices that are meant to maintain this communication. Whole Team supports communication of requirements to engineering. Pair Programming maintains high levels of communication within the engineering team. A System Metaphor is a tool for ensuring clarity of expression among the stakeholder group and Coding Standards facilitate communication using the code as a medium.



Whole Team

Based on the premise that requirements are going to change significantly over the course of the project, there isn't much point in spending a lot of time specifying them at the beginning. Instead, user stories (which take the place of requirement specifications in an XP project) consist of just a few lines describing what the author wants. There is only enough detail for the developers to make a reasonable time estimate for the story. Writing stories is actually one of the most difficult parts of XP, and to give it proper attention would require more space than I have here.

Given the lack of detail in a story, the developers need access to the author to gain clarification and detail when the work is actually underway: Providing that information is the role of a stakeholder (or the stakeholder proxy). That is the concept behind the Whole Team practice — that everyone with a stake in the project is available to add clarity when necessary.

On one product, our stakeholder proxy spent about seventy five percent of his time fielding questions from the development and QA teams. To give an idea of how valuable this approach is, there was a story that turned out to be far more difficult that we thought. Our estimate was for three days and after we started it, we didn't think we could finish it in ten. Examining the story (and the details we got from the proxy) we found that it didn't make sense to us — it broke a few user interface standards and seemed clumsy. We also realized that by changing the story so that the product would work the way we'd expect it to, we could have the task done in two days: less than the original estimate! When we showed our customer representative how we thought it should work, he said, "You can do that? That's what I wanted in the first place, but I didn't think you could do it. I came up with this other thing instead."

Pair Programming

In a rally car, there is one driver and one navigator. The driver performs the technical work; in order to go as fast as possible, the driver must concentrate fully on the mechanics of driving the car. Navigators see the bigger picture; they make sure that the driver is able to concentrate, and they warn the driver of obstacles so that the car stays on the road. Both tasks require full attention, so the car can go much faster when the driver isn't distracted with navigation. Similarly, programmers can go faster in a pair. The one writing the code doesn't have to worry about the 'road hazards' – his partner is doing that for him – so he can just program.

The level of communication that this generates between the programmers is invaluable. Some of the benefits include constant peer-review of the code, lack of distractions, and increased collective code ownership and learning. The greatest benefit, though, is in the quality of the code. We found that code written this way is much more stable and has significantly fewer defects. In fact, the quality is so much higher that it offsets the extra time you spend by having two people work together on one task.

I expected people to pair on a task by task basis. What happened, though, is people paired based on availability. The first two developers in to work would pair up. When one of them left for a break, a third person would step in. When the first guy returned from his break, he'd join another team. It worked remarkably well. A team of five working in this fashion had a pace about ten times faster than average. On top of that, the code was terrific: very stable and extremely resilient. We changed it frequently and it always worked.

System Metaphor

A system metaphor is meant to express the nature of the project in non-technical terms. For example, the metaphor for my first XP project was a desktop. You have pens and post-it notes on your desktop and you can use them to mark up pictures, to send your thoughts along to coworkers. If you understand that metaphor, with all its implications, you understand our product.

I've found the power of the metaphor is in the implications. By considering the metaphor rather than the product itself, I can often predict problems or identify strengths and weaknesses in our work.

The metaphor is a terrific artifact to use as a conversation piece when discussing the product with stakeholders.

Coding Standards

The code is a medium for communication between developers. A computer language is just that — a language. Another developer who reads it can extract the ideas one developer expresses in his code. It's important for the code to have a consistent look and feel, to make it easy to understand.

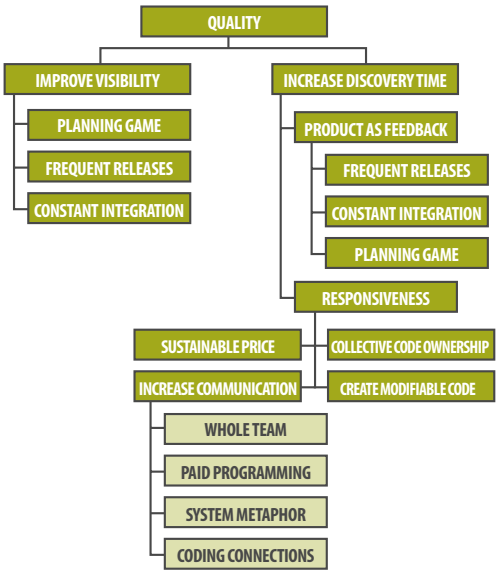
Apparently, Einstein had five identical suits, one for each day of the workweek. He explained that he didn't

want to waste mental effort deciding what to wear - he had more important things to think about. Similarly, developers shouldn't need to worry about how to dress up their code. A set of coding standards ensures that code remains a valuable mode of communication without costing time.

