

Software Quality Throughout the Product Lifecycle

By Jeffrey Feldstein

jbf@sherps.com

<http://www.sherpas.com>

Copyright 2005-2006 Jeffrey Feldstein

Software quality is a team effort. Every member of the product team, whether they are a Product Manager (Marketing), Development Engineer, Test Engineer, Technical Writer, or Customer Support Engineer, plays a vital role in quality of the software delivered. Each person involved in the project must constantly keep quality at the forefront of their thinking. Most software development projects are made up of four major teams; product management, development, test and documentation. Holding the teams together and keeping them on track is usually a program manager. Each of these teams, including the program manager, looks at a software project from different view points but each of these view points plays a key role in assuring software quality. Other teams may be involved in developing the product such as customer support, value-add partners and at times, customers themselves. The members of these teams can also contribute to ensuring software quality.

Quality cannot be tested into the product; it must be emphasized, monitored and measured from the beginning of the project. The idea of “quality control”, where a person or a team checks at the very end of the manufacturing or development process for quality, then accepts or rejects the final product based on a series of measurements does not work well for software development projects. Bugs, performance issues and system integration issues are far too expensive to correct so late in the product lifecycle.

Traditionally, the test team’s role in ensuring product quality did not start until the product was handed off to test. Testers would install the product (if they could), try out all the features, perform a round of system testing including performance checks, try to break the product by acting like a dumb user or giving bad input, maybe supervise some external testing (beta or early field trial) and when they got tired of finding bugs, ship the product. The problem, of course, is that many bugs are found too late in the process when they are expensive to fix. Serious performance issues or integration issues are uncovered too late and can require a significant re-work of major sections of the code, causing repercussions throughout the product team, the customers, and the company’s delay in receiving revenue.

A carefully planned application development lifecycle is a key requirement to successful delivery of on-time quality software. This article focuses on the role the test engineering team plays during each step of the product lifecycle.

The product development lifecycle consists of four broad phases, requirements, development, test and post-customer ship. Each phase has important activities which directly affect the quality of the delivered software. This article will explore each phase

in detail from a software quality perspective, describing the role that a test engineer or a quality assurance department would play during each phase.

Requirements Definition

The software lifecycle typically starts with a Product Requirements Document (PRD). This document is authored by a product manager from the marketing department and it describes the product requirements. The developer reading through the document is trying to figure out how to build what the product manager wants in the product. The developer might immediately be thinking in terms of objects, data-flow, methods, procedures and data structures that might be needed to meet the requirements. He might also, immediately start worrying about resource utilization and performance issues such as CPU, memory, user network utilization, security, and user-response times. With all of these decisions, it is often difficult to step back and think about the testability, coherence and overall quality from a customer's point of view of the final product.

It is a widely-understood fact that the earlier a bug is found the cheaper it is to fix. Bugs can be detected and resolved even in the requirements phase. Fixing (or avoiding) a bug during the requirements phase ensures that it is fixed as inexpensively as possible. At first glance a test engineer might ask how he can find bugs on paper without even a prototype but they are there in the requirements if you know what to look for.

Looking for Requirements Bugs

My recommendation to test engineers reviewing a PRD is to ask themselves the following questions:

- Are the requirements clear and understandable?
- Are the requirements testable?
- Are there conflicting requirements?
- Do the requirements, taken as a whole, describe a cohesive system or do they seem fragmentary?

Understandable requirements

If the test engineer does not understand the requirement as written, it is probably unclear to others as well. Each person who reads the PRD must interpret it in exactly the same way. If a developer interprets a requirement differently than a tester, there is a high risk that either the requirement is not being met to the Product Manager's expectation or that it is not being properly verified. This can lead to hiding bugs or issues until very late in the process. This bug will be found only after the requirement is developed, the test is written, and the entire product is handed off to test, the test is run and a bug is opened. The first opportunity to start the discussion about the various interpretations of the requirement is happening extremely late in the process. Ensuring that the requirement is clear and understandable, before even a line of code is written, or for that matter, before even a functional specification or test plan is written, can avoid a potentially expensive fix much later in the product development lifecycle.

