System Testing

Though defect prevention techniques can be quite effective, and are to be encouraged, defects can and will be introduced in a variety of different ways throughout design and construction, regardless of the process used, whether traditional or agile. Hence software testing, which makes use of the product while it is in operation or being operated on, is an essential part of all defect elimination efforts. As discussed in the chapter on Software Quality Assurance, software testing attempts to create trigger conditions (inputs) that will give rise to failures indicating the existence of faults that can be isolated and eliminated. What differentiates system testing from unit testing and integration testing is that the product is treated as if it were atomic, that is, as if it can't be decomposed into constituent parts.

We begin with an overview of testing and a comparison of system testing and sub-system testing (unit testing and integration testing are discussed in the chapter on quality assurance in construction). We then discuss different aspects of system testing in greater detail. Next, we consider system testing in context. That is, we consider how system testing is used in traditional processes and agile processes. We then consider different system testing tools and how design and construction practices can facilitate system testing. We conclude with a discussion of various organizational issues related to system testing.

Overview

In the chapter on Software Quality Assurance, testing was divided into four stages—unit testing, integration testing, alpha testing, and beta testing, with the last two collectively referred to as system testing. To properly discuss system testing, we need a somewhat less abstract view.

As discussed before, testing often begins during the design process (with the creation of unit tests) and certainly begins during construction with the creation and execution of unit tests and integration tests. Unit testing happens in parallel with construction. That is, each developer tests the units that he or she is implementing.

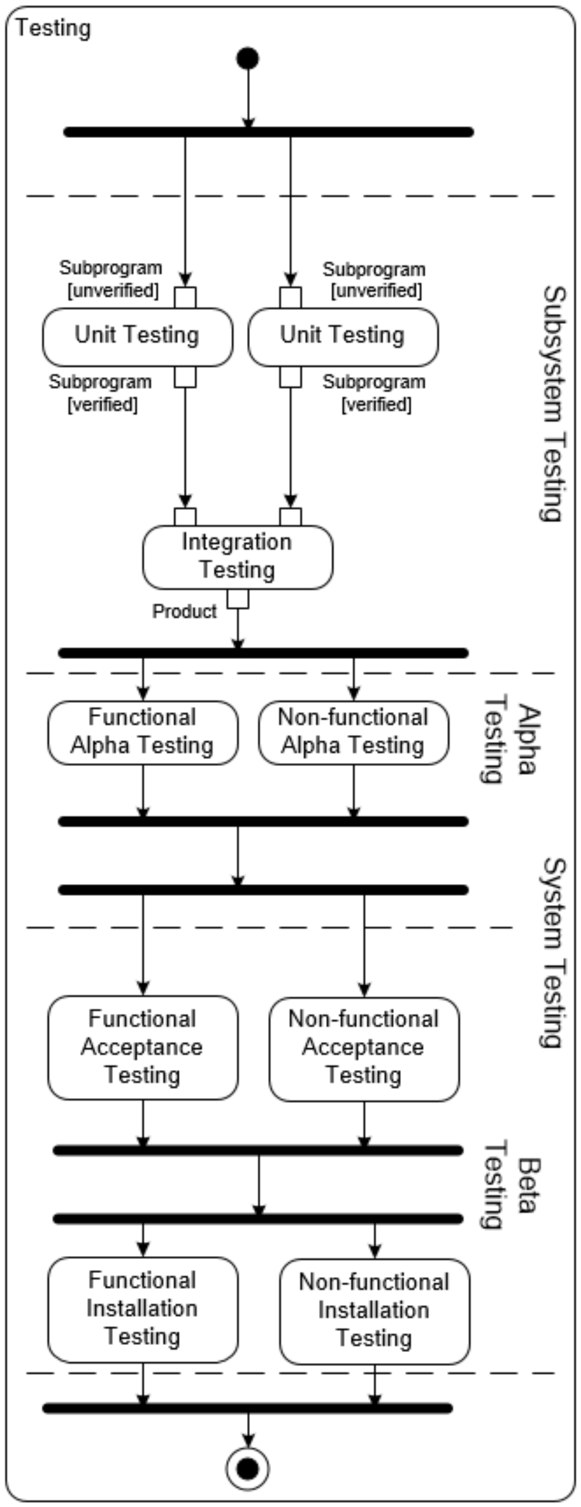
After a sufficient number of units have been constructed and tested, they can be combined into larger sub-systems (sometimes called *assemblages*) and tested together. Conceptually, integration testing begins after unit testing is completed but, in practice, there is considerable iteration and it is often difficult to distinguish one from the other.

