Investing in Software Testing:
The Importance of the Right Technique

Abstract

The choice of the right techniques is critical to achieving a good return on the test investment. Some tests happen before we can even run the software. Some tests involve analyzing the structure of the system, while others involve analyzing the system's behavior. Each technique can involve special skills and particular participants, and might appropriately entail the use of tools—or not.

Introduction

Not only does testing the right things affect the return on the testing investment, but testing them in the right way makes a big difference, too. Various techniques exist for test design and execution. Some tests don't involve running the system at all, others do. Some tests require inside knowledge of how the system works, while others require an understanding of what the system does. Some tests are run step by step by hand, while in other cases a computer does the test execution, though skilled test engineers must perform the test design and result interpretation. The skilled test professional achieves the optimum return on the testing investment by selecting test techniques wisely.

Static Testing

A static test is one that evaluates the quality of the system without the system actually running. This may seem very strange, even to some experienced testers. How can we run a test without running the system?

A static test looks at the system—or portions of or artifacts related to the system—to see if we can find problems early. When programmers read their code after writing it, this is called desk-checking, which is one kind of static test. Another kind of static test, in this case a social one, is a review meeting that evaluates requirements, design, or code. The benefit is clear once you think about it: If you can find a problem in the requirements before it turns into a problem in the system, that will save time and money. (In *Software Requirements*, Karl Wiegers reports a return on investment for requirements reviews of as high as 800%.) Even if we find the problem by reviewing the code, that saves us the overhead of building, installing, and running the system to find the bug.

Not all static testing involves people sitting at a table looking at a document. Sometimes automated tools can help. For C programmers, the *lint* program can help find potential bugs in programs. Java programmers can use tools like the JTest product to check their programs against a coding standard, which can be customized.

To get value from static testing, we have to start at the right time. For example, reviewing the requirements after the programmers have finished coding the entire system may help testers design test cases. However, the significant return on the static testing investment is no longer available, as testers can't prevent bugs in code that's already written. For optimal returns, a static testing should happen as soon as possible after the item to be tested has been created, while the assumptions and inspirations remain fresh in the creator's mind and none of the errors in the item have caused negative consequences in downstream processes.

Effective reviews involve the right people. Business domain experts must attend requirements reviews, system architects must attend design reviews, and expert programmers must attend code reviews. As testers, we can also be valuable participants, because we're good at spotting inconsistencies, vagueness, missing details, and the like. However, testers who attend review meetings do need to bring sufficient knowledge of the business domain, system architecture, and programming to each review. And everyone who attends a review, walkthrough, inspection, or whatever it's called in your organization should understand the basic ground rules of such events. A good reference on this topic is Daniel Freedman and Gerald Weinberg's *Handbook of Walkthroughs, Inspections, and Technical Reviews* .

Structural ("White Box") Testing

Static testing is a powerful—but not a sufficient—test technique. At some point we must actually run the system and look for bugs in it. Tests that involve running the system under test are called dynamic tests. Structural tests are one major example of dynamic tests.

Structural tests are based on how the system is built. Some people refer to structural tests as "white box" or—perhaps more appropriately—"glass box", because the tester looks inside the "box" (the system under test) and derives tests from this knowledge. Structural tests are most typically applied to individual components and interfaces, being particularly effective at discovering localized errors in control and data flows.

One elegant and powerful form of structural testing involves reusable, automated test harnesses for the system under test. Using this harness, programmers can create structural test cases for components as they code them. Programmers can then check these tests into the source code repository along with the component. A cleverly-constructed test harness will execute this ever-expanding, automated regression test suite each time the source code is built into a new release. A colleague of mine, Greg Kubaczkowski, and I described a specific example of this kind of structural test harness in the article "Mission Made Possible," available on www.stickyminds.com.

Another form of structural testing involves the creation of custom test data. Testers creating such data often must understand the underlying database schemas or designs, especially when working on carefully crafted data sets that

probe the boundaries of data validity in one or more ways. In addition to testing individual components and data flows, with careful design we can reuse such test data for other kinds of tests, a topic we'll revisit later in this series of articles.

Because structural testing requires an understanding of the internals of the system under test, programmers and other development engineers can do an excellent job of structural testing. In enlightened organizations, you may find the programmers charged with structural testing, but ably assisted by a test team that includes programming experts and test toolsmiths, helping to set up test harnesses. Testers with experience in automation also tend to know tricks for developing reusable and sharable test cases and data. This can be especially useful when it comes time to test the interactions between components.

No matter who does the structural testing, they will need to understand some fundamental test design techniques to do a good job. For example, one of my client's programmers were talking about structural tests that "tested everything that could break" when we started working with them. In addition to building the test harness described in "Mission Made Possible," my associates and I taught these programmers effective techniques for designing structural tests that find bugs, such as boundary analysis, condition coverage, basis tests, and the like. Books such as Boris Beizer's *Software Test Techniques* and Bill Hetzel's *The Complete Guide to Software Testing* discuss these and other structural testing techniques in more detail. Those charged with structural testing should understand and apply these methods.

Behavioral ("Black Box") Testing

Another form of dynamic testing is perhaps most familiar to test professionals, the behavioral or "black box" test. Behavioral tests, as the name "black box" implies, focus on what the system does, not how it does it. Some test professionals believe that inside knowledge of how the system works can actually make a behavioral tester less effective, because this understanding can interfere with the ability to test the system in ways that the user would actually use it. However, I've found that I can minimize this problem through careful test design and execution, and insight into how the system works can help behavioral testers find more bugs and write more perceptive bug reports.