Testable requirements

One aspect of the test team's job is to verify that a product meets the requirements specified in the PRD. In order to ensure verification, the requirements need to be deterministic and "testable". A testable requirement is one that can be verified (labeled pass or fail) by a test case or a suite of test cases. An example of a bad requirement would be, "The GUI must be intuitive and easy-to use." Since this requirement is subjective and not precise, there is no way to verify whether the product is meeting this requirement. A better requirement might be: "All data entry screens should have a button labeled "View Shopping Cart", which resides in a constant location in the window. "View Shopping Cart" must lead the user, with a single click to their current shopping cart". To verify this requirement a test engineer might design a scenario that visits every GUI screen ensures the button is in the correct place and clicking on that button navigates to the correct screen.

The detailed description of the requirement also suggests that the requirement has been more thoroughly thought through and there is a higher degree of likelihood that the product will meet both the Product Manager's and customer's expectation.

It is especially important to ensure that expected performance requirements such as hardware sizing, user response times, database capacities and scalability requirements are clearly defined and testable.

Any environmental assumptions such as required operating systems, browsers, web-servers, database engines and other software need to be clearly defined in the PRD. Without this detailed information, not only will verification be difficult, but the test engineer will not be able to accurately estimate the test effort.

Conflicting Requirements

PRDs often contain individual requirements numbering in the hundreds. When a large number of requirements are present it is not hard for one requirement (or set of requirements) to be in conflict with requirements in another section of the document. It is important to have at least one test engineer read through the entire PRD with an eye for checking for conflicting requirements. Examples of conflicting requirements might be a screen mock-up that is missing the required "View Shopping Cart" button above, or have showing it in the wrong location relative to other screen-shots.

In this exercise success is not necessarily measured by avoiding all PRD conflicts, a few will probably always slip through, no matter how carefully or how many times the document is reviewed. But any conflict caught and rectified during the product definition phase, ensures smoother execution of the later phases of the product lifecycle.

Cohesive System

Another valuable input that a test engineer can give while reviewing a PRD is to provide feedback on whether the requirements, taken as a whole, define a cohesive system. A requirement document often begins with some introductory information on the philosophy of the product, and customer concerns and needs that the product being described will address. A test engineer can use their knowledge and experience of the customer, previous versions of the product as well as familiarity with the competition to inject their impression of whether the specific requirements, taken as a whole meet the goals described in the beginning. This impression, input and feedback might not be as rigorous as the previous questions, but when viewed this way, a test engineer might be able to spot a missing requirement or a requirement that is insufficiently stated.

For example if the introductory section describes “serving a company with 10,000 – 30,000 employees with up to one third of these employees access the system at a given time”, but the details later say that the “system needs to scale to 25,000 simultaneous users with no significant performance degradation”, she may have spotted a problem with the cohesiveness of the described system. The distinction here between conflicting requirements and cohesiveness might blur at times, but it is important to catch both of these error types as early as possible.

Development

The development phase typically starts when the PRD is approved and the program or project is committed. Within the development phase, some variation of the following steps usually occurs:

- Functional Specification
- Coding
- Unit test
- Integration test
- Base-level Performance test

Note that although three of the steps above contain the word “test” we have not yet reached the test phase. It is important from a quality standpoint that unit, integration and a rudimentary performance test be done during the development phase. This concept is explained further in the detailed sections below.

Functional Specification (FS)

When reviewing the functional specification the test engineer should ask themselves the following questions:

- Is the specification clear and understandable?
- Does the specification address all of the requirements listed in the PRD?
- Are there any extra functions, not required by the PRD?
- Does the specification address the related requirement fully?
- Does the specification describe a testable architecture?
- Is all the information required to write test cases present in the specification?