Creating Modifiable Code

Finally, we come to creating modifiable code. If the customer requests a change, the development team needs to respond to that request. Typically, teams hesitate to make changes to code because of the likelihood that defects will be added with the changes. This is truly crippling in terms of agility.

Software is exactly that — malleable. To artificially restrict it is to compromise the very strength of the medium. If developers could fearlessly change and evolve code, customers would end up with what they want much more frequently. The following three practices are intended to achieve that end.



Simple Design

Developers have a tendency (in fact, they have often been trained) to try to predict future uses of their code, and to build designs that anticipate customer desires. In fact, it is nearly impossible to predict any future need. Consequently, much effort is lost attempting to allow for situations that never actually come to pass.

An alternate approach is to take the attitude of "You aren't going to need it." According to this philosophy, you do the minimum amount of work necessary to accomplish the task at hand. When working with a queue of stories, this doesn't mean that you ignore future needs, only that you account for the things you know are coming down the pipe, and ignore anything else.

As a consequence of this approach, code grows organically toward the customer's need. It stays as simple as possible so it is much easier to understand and maintain without risking the introduction of defects.

Basically, there can't be bugs in code you don't write, so don't write any code you don't need.

This approach is part of the reason that communication is so high between the developers and the stakeholders. As we're working, we're constantly questioning the stakeholders to find out what they really want, so that we're not doing any extra work.

Refactoring

The Simple Design practice is aimed at starting things off in the simplest way. Refactoring is aimed at keeping things as simple as possible on an ongoing basis. As understanding of the product grows, past mistakes will become obvious. Refactoring is the process of going back to the original code and correcting those mistakes, rather than compensating for them in the new code.

In one of my products, we knew we needed to support Document Type Definitions. After a while, we discovered that we also needed support for Schemas. As we were adding schema support, we realized that we were reproducing a lot of the functionality of the DTD code. Rather than duplicating the effort, we created an abstraction of the common functionality and re-wrote the DTD code to use that abstraction, simplifying it in the process.

Constant Testing

One of the biggest issues developers face is that any code change can have unpredictable effects. These side effects often go unnoticed for long periods of time and are discovered only when QA, or worse yet the customer, has the opportunity to use the application. Obviously, it would be much less costly if the developer could be sure that what they've changed has had no negative side effects, and that is the intent of constantly testing the product.

There are two relevant types of testing: acceptance and unit. Acceptance tests express the customer's requirements in a measurable way. For each story, there is a suite of acceptance tests to ensure things are working the way the customer expects. Ideally, these tests are executable and the developers can run them every time they add code.

Unit tests perform the same purpose, but rather than being an expression of the customer's requirements, they express the developer's intent when writing a particular function or module. Such tests can be seen as tripwires: When the code changes and no longer satisfies the original developer's intent, tests fail and it is obvious that something needs to change. It may be that the test itself needs to be changed, or perhaps an error was made in the code that was added or changed. Either way, the developer knows that something unexpected has happened.

As in the case of acceptance tests, unit tests should be automated so that the developers can run them at any time. In fact, it's best if they are part of the build process and cause a build break if one fails. With a suite of such tests, each developer can rest assured that the changes they've made don't cause unexpected behaviour anywhere else in the product.

This practice supports refactoring. The tests give developers insurance that any changes they make when refactoring don't have unforeseen effects somewhere else in the product. By having these two suites of tests, there is very little risk that that state of the product will ever regress. Of course, that assumes that all expectations, requirements and assumptions are tested so care is needed in writing the tests.

An interesting additional benefit to having the tests is that they act as documentation. When looking at a piece of code, reading the tests gives a developer a good understanding of why the code was written, which in turn helps them understand the code itself.

If asked to identify the most important practice in XP, I'd pick this one. Having tests gives the engineering team the courage to change the product in response to stakeholder requests. It's the tests that make us agile. If I ever find myself working on a project that has no tests, my first order of business will be to set up an automated testing system, and the following example will show you why.

My largest project did not originally have an open programming interface, so third parties couldn't integrate with it. To add a programming interface, we had to rewrite huge swaths of code in the very core of the product. Thankfully, we had a test suite of more than six thousand tests that ran every night. Because of this test suite, we had the courage to change some very fundamental aspects of the product. We knew we were done when all the tests passed again. Without that test suite, we would never have been able to justify the risk involved in creating a programming interface and

we would have missed many opportunities to make money with the product. We so valued the tests that one of the developers spent a month writing code to enable further automated testing. I have no doubt what-so-ever that it was worth the time.

Using these practices in our work has served us well. One of my projects was a component in a larger suite: It was the first to turn a profit. One of the others never actually shipped, but it too was a success. Our efforts helped the business analysts understand the product domain well enough that they could see we wouldn't make money with a product. The third project is the one I consider most successful: It won a Best of Comdex 2002 award. I hope that the experiences I've shared will help you find at least the same level of success.