After integration testing has been completed, (an increment of) the product can be built from the supposedly fully-tested sub-systems. Despite the fact that the sub-systems are fully tested, there is no reason to expect that the product will not fail. There are several reasons for this. First, the sub-systems could still contain faults. Second, there could be faults at the interfaces between the different sub-systems. Third, because the operation of the sub-systems often depends on the state of the system, it may not have been possible to truly fully test the sub-systems. Hence, system testing is needed.

The first stage of system testing is normally referred to as **alpha testing**, in which developers test an entire product. During this stage, developers simulate the behavior of actual users. Alpha testing involves both functional and non-functional tests, which often can be executed in parallel.

The second stage of system testing is normally referred to as **beta testing**, which involves actual users testing the product in either a simulated environment or the actual environment. Like alpha testing, beta testing involves both functional and non-functional tests, which can be executed in parallel.

This process is summarized in the following activity diagram.



One of the things that can be somewhat misleading about this diagram is the end node. While the testing activity does, indeed, end at some point, the *acceptance criteria* used to determine when testing is complete (and the negotiations that often take place) can be difficult to define. Particularly important in this regard is the fact that one fault often hides or obscures other faults, which means that testing must be conducted iteratively. This aspect of test planning will be ignored for now but considered later. At this point, we focus attention on the specification, description and categorization of tests.

Differences between System and Sub-System Testing

Sub-system testing (unit testing and integration testing) and system testing differ in two important and related ways that are partially evident from this activity diagram. First, as mentioned above, system testing involves (an increment of) the entire product whereas sub-system testing does not. Second, system testing typically involves units or sub-systems that were constructed by several people. Unit tests, on the other hand, are typically created and executed by the developer who constructed the unit.

Combined, these two differences have important implications for the kinds of faults that give rise to the failures detected during system testing. Inputs that can only be handled by the entire system rather than some part of it can lead to failures of non-functional requirements that apply to the entire system, like low reliability, inadequate throughput, or insufficient precision of results. System tests can also reveal problems arising from different people understanding requirements in different way, or making differ assumptions about the system. For example, one part of the sytem may assume that angles are in degrees while another parts assumes they are in radians. Under this assumption, each sub-system could pass very thorough suites of tests, but completely fail system tests.

Sub-system testing and system testing differ in several other important ways that are not evident from this activity diagram. They differ in the kinds of people that are involved (not just the number), the aspects of the design process that are used to create tests, the dependence of tests on the nature of the product and the design and implementation paradigm used, and the relationships between the different individuals that are involved.

- Sub-system testing is performed by people who have intimate knowledge of the engineering design and implementation of the product, and, hence, can include both clear or white-box and black-box techniques. System testing is performed by people who have an intimate knowledge only of the product design (the requirements) and, hence, only involves black-box techniques.
- Sub-system tests are usually created from detailed requirements specifications or detailed user stories (either sprintable stories or user stories about tasks). System tests, on the other hand, are usually created from descriptions of interactions between users and the system (such as use cases or user stories about features). This has implications for the nature of the tests, the kinds of failures that they are likely to trigger, and the state-dependent nature of the tests (that is, the state of the system is likely to change during a test).
- Unit testing and integration testing depend, to some extent, on the nature of the product (for example, a WWW service vs. a desktop application), whether the product is bought and customized or built, and the approach used to either customize or build the product (for example, object-oriented vs. functional). By the time system testing begins, however, these differences no longer matter (or, at least, matter much less). Regardless of the way in which the product arose, it is going to be tested in essentially the same way.
- Unit testing is more of an individual activity than system testing, which necessarily involves the entire team or a separate testing team. Hence, the problem of finger pointing is less likely to arise in unit testing and more likely to arise in system testing. For example, in system testing opportunities arise for the software people to blame the hardware people, the GUI people to

blame the database people, etc. Obviously, this kind of finger pointing can be very damaging, and steps must be taken to avoid it.

Details of System Testing

System testing can be considered at many different levels of abstraction. In the chapter on quality assurance it was divided into two components, alpha testing and bet testing. In the activity diagram above, it is divided into six different components. This section looks inside each of these six components to provide additional detail. Despite this, the discussion that follows is still an abstraction and considerable detail is omitted.

The differences and similarities between alpha testing, beta acceptance testing, and beta installation testing are summarized in the following table and discussed more fully below.

