informIT

Should We Be Doing More Unit Testing?

Date: Jun 3, 2005 By Michael Kelly.

Quick feedback. More testable code. Finding serious problems early. What's not to like? Michael Kelly pleads the case for unit testing.

While test-driven development and test-first programming seem to be huge in the agile community, they're hard to come by in my local community. I'm the unhappy (and sometimes unknowing) observer in a geographic community that appears to be doing very little unit testing indeed. This is problematic for me and others in my community because we want to generate more excitement and understanding about the practice of automated unit testing. And we don't think we're alone in our suffering; we believe that there are others out there who don't know about unit testing (mostly in the automated test-driven sense of the practice). This article is an attempt to shed some light on why we create unit tests and how they can help.

How Unit Tests Help Developers

Developers might want to automate unit tests for many reasons. These are the most common reasons:

- Unit tests can provide quick feedback to the developer.
- Unit tests can simplify the structure of the system.
- Unit tests can mitigate concerns about the effects of refactoring.
- Unit tests can be used to validate code integration.
- Unit tests may result in more testable code.
- Unit tests can document the code they test.
- Unit tests are typically inexpensive to run and maintain.
- Unit tests report serious problems early.

Providing Quick Feedback

The sooner developers receive feedback on their code, the sooner they can implement changes based on that feedback. Feedback can be as simple as an error or as complex as having to refactor the code because it's just too difficult to develop a comprehensive test for it. An automated unit test provides feedback to the developer before other test systems exist for the code. This feedback is provided in the environment closest to the developer, making it less expensive to fix any errors found. As we all know, errors found by the developer who's writing the code are less expensive to fix than those found and recorded by a tester. The time needed to record, process, and track the error becomes overhead on the project.

Simplifying System Structure

There is an increasing belief that a properly structured system is easy to unit test. A desirable unit test that's difficult to implement is seen as a sign that the system needs improving. As discussed earlier, this immediate feedback is valuable to the developer. Often, developing unit tests will help to focus and clarify thoughts on the code being developed, by forcing issues and ambiguities to the surface before implementation and release. In general, a simpler system is cheaper to maintain and allows developers new to the project to come up to speed more quickly.

Mitigating Refactoring Concerns

Unit tests are typically written at the time of the highest flux and least reliability in the code. By providing quick feedback and reducing system complexity, developers who use unit tests can work confidently during times of flux and make changes with confidence, knowing that any omissions or new errors introduced should be caught by their suite of unit tests. In my experience, developers who have a suite of unit tests for their code introduce fewer errors during project crunch times, when people are working long hours and requirements are changing daily. It's easy for the test team to identify those individuals who developed and maintained unit tests throughout the project.

Validating Code Integration

Unit tests can be reused during code integration to ensure that changes in other developers' code don't adversely affect the code tested via the unit test: "If it worked before, it should work now." While typically not a complete test of integration, a complete test of all the parts of a whole is the logical first step in testing the whole. In many agile development groups, unit tests are executed against every build of the software (which can happen several times a day). It's a big deal if a unit test fails, and tremendous effort is put into fixing the build before "regular" work continues. The unit tests for the project are the first line of defense against bad software; by getting feedback so quickly, project teams can more easily identify which changes caused the tests to fail.

Producing More Testable Code

In the process of developing unit tests, developers may be required to build in test harnesses, logging and debug utilities, and APIs to exercise functionality in isolation. Usually, this functionality can be used downstream in the testing process during integration testing, system testing, regression testing, performance testing, and user acceptance testing. In some of my projects, we've used both log files (developed for unit testing) and application APIs (also developed for unit testing) while developing our automated test scripts. This practice not only allowed us to test faster (due to faster-executing regression scripts), but also to test *better*. We were able to read the logs at runtime to see whether errors were occurring that we weren't seeing at the GUI level of the application.

Documenting Tested Code

Unit tests can be used as a form of executable documentation for the code they test. These tests provide substantial value for programmers doing maintenance—both for programmers trying to understand their own code at a later date, and when looking at someone else's code. The simplicity of a unit test clarifies the intent and the expectations of the code.