Understandable specification

Just as the PRD must be understandable, the Functional Specification must all be clear to all readers not just the developers. All of the consequences of an unclear PRD can happen during the FS review as well. It is important to encourage the test engineer to ask any questions they might have about unclear sections. If they do not wish to ask the question in a large review situation or in a meeting with many others present, they can either talk with another test engineer to see if they have the same question, or seek out the author of that section for a conversation around the confusing issue. Even when there is a formal review phase, with tracked comments and questions, there is still no substitute for having test engineers and development engineers working one-on-one, during the very early phases of the project. Formal process is not a substitute for a close working relationship, in a team atmosphere. Close interaction allows the engineers to begin to come together as a team, develop a relationship with each other and work towards a common goal. All of this will pay off during the frictions that inevitably arise in the testing cycle.

Addressing all requirements and extra functionality

Requirements traceability is an important part of the product development lifecycle. Traceability is the ability trace a requirement from the PRD to the FS, to the test plan (including specific test cases) and test results. There must a direct connection for every requirement so that it can be tracked through all subsequent steps. It is best to have a database to track these issues but when one does not exist, it should still be done manually, by spreadsheet or any other means available. Traceability is important because you do not want the Product Manager, or worse, the customer, finding out after the product is developed that one of the requirements has not been met. Again, it is much cheaper to find and fix the functionality holes during the specification phase of the product then after coding occurs.

Extra functionality should not be overlooked as a serious quality concern. It is often very tempting for an innovative and creative development engineer to add functionality that has not been asked for. The problem with this is that it will distract the development team from writing the required features, it will introduce more bugs into the system, and since the requirement is missing it will be difficult to understand how to test the feature because there is no requirement to test it against. The proper place to add new product features is in the PRD and the development engineer had her opportunity to pitch the idea to the product manager during the review of the PRD. Extra features added at this point are already more expensive then before because it requires another revision of the PRD, and that section, at least, will need to be reviewed by the other teams with the same questions previously raised and answered.

Does the specification address the related requirement fully?

When reviewing the specification of a feature, the test engineer may wish to go back and re-read the requirement. This will ensure that the feature as specified completely meets

the requirement it addresses. If the requirement is only partially met it or does not seem to be met at all, the section will need to be clarified. As test engineers, we are hoping that the product management team is reading the FS as carefully as we are but we should not count on it. In case an important detail is missed, this is a good time to find it and it really does not matter which department requests the clarification. In addition test engineers are constantly keeping in the back of their mind the set of test cases they will write to verify, stress, scale and break this feature, which forces a slightly different interpretation of the FS than a product manager, or another developer may have.

Does the specification describe a testable architecture?

The testability of the application can have a profound effect on its overall quality. An application that is designed for testability will allow bugs to be found earlier, allow for a greater degree of automation (increasing the accuracy and efficiency of repeated tests), and frees up the test engineer to write more sophisticated testing instead of spending his time figuring out a way to automate a simple scenario. The FS review is of course, far too late in the development process to begin discussing product testability and some testability issues will not be determinable until a more detailed design is produced, but it is a good place for a check-point that previous discussions were incorporated.

A full discussion of testable architectures is not appropriate here, but in general, for the most part, the characteristics of a testable system also make sense from a software engineering point of view and will have benefits throughout the product. One example of this is building a user interface (UI) (graphical, command driven, voice activated, etc.), that is separate from the actual business logic. The interface sections should simply have the job of taking input from and displaying output to the user. Any other work (input verification, business logic, error checking etc.) should be done by the back-end or engine. The UI and back-end should communicate through a well-defined and fully specified Application Programming Interface (API). This architecture makes sense even if testability was not an issue, because it allows UI changes (or managing multiple UIs) to occur in isolation from any engine work. Test can also make use of this approach by using the API to fully test the back-end of the product, independently test the user-interface portion, and put them together for testing only after each piece is working independently of the other. Since user interface design is often fluid, controversial and subjective, having a solid separation between front and back-ends makes sense for both the development and test teams. Other examples of testability often have the same effect.