	Alpha Testing	Beta Testing	
		Acceptance Testing	Installation Testing
Personnel	Testers	Users	Users
Environment	Controlled	Controlled	Uncontrolled
Purpose	Validation (Indirect) & Verification	Validation (Direct)	Verification
Recording	Extensive Logging	Limited Logging	Limited Logging

Alpha Testing

Alpha tests are used both to validate that a product satisfies the needs and desires of users and to verify that it does so correctly. However, validation is often indirect in that alpha testing does not involve real users. Instead, it typically involves a group of testers attempting to behave like real users. In other words, alpha testing involves internal testers who are part of the organization developing the software product, either structurally or contractually. Hence, they rely on product specifications (for example, the software requirements specification) to validate that the needs and desires of the real users are satisfied.

In many organizations, an attempt is made to create an independent test team that organizes, creates and runs alpha tests. The rationale is that the designers, developers, and authors may make inappropriate assumptions (for example, that what they have implemented is what was required). They may also, knowingly or unknowingly, have conflicts of interest. That is, it may not be in their best interest to identify failures; indeed they have a vested interest in demonstrating that the product is fault free.[1] The test team may include some product designers or requirements analysts so that their expertise about user needs can be incorporated. Finally the test team might include designers or developers involved with the engineering and construction of the product to take advantage of their expertise. In some cases, developers are also needed to create some tests, depending on the type of testing tools used. Obviously, in such situations it is important to take steps to ensure that the testing team and the development team do not become adversaries.

Alpha testing is conducted in a controlled environment, though one that attempts to simulate the real environment in which the product will be used. This helps testers isolate and record failures. Though the testers are responsible for recording failures, the product itself will often keep an

1 Obviously, testing can never demonstrate that a product is "fault free", it can only identify failures. A product that does not fail, no matter how many tests it is subjected to, may still contain faults.

extensive log of activity during alpha testing. Though the instrumentation required for logging may interfere with the operation of the product, at this stage in the process the benefits of logging usually far outweigh the costs.

Functional alpha testing attempts to validate and verify the features provided by the software product. Validation tends to occur when the tests are created and verification when the tests are conducted. For example, consider a simple product that performs various calculations involving weights (such as addition, subtraction, multiplication and division). When constructing the tests, the testers would use the product design documents (requirements) to validate that the product performs all of the required operations and that they will all be tested. Then, the tests would be conducted to verify that these feature operations are performed correctly. As with all testing, the tester must know the expected correct behavior. Hence, functional alpha testing can be very resource intensive.

Functional alpha testing is often conducted in a bottom-up fashion with the most basic functionality tested first, as it makes it easier to isolate failures. For example, again consider a simple weight calculator. The most basic features of such a product are the ability to enter weights and display weights. So, before testing whether the product can add two weights one should test whether it is possible to enter a weight and whether the product can display a weight. Otherwise, the tester might misconstrue or miscategorize a failure. Of course, the tester's job is to find the trigger condition for the failure, not the fault that resulted in the failure, but the tester should avoid adding confusion if at all possible.

Because of the number of paths through a typical product, some organizations use **operational profiles** that contain information about the relative frequency of different use cases and the order in which use cases typically occur. Test cases are then created that reflect these patterns of usage.

Non-Functional Alpha Testing—Recall that there are two broad categories of non-functional requirements: development requirements and execution requirements[2]. Since development requirements are related to the needs and desires of the development team (such as portability and maintainability) they are not, in general, testable. Instead, they must be validated using a review process. Execution requirements, on the other hand, are related to properties of the use of the product (such as response time and memory usage) and are, in general, testable. Some kinds of non-functional execution tests are so common that they have been named, including the following.

> **Timing tests** are used to evaluate non-functional requirements related to the time required to perform a function. Timing tests often make use of **benchmarks**, which are standardized timing tests. Some timing tests involve user interactions and some do not.
>
> *Reliability and availability tests* are used to assess the performance of the system under normal operating conditions. Formally, **reliability tests** aim to determine the probability that the product will fail in a given interval of time while **availability tests** aim to determine the probability that a product will be usable at a particular time. Conceptually, both are concerned with the dependability of the system. For some kinds of products, these kinds of tests involve the calculation of statistics like the *percent up time* and the *mean time to failure*.
>
> **Stress tests** (sometimes called **torture tests**) are used to determine the **robustness** (the ability to operate under a wide range of conditions) and the **safety** (the ability to minimize the damage resulting from a failure) of the system. Stress tests that are focused exclusively on safety are sometimes referred to as **recovery tests**.