A final thought: When you're doing it right, Extreme Programming is exhilarating. It makes work fun. When the chips were down in past projects, I saw teams crack under pressure. Things would descend into chaos and it wasn't always the most important work that got done. I imagine a giant fist squishing a lump of dough. On Extreme projects, though, it felt like the team was on rails. The higher the pressure, the faster we'd go, but always in the right direction, toward the right destination.

It's really an awesome feeling. ■

References

Anderson, David J., *Agile Management for Software Engineering*, Prentice Hall, 2004.

Beck, Kent, *Extreme Programming Explained - Embrace Change*, Addison Wesley, 2000.

Beck, Kent, and Fowler, Martin, *Planning Extreme Programming*, Addison Wesley, 2001.

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

Extreme Programming wiki-wiki web, http://c2.com/cgi/wiki?ExtremeProgramming

Goldratt, Eliyahu M., and Cox, Jeff, *The Goal, 2nd Rev Ed.*, North River Press, 1992.

Nevison, John M., (December 1997). *Overtime Hours: The Rule of Fifty*, Concord MA: Oak Assosciates.

World Wide Web Consortium, *Extensible Markup Language (XML) 1.1*, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, John Cowan editors, February 2004, http://www.w3.org/TR/2004/REC-xml11-20040204/

World Wide Web Consortium, *XML Schema Part 1: Structures*, Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn editors, May 2001, http://www.w3.org/TR/xmlschema-1/

Dana's first four years at Corel were spent as a software developer. After moving into a management role, he helped define Corel's software development methodology. His style evolved to include Extreme Programming, which he used to rescue two troubled projects; one, Grafigo, took a Best of Comdex award in 2002.

Get CSTE/CSQA Certified...

OSQA is hosting the following exams for CSTE/CSQA Certification:

Examination Date	Candidate Application Deadline
Saturday, December 4, 2004	October 4, 2004
Saturday, March 12, 2005	January 11, 2005
Saturday, June 18, 2005.....	April 18, 2005
Saturday, September 17, 2005	July 18, 2005
Saturday, December 3, 2005	October 3, 2005

For more information, and registration forms, see **www.softwarecertifications.com**.

The Good News/Bad News of Software Testing

Steven Barry

So your team has decided to start a testing process: That's the good news.

You're probably planning an infallible and uncompromising process that covers every aspect of project work from initial project idea to uninstall from a user's computer. Here's the bad news: That isn't going to happen. You're going to have to cut corners in order to meet schedules and the pressure for release.

But the best news of all is that there are some things you can do to build Quality processes and practices into your development cycle that are relatively painless, inexpensive, and easy to sell to The Powers that Be.

1. Consult with the project team to write a broad Test Strategy

A Test Strategy broadly describes your approach to testing. It helps you address testing and quality concerns early in the development lifecycle.

Involve input from stakeholders (Project management, development, and users) to allow clear communication of your project testing approach.

Typically the Test Strategy includes at least:

Project type and scope: This gives you the "footprint" of the project. The project type (new development, enhancement, or third-party software) gives you an understanding of the arc of work. Scope helps you understand the business areas and systems involved.

Critical Success Factors: These help you discuss what the system must do and be. Some examples of CSFs are: Data Integrity, Correctness, and Ease of Use. These factors describe system-wide attributes that influence subsequent testing.

Necessary trade-offs: If there is a problem in developing and testing this system, what compromise will you make? Trade-offs balance Time, Resources, Scope, and Quality. If you document this early on with stakeholders, you can tune your approach to testing accordingly.

Keep the Test Strategy a brief and open-ended reference point for the Quality effort. The Test Strategy addresses broad issues that may get overlooked when you write the Test Plan. It defines important issues such as types of development, and lists of software and hardware. Develop it in conjunction with the project team. This way, you address their concerns and get early buy-in to the testing effort.

2. Assign testing duties

Typically, the test team represents the various project stakeholders. Users, developers, and management should have representatives on the test team in order to

promote and facilitate discussion of testing issues from a variety of points of view. The Test Team Lead defines the work and insures skilled resources are available to do it.

The test team completes seven major tasks. Project size, type, and scope affect the implementation of these tasks.

1. **Define objectives.** From your Testing Strategy, particularly the Critical Success Factors, determine important aspects of the system and define the testing criteria. For example, if you establish Ease of Use as a CSF, how will you test this objective? If Ease of Use refers to the user interfaces, then your objective might be user approval of the interfaces and the testing criterion might be a user review of a mock up or prototype.

2. **Determine Test Plan structure and contents.** Define what information you need to collect to meet the testing criteria. (Later, you can establish a Test Plan template.)