Is all the information required to write test cases present in the specification?

Detailed test plans are typically written based on the PRD and the FS. Within these two documents must be all of the information required to write test cases. Writing the detailed test plan, and often, implementing automation need to take place at the same time as coding the system. If a test department is planning on API testing, the FS, must contain the complete function or method signature of all published methods. Expected outputs

and handling of errors should be detailed in the FS so that test plans can be generated from it.

Coding

Ideally coding of the application and writing of test cases (both manual and automated), happen simultaneously. For scheduling reasons, these activities ideally should not occur sequentially. The more time spent up front on designing the application this smoother the execution of the coding phase.

Unit Test

When a developer checks-in his module or unit, the unit test software should be checked in at the same time. This ensures that a level of bugs has already been found and some coding has taken place to check for future regressions on that unit. By writing the unit test code in a system such as JUnit (for Java applications), and delivering the test code with application the development team is freed up from the laborious task of documenting and demonstrating their unit test results. Automating the unit tests allows for an automatic unit-regression facility that can be run with all subsequent product builds.

Integration Test

Integration Test is built on the same technology and automated for the same reasons as the unit test. The difference here is one of emphasis on what is being tested. For integration testing, developers write code to test the integration between two units. In other words, if unit "A" calls unit "B" for certain kinds of transactions, an integration test of unit A will execute those transactions. In a unit test, unit B's code, will usually be stubbed out in order to perform a test on unit A in isolation. In the integration test, both units will be executed.

Base-level Performance Test

It is also advantageous to the entire project if the development team runs a set of performance tests before handing off the code to test. This will check the reasonableness of the performance of the functions and allow development to have its first early examination of overall performance. This can then be matched against the requirements from the PRD to see if the project is on track. This is also a good time to determine whether issues can be resolved with minor changes or if major re-work is required.

Test Strategy

The test department will typically write two major documents. The first is the overall Test Strategy. This is authored during the writing of the functional specification and is produced slightly after the functional specification. It will outline goals for automation, how automation will occur, tools to be used, the test lab equipment and topology, how the

system will be tested for security. This document describes how the test will occur and the major test types including: functional, system, performance, scale, stress, and soak. It will describe, in a general sense, how these tests will occur. The other goal of this document is to have enough information in it so that the test effort can be estimated. Test effort includes number of testers, time to implement and run the tests, money and time to set up the test laboratory, the strategy for external testing (Beta and/or Early Field Trials). This document also gives the other teams (product management, development, customer support) a view into how testing will be performed and allows them to be sure that all of the major areas are covered. Review of the test strategy provides a forum for feedback and suggestions on test approaches and techniques.

Detailed Test Plan

The detailed test plan describes the specific script or scenario for each case. The word “script” refers to detailed documentation that describes step-by-step instructions for validating one or more features of the software. We are not referring here to automation which is sometimes referred to as scripts. The test plan is the test department’s equivalent of the Functional Specification. It needs to specify the testing that will occur as completely as possible. All pre-conditions, the script details, and expected results should be present in the test plan. The test script should be written regardless of whether the test case is going to be manual or automated, but the plan should indicate whether the test case will be automated for this release.

The test plan and should include “negative” (meaning handling errors, unexpected events and bad input properly, without adversely effecting the system) tests. The importance of negative tests should not be overlooked. Many bugs are discovered by a customer not understanding a feature, and selecting wrong options, or an unexpected input from the environment. Low memory or disk, user response time-outs, and network interruptions are a few examples of environment problems that the software should protect itself against.

Just as documents written by marketing and development are approved by test, both of these documents should be reviewed and approved by the entire product team. This is marketing’s opportunity to obtain a level of confidence that the product will meet its requirements. It is a good idea for developers to pay careful attention to the test plan, because they can give good feedback on whether the test cases are valid (Do the scripts make logical sense?), while checking for missing test cases. In addition, simply by reading and understanding the test plan, the developer may code to be sure the test case passes and therefore avoid a bug report entirely.