2 Execution requirements are commonly called operational requirements. As discussed earlier, the term "execution requirements" is used here so as not to cause any confusion with operation-level requirements.

Configuration tests are used to assess the product on different hardware or software platforms and in different operating environments.

It is important to note that some execution requirements, while testable in principle, are very difficult to test in practice. For example, security requirements are sometimes difficult to test. While it is possible to simulate users for purposes of stress testing, it is almost impossible to simulate attackers or adversaries realistically for purposes of security testing.

It is also important to note that some kinds of tests are difficult to categorize as either functional or non-functional. Perhaps the most important example of this is tests that specifically target the user interface. For example, consider the following.

When the user slides the units toggle from degrees to radians the units indicator on the display must change from degrees to radians.

Is this a functional requirement about how a product maps inputs (like button clicks) to outputs (what appears in the display) or is it a non-functional requirement that a product must have certain properties? Most people tend to categorize this kind of requirement as functional, but consider a requirement like the following.

The GUI must be available in both Italian and Arabic.

Most people would consider this a non-functional requirement.

Tests checking requirements about the user interface are called **usability tests** (or, sometimes, **human factors tests**).

Internationalization/localization tests are a particularly important kind of usability test. They involve both translations and locale-specific information like currencies, number formats, and time and date formats. While some of these things can be verified when the product isn't in operation (for example, some translations), many have to be tested. For example, the appropriate word or phrase to use might depend on the context in which it is being used or the part of speech or gender. For example, "New" might be translated as either "Nouveau" or "Nouvelle" depending on the gender of the new thing. In addition, tests are needed to ensure that all strings are translated (and not mistakenly hard coded) and to ensure that character encodings are available and used properly.

Increasingly, an important part of usability testing is **accessibility testing**, which involves ensuring that the user interface is appropriate for all people, regardless of physical abilities. For example, Section 508 of the Rehabilitation Act of 1973, which applies to all Federal agencies, requires that people with visual acuity between 20/70 and 20/200 be able to operate software without relying on audio input. Accessibility testing, and other kinds of usability testing, can often involve hardware or software that measure the amount of time required to complete a task (for example, eye-tracking systems).

Beta Testing

Beta testing, sometimes called *pilot testing*, differs from alpha testing in two important respects. First, whereas alpha testing involves internal testers, beta testing involves external testers. Second, there is increased burden on the user (or trained observers) to record failures. Logging can be used in beta testing, but only to the extent that the code used to perform the logging will remain in the final product (since it could, itself, introduce faults or change the behavior of the system, and therefore is should not be active during this phase of testing).

Acceptance testing is done by clients to validate the product (that is, to determine that it satisfies the clients' needs and desires). Like alpha tests, acceptance tests are conducted in a controlled environment (provided by the developer). Sometimes, though not always, it is the same environment that was used for alpha testing. Unlike alpha tests, acceptance tests are conducted by real users, hence are a direct form of validation. In some cases, trained observers monitor the tests.

As with alpha testing, acceptance testing involves both functional tests and non-functional execution tests (but not non-functional development tests). Unlike alpha testing, acceptance testing does not typically involve operational profiles. Instead, users interact with the system as they would in a real environment. In other words, the order and frequency in which features are used is not controlled. However, like alpha testing, all features are tested (even if they are used infrequently in practice) so that the product can be validated.

Installation testing involves real users in real uncontrolled environments. Since validation problems should have been found during acceptance testing, installation testing is used principally to verify the product, in particular, to determine that it operates properly in the user's environment. Of course, in practice, validation failures do arise during installation testing and must be noted. Ideally, the users have received some training or instruction about how to report failures, but they are in not expert testers. Hence, the failures they report (and the trigger conditions that resulted in those failures) can be difficult to understand and hence to replicate. For this reason, testing programs should attempt to minimize their dependence on installation testing. Though it is an essential part of the process, it is far and away the least efficient kind of testing (despite the fact that installation testers are often volunteers for mass-market products).

When a new product is replacing an existing product, it is sometimes possible to use the two products in parallel. Though this reduces user productivity and, as a result, can be quite expensive, it is invaluable, especially for mission-critical software products. Using both systems during a changeover period also provides a margin of safety in case the new system fails disastrously—the old system can simply continue in use while the new system is fixed.