Behavioral testing most typically focuses on global issues of workflows, configurations, performance, and so forth. Because of the focus on how the system works, behavioral tests can cover many of the important usage profiles and quality risks. While you can perform behavioral testing of discrete components—and some programming methodologies promote doing so—most behavioral testing focuses on large portions of the system or fully integrated systems. As the system comes together, the system's behaviors start to become most visible. Testers can then gain insight into the user's experience of quality.

Behavioral testing is supported by a variety of automated tools. For example, performance, load, and reliability testing require that testers deliver precisely-calibrated streams of data, transactions, and user inputs to the system under

test. In addition, we often must gather statistics about how the system responded to these streams. In most situations, an automated tool does such testing best. For client/server and browser-based applications, you can find a variety of commercial tools for this purpose. Similarly, commercial tools exist to perform functional testing of many platforms.

Test experts classify these behavioral testing tools as intrusive or non-intrusive. An intrusive test tool runs on the system under test; for a client/server or browser-based application, the test tools typically runs on the client. A non-intrusive test tools runs on a separate system and interacts with the system under test in ways that are, ideally, indistinguishable from one or more real users operating the system.

Manually testing is an important—and, in many teams, primary—behavioral testing approach, too. Manual behavioral testing generally involves insightful planning, careful design, and meticulous result checking. A good manual tester also applies on-the-spot judgment to observed results that an automated tool can't. Skilled manual testers know how to follow a trail of bugs—or blaze a new trail as they go. Elisabeth Hendrickson's bug hunting technique, explained at www.qualitytree.com, and James Bach's charted exploratory testing, which he discusses at www.satisfice.com, are two examples. You can blend traditional scripted testing techniques with the exploratory styles, too.

Another important type of manual behavioral testing—almost a flavor of testing on its own—is what I call *live testing*. In live testing, testers—or users—subject the system to real data, user workflows, field configurations, and other true-to-life conditions. Acceptance testing is one form of live testing common in IT development projects. For market-driven systems, beta testing is a common live test technique. One of my clients produces a complex geological software package. A key part of their behavioral testing is what they call *guest testing*, where their customers send their most proficient users of the system to my client's offices to participate in the final stages of behavioral testing.

Good behavioral testing, logically enough, requires some amount of understanding of the proper behavior of the system. In other words, the test team needs a certain degree of insight into the business or application domain of the system. This is especially true for complex systems like the geological package I mentioned, medical systems, and so forth. However, people with backgrounds only in the application domain—and without any understanding of testing—often fail to understand how quickly an experienced, seasoned tester can come up to speed on the application domain. I once had a banking client whose users expressed surprise at how effective testers without a banking background were at finding bugs in the system.

Testers are flexible like this in part because test professionals often get to be generalists, rather than having to specialize. Testing allows us to see the whole picture, which is why some people choose to be testers rather than programmers. In addition, beyond business domain skills, true test professionals have testing skills. We know how to use automated tools and how to apply

exploratory testing. We know how to analyze equivalence partitions, state transitions, transactions, and input domains, then translate that analysis into effective and efficient manual and automated test cases. We can deconstruct a system and figure out where the bugs live. (These are skills a typical user does not have, which is why behavioral test teams that consist only of users, without any skilled and trained testers, tend to do a relatively poor job of finding bugs.) Good explanations of behavioral test techniques can be found in the classics *Testing Computer Software* by Cem Kaner, Jack Falk, and Hung Nguyen, and (the appropriately named) *Black-Box Testing* by Boris Beizer. The wise and battle-scarred test professional will have these—and many other—books on the desk, ready at hand. Testing as a skill, not just an activity, is finally coming into its own.

Sharing

While I've discussed each of these techniques as separate, the reality is that sharing of data, cases, procedures, and tools is not just possible, but desirable. A test team that focuses on behavioral testing can—and should—borrow structural test tools and data from the programmers—or vice versa. Test techniques are not—or at least should not be—religions, where one must chose one church or another. Test techniques are among the wrenches, screwdrivers, hammers, and other tools of the software professional. Pick the techniques that suit the tasks at hand. If you're a test manager, encourage cross-pollination, re-use, and sharing of techniques.

Automated Or Manual?

I've referred throughout this article to automated tools that can help us apply the various testing techniques. (An unbiased, comprehensive listing of testing tools can be found at www.tejasconsulting.com.) But how important is automation? Isn't manual testing still widely practiced, even by enlightened test groups? Are there circumstances where automation is downright dangerous and inappropriate? The answers are "very," "yes, indeed," and "certainly," as I'll explain in the next article.

Author Biography

Rex Black is President and Principal Consultant of RBCS, Inc. (www.RexBlackConsulting.com), a consultancy that provides testing experts world-wide, serving clients such as Bank One, Cisco, Hitachi, IMG, and Schlumberger in consulting, training, and hands-on implementation. He's written *Managing the Testing Process, Critical Testing Processes*, and numerous articles, along with presenting papers and keynote speeches at international conferences.

Bibliography

B. Beizer, *Software Testing Techniques, 2nd Ed*. New York, Van Nostrand Reinhold, 1990.

B. Beizer, *Black Box Testing.* New York: Wiley, 1995.

R. Black and G. Kubaczkowski, “Mission Made Possible,” Software Testing and Quality Engineering magazine, Volume 4, Issue 4.

D. Freedman and G. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews* . New York: Dorset House, 1990.

B. Hetzel, *The Complete Guide to Software Testing*. New York, Wiley, 1988.

C. Kaner, et al., *Testing Computer Software, 2nd Ed*. New York, Wiley, 1999.

K. Wiegers, *Software Requirements*. Redmond: Microsoft Press, 1999.