Test

Now that all of the documents are written and approved, the timelines are understood, and deliverables are detailed, it is time for the execution phase. During this time, developers are coding and testers are coding the automation. This will continue until development is ready to hand-off either a distinct component or possibly the entire

product to test. This milestone, Handoff To Test (HOT), is an important date from a project management and scheduling point of view. It says that the software is complete, all features are implemented and it is ready for formal testing. Any bugs found in or after HOT, should be formally tracked in the bug tracking system.

Product teams (including test) tend to want to jump to the execution phase as quickly as possible. Management, investors and customers, often pressure teams for demonstrations and visible progress. The problem is that the faster the software moves to the execution phase, the higher the risk of low quality, serious performance issues, integration mismatches, or schedule delays.

Bugs found during the execution phase of the project are inevitable, and although they are more expensive to fix (a bug is filed, code must be changed, regression testing re-executed, and the fix verified), it is still cheaper to fix bugs now than after the product ships. Bugs that are shipped incur the additional expenses of interaction with customer support personnel, interrupting development and test activities on future releases to reproduce the issue in the lab, fix the bug, verify it, re-build the system, and re-release and re-distribute the fix. In addition, customer satisfaction can be adversely effected with any bug they might find.

Ship

Now that we have been running tests, finding and fixing bugs, four things occur simultaneously:

- Product quality improves
- Upper management pressures to release for business or revenue reasons
- Product managers pressure for release (they are anxious to please their customers)
- Fatigue sets in for the development and test teams

The best way to avoid shipping because of the latter three reasons is to ensure that the measurable, objective product quality goals are agreed to at the beginning of the project. Some suggestions for shipping goals are:

- All system crash and “must fix” bugs resolved and verified
- All fixed bugs verified
- All customer found bugs, from previous releases, are fixed and verified
- All performance goals met and results documented
- Total number of open bugs not to exceed XX
- Defect density not to exceed X bugs per thousand lines of code (KLOC)
- Code and branch coverage of tests to a specific target

The specific numbers above might vary for each project. A new product or company where immediate revenue is important and the customers (possibly early adopters), may be able to tolerate a higher bug rate, may set themselves different goals for the release than an established product where customers have a large investment in the current product, and upgrades are expensive. The emphasis, however, should be towards the team

committing to objective goals, continual improvement, and ensuring that the product's quality increases with each release

Post Ship

Many people might ask why continue testing after a product is released? This is however, a good opportunity to find additional bugs. The automation can be run continuously, perhaps using some Model-Based testing if it is available, for an extended time. Typical bugs found after shipping might be very slow memory leaks, new security vulnerabilities and performance issues related to running the software continuously over an extended period of time. With good automation the cost of this is relatively small; just set up a system, check it periodically and learn from what happens. Lessons learned during this phase can be brought into pre-release testing of future versions.

Escapes

No matter how well the software is tested, and how carefully the principles above are followed, some bugs will escape the testing process. These bugs will be found and reported by the customer. It is important to have a process in place to constantly monitor the incoming customer found bugs, analyze them, figure out why the escape occurred and ensure that this bug is caught in testing subsequent releases, by adding test cases to future test plans.

Conclusion

I will end where I began: Software Quality is a team effort. Having just one team assuring software quality does not work as well as everybody on the team dedicated to it. Each team in the development process should have an equal say in all major product decisions. The best way to achieve high quality, on time, software is to have ownership of all goals and schedules from each member of the team. Each department's work is reviewed by all of the rest to get inputs from the entire team. Buy-off and agreement is emphasized over dictating either from upper-management or from the various department members. By following the guidelines above the hope is that we can improve quality, meet or exceed customer expectations and significantly contribute to increased customer satisfaction, leading to improved business (or department) success.