Obviously, the details of installation testing vary dramatically with the architecture of the product. For example, standalone products and distributed products will be tested in very different ways. When testing distributed applications, for example, one might test client functionality and server functionality independently, and then conduct transaction tests that involve both. As another example, when testing web apps one might distinguish between content tests, function tests, usability tests, navigability tests, interoperability tests, and security tests. Regardless of the architecture, however, the objective is the same—to verify that the product will work properly when it is released.

System Testing in Context

As with unit testing and integration testing, the timing of system testing varies with the software process being used as does the amount of planning that is devoted to system testing.

Testing in a Traditional Process

In traditional processes, specifications should always be verifiable and hence individually testable. Thus, testing in traditional processes starts with the software requirements specification. That is, in traditional processes, each requirement is mapped to one or more tests. Some requirements will be mapped to unit tests and others will be mapped to system tests. In practice, few requirements tend to be mapped to integration tests.

Unit testing is done during the implementation or construction phase. Hence, it tends to occur early and often in traditional processes. Coders are intimately involved with unit testing in a traditional process but, because of the length of cycles, testers are also often involved in the creation and execution of unit tests. Since integration testing and system testing occur much later, rigorous unit testing (and the debugging that results) is critical in a traditional process.

Integration testing often doesn't begin until fairly late in traditional processes. In other words, it is common for many units to be constructed before they are combined and tested. As a result, in traditional processes it is often necessary to create **stubs**—placeholder code that stands in for units that have not yet been constructed. A stub has the same interface as the code it is standing in for, but provides none (or very little) of its required functionality. Not surprisingly, when the stub is replaced with the fully-functional unit during integration testing, faults are often uncovered. As a result, integration testing is particularly important in traditional processes.

Alpha testing often can't begin until the end of a traditional process. This is because traditional processes make no effort to have a working (if limited) version of the product at all times. Since one can't test the system until it exists, and the system doesn't exist until the end of a traditional process, system testing can't occur until very late. Furthermore, traditional processes tend to have long cycle times, so there is typically a long period between system specification and system test. This is, perhaps, the biggest reason that traditional processes are not considered agile.

Because of the length of the cycles in traditional processes, the product that is released at the end of each cycle is often thought of as the "final product" (even though the product has a version number), and changes are described in relation to it. Of course, though one would like to think that all such changes are improvements, any change to a software product might introduce new faults. Regression testing is crucial in determining whether changes to a product have introduced faults.

In a traditional process, a considerable amount of effort is devoted to the planning of system testing, the result of which is a document called a **test plan**. A test plan typically includes

- The business and technical objectives of the test suite;
- The test cases;
- The review process and acceptance criteria;
- An estimate of the size of the testing effort;
- A schedule for the testing effort.

Testing in Scrum (and Other Agile Processes)

In a traditional process, tests are created from the requirements specification. Since Scrum does not make use of a formal requirements specification, test development in Scrum proceeds differently from test development in a traditional process.

Perhaps most importantly, in Scrum integration testing and alpha testing are conducted each and every sprint. Indeed, most agile proponents argue that frequent integration and alpha testing is a big part of what it means to be agile. In Scrum, integration and alpha testing are used to demonstrate that features in the product backlog have been completed.

Unit testing in Scrum is left entirely to the developers. Those who argue for this approach believe that unit testing will find implementation failures and integration and system testing will find design failures. Alternatively, one can specify acceptance criteria (also known as *conditions of satisfaction*) as unit tests. Using this approach, it becomes very clear when a user-level product backlog item (PBI) has been completed.

As part of alpha testing[3], many agile processes advocate the use of daily **build verification tests** in which the product is built and tested (using automated tools) each day. Such tests are sometimes called **smoke tests**, and ensure that the product remains at a given quality throughout its incremental development. Of course, smoke testing requires discipline on the part of the team—each developer must smoke test her or his code before committing it. To encourage the necessary discipline, many organizations penalize developers that break a build (sometimes in comical ways but sometimes in more serious ways).

The timing of beta testing in Scrum is somewhat more ambiguous. Some practitioners beta test regularly (though, perhaps, not every sprint) and use beta testing as part of the product design process. That is, each increment is used to elicit user needs and refine user stories. Other practitioners only beta test **release candidates** (RCs), which are product versions containing all features planned for a release.

In Scrum, it is much more difficult to isolate regression testing from the rest of the testing process. This is because, without formal a requirements specification, it is difficult to distinguish old features from new features and, hence, difficult to have a baseline set of tests.