3. **Write the Test Plan.** Gather the requisite information (Step 2) from project members to ensure you can meet your objectives.

4. **Validate, inspect, and review the Test Plan.** Walk through the plan before implementation so the team understands what tests to do and can make suggestions.

5. **Approve the Test Plan.** Have the project team members who did the walk-through sign off. Make sure the Test Plan is understood. Ensure that the Test Plan becomes a valid part of the development process for all stakeholders.

6. **Implement the Test Plan.** Make sure you communicate what testing is being done at what time to keep the project team engaged in the quality effort.

7. **Assess the Test Plan's effectiveness.** After implementing the Test Plan and reporting results, go over the testing timeline with the contributors and evaluate how testing performed on the project. Allow people to assess what went right, what went wrong, and how to improve things for next time. This encourages people's involvement and gathers valuable information that might otherwise slip through your grasp.

If you are engaging the stakeholders of the project in this process, have them help with the tasks. Keep their assignments small; don't load stakeholders down with work, but involve them in the testing effort.

3. For existing software, set up an approval form

If you give your software to users for evaluation, give them an approval form. The approval form should tell the evaluators the level of feedback you're looking for. If you want a sign off, encourage that. If you want them to survey the functionality, have your approval form reflect that.

A good approval form can get your users thinking critically about the way the product functions now, and also encourage suggestions for how they would like the product to function in the future.

Don't delete the approval forms at the end of a project. Over time you will develop metrics mapping the course of the software's development. It will also help you predict the effort required for development of new products by providing a history of user feedback.

4. Develop requirements—even if you already released the product!

Have somebody (preferably a user) review the existing product and write down what it does. This way, you can map revisions to the software against existing

requirements, giving testers a better understanding of the integration of the revision to the system. Make sure that both the developers AND users sign off on the requirements. This establishes communication among all system stakeholders.

By developing the requirements for a project, even after it is released, you instill the importance of mapping out exactly what software does (or is supposed to do). This practice provides an additional opportunity for review in the requirements phase of subsequent projects.

5. Look into a bug tracking system

Invest in good bug tracking software. When used correctly, it will never lead you astray. It gives you a platform for formalizing bug definitions, severity, and reporting structure, and removes the difficult task of tracking from the tester's job list. Additionally, bug tracking software provides a centralized repository for all errors on multiple projects, letting the tester compare the test efforts of different projects from the same perspective.

JIRA is a good cross-platform choice (http://www.atlassian.com/software/jira/), or Bugzilla (http://bugzilla.redhat.com/bugzilla/) for UNIX or Linux only.

Testing is a full time job

Keep in mind that testing is a full time job. Dedication and proper training, along with these simple techniques, will put you on the path to good testing practices. These approaches to testing will start you down the path of communication, documentation, and metrics that will improve your Quality Strategy. ■

Steven Barry, Director of Education, Data Kinetics Leading the education department at Data Kinetics, Mr. Barry is the primary instructor and courseware developer for Data Kinetics' Software Testing Practicum (STP), a full line of instructor-lead software-testing training courses aimed at educating information-technology professionals about effective software-testing practices and techniques.

Next Issue: Q1 2005

Software Quality Call for Articles

Software Quality is looking for original articles with a friendly professional tone of voice highlighting quality assurance and best practices for ensuring quality in software. We're especially interested in project stories that demonstrate successful practices in real world situations.

If you have an article idea, please contact Teresa Wilde, Managing Editor, at twilde@osqa.org.

Your initial submission should include:

- A 100 word summary of the article. Please do not submit a completed article.
- Projected word count (Maximum 3000 words).
- Your target completion date.
- The audience your piece would address (developers, quality assurance managers, and so on).
- A 25-50 word author bio focusing on your qualifications to write the article.

Note that all articles must pass a peer review, and that we do not pay for articles.

When all the caffeine in the world can't help,

Macadamian can.

You're developing your software products under demanding circumstances—intense customer demands, shorter development cycles, tighter budgets. But pushing release dates or cutting product features are not viable options.

Our agile software lab helps you deliver new products faster and improve focus on core R&D by:

- developing new modules and features to accelerate a product release
- customizing your products for key customers or OEM partners
- building drivers and software for devices and peripherals
- assuming complete responsibility for development of a new product end-to-end

We have a solid track record of hitting the ground running, delivering on time and on budget, and quickly adapting to your core team and development processes. Now you can hit that release date without so many hits of caffeine.

Soon you'll have more time to focus on what really matters. Discover how in your FREE COPY of *The Best Case Scenario: Strategies for consistently meeting software product release dates* by visiting **www.macadamian.com/bestcase**

---

macadamian

Time to Succeed

software@macadamian.com • 877-779-6336 • www.macadamian.com