Though testing is an important part of Scrum and other agile methods, test planning tends to be informal. Hence, though organizations will have a generic testing process in place, there may not be test plans for individual products.

System Testing Tools

A variety of tools can facilitate system testing. Some assist with developing and executing tests and some assist with recording and reporting results.

Test Development and Execution Tools

Software products that do not have a graphical user interface (GUI) can, in large part, be tested in an automated fashion in much the same way as individual modules can. Though a custom test harness may be required for system testing because existing unit testing frameworks may not be appropriate, the concepts and process are the same as in unit testing.

Automated testing of software products with a graphical user interface (GUI) is rather different, however. Fortunately, many tools exist. Some tools allow the testing team to record an interaction with the product, indicate the anticipated state of the system after the interaction, and compare the two. These recordings, which include such events as mouse movements and key presses, can then be played back to conduct the test again for debugging or regression testing. Though the initial creation of the tests is very labor intensive because a tester has to interact with the product, repeated execution of the tests is not.

Other tools allow the testing team to write scripts that describe the interaction, rather than record an actual interaction. Creation of these scripts is labor intensive and requires specialized expertise, but the scripts are often much easier to understand than recordings (since they are a verbal rather than visual representation of the interaction). Rerunning the scripts is as easy as rerunning the recordings.

Hybrid automation tools support both. That is, user interactions can be recorded, but the recordings generate scripts that can then be edited in various ways. For example, one might record an

3 Some authors consider smoke testing a part of integration testing while others consider it a part of system testing. We consider it a part of system testing because, given the amount of effort required to keep the smoke tests up-to-date and conduct the daily builds, it is often the job of a dedicated person or team. In practice, smoke tests often start as integration tests but, as the product and tests become more complicated and sophisticated, become system tests.

interaction that involves saving a document, copy the resulting script, and then modify it to create a test for opening a document.

Test Recording and Reporting Tools

Tools also exist that help testers record, track, and report test results. Commonly called **bug tracking systems**, they essentially are specialized database systems that manage information about defects, most commonly failures and the trigger conditions that led to those failures.

Bug tracking systems are often used both during development (in alpha and beta testing) and after a product is deployed. Though these are conceptually very different activities, in practice they have enough in common that one tool can often be used for both.[4]

Preparing for System Testing

Many things that can be done during early stages of the software process that facilitate system testing. Most importantly, software products can be both designed and constructed to make them easier to test.

Engineering Design for System Testing—Some design patterns make it much easier to conduct system tests. Perhaps the most important example of this is the Command Pattern discussed in the chapter on engineering design. Recall that the Command Pattern encapsulates functionality in individual objects. This makes it easy to programmatically initiate functions that would normally be triggered by a GUI event. For example, there might be an `Open` class that encapsulates the functionality required to open a document in a word processor. A test case can be written that constructs an `Open` object and call its `execute()` method without having to actually click on the button that would normally initiate this action. While the GUI still must be tested, the Command Pattern makes it easier to test the functionality independently of the events that trigger that functionality.

Other patterns, for example the Observer Pattern and the Proxy Pattern, make it easy to unobtrusively instrument code for system testing. Though the details of these patterns are outside of the scope of this discussion, essentially one can use them to create monitors or loggers that can interact with other components in exactly the same way as components that are providing required functionality. Hence, though they can change the timing of and order in which code is executed, they have minimal impact on the product and are unlikely to introduce or mask defects.

Construction for System Testing—It is also possible to make decisions during the construction phase that facilitate system testing. For example, some languages and libraries have built-in support for logging (for example, the Java Logging Framework includes `Logger`, `Level` and other classes that greatly facilitate logging). As another example, the use of exceptions (that is, using `try-catch` blocks rather than special return values) generally makes it much easier to build a flexible test harness.

References

1. Gilb, T. (1995) "What We Fail to Do in Our Current testing Culture", *Testing Techniques Newsletter*, Software Research, January.
2. Musa, J. (1993) "Operational Profiles in Software Reliability Engineering", IEEE Software, March, pp. 14-32.

4 Post deployment systems are sometimes referred to as *issue tracking systems* and often include a knowledge base of work arounds that can be used by customer support specialists to provide assistance to users. Individual entries in an issue tracking system are often referred to as *tickets*.

3. Jones, Capers. *Software Quality In 2013: A Survey Of The State Of The Art*. Http://Namcookanalytics.Com/Software-Quality-Survey-State-Art/.