In addition, if you're attempting to figure out how to use a specific piece of code, sometimes it's helpful to look at the unit tests for that code to see how they exercise the code. I use this approach when developing Watir scripts, for example. Watir is a free open-source functional testing library for developing automated tests for web applications. Because the tool is constantly under development and new features are added almost daily, it's unrealistic to expect the documentation to remain up to date with the code. To learn how to use a new feature, I often look at the unit tests that were checked in with the feature.

Reducing Testing Costs

Relative to all the other types of testing, unit tests are inexpensive to create, maintain, and run. Tools are typically free or included in the enterprise IDE being utilized, resulting in no additional tool investment. Unit tests are typically coded in the same language as the code they test, which saves on the additional costs associated with maintaining a specific language skill set. Unit tests are typically simple enough that no extra documentation is necessary for their longevity, unlike all other types of tests (with the exception of exploratory testing).

This is not to say that there is no cost to unit testing—just that it's significantly cheaper than the other types of testing traditionally associated with software development.

Reporting Serious Problems Early

For a good many unit tests, failure would indicate a very serious problem. Unlike system tests, which can involve subjectivity and ambiguity, unit tests typically focus on technology issues and coding errors. An error that might be missed as a consequence of failing to unit test (or because of poor unit testing) typically will manifest itself as a high-priority and high-severity problem when encountered in system testing. Unit tests usually won't reveal cosmetic errors that can be postponed to a later release; more often than not, they uncover problems that cause runtime exceptions, crashes, loss of data, and other exciting issues that testers everywhere salivate to find.

How Unit Tests Provide Overall Value

Aside from their immediate value to developers, as previously discussed, unit tests also provide value to the overall testing process:

- Unit tests create a test harness that can be leveraged for other types of testing.
- Unit tests can reduce the overall scope (coverage analysis and risk analysis) of other types of testing.
- Good unit tests remove the necessity for in-depth domain testing and in-depth boundary value analysis.

Creating a Test Harness

I mentioned earlier that unit tests and unit test harnesses can be leveraged elsewhere. Here's are some examples:

- Unit tests can be repurposed to address risks that may not have been envisioned when the tests were written.
- Unit tests can be used as a starting point for APIs used for test automation. They can be used to seed a suite of automated tests.
- Logs developed for unit testing can be leveraged throughout the test lifecycle.
- Test data that's developed or identified can be leveraged throughout the test effort.

All of these uses potentially lower the cost of testing throughout the project.

Reducing Scope for Other Types of Testing

Unit tests can reduce the overall scope (coverage analysis and risk analysis) of other types of testing. For example, system tests can be designed by reviewing the existing unit tests—in some cases, tapping into interfaces that developers had written for their own tests—and can focus on efforts that developers didn't address. System testers should take advantage of unit tests and design their own tests to mitigate risks not already addressed by the unit tests. For example, if you're concerned with code coverage, as many teams are (we won't go into the validity of those concerns), you can run your unit tests using a code-coverage tool and then develop system tests to exercise the code that hasn't already been executed. If you have concerns about specific high-risk features, such as integration with a third-party web service, you can develop system tests that complement the testing already performed at the unit level, instead of duplicating that testing.

Reducing the Necessity for Other Processes

Good unit tests remove the necessity for in-depth domain testing and in-depth boundary value analysis. I stole these two concepts from the mind of Cem Kaner. As a tester, I find them extremely valuable. With the exception of a quick sampling to verify that the right testing was done at the unit level, domain testing or testing at the boundaries for individual components might be omitted entirely, or at the very least drastically reduced. These classes of testing might still be needed at the application's system level, but the trivial cases could be executed up front during unit testing.

This seems to be an excellent place for developers and testers to collaborate. Many developers could benefit from a better understanding of domain and boundary value analysis, but because it's typically outside of their direct focus (get the code working), they may not be aware of these techniques. Testers can help by supplying this knowledge and/or generating the data for the tests. Specialized tools help to facilitate these discussions and activities.

Getting Started with Unit Testing

At this point, you might be thinking, "All of this sounds good, but how do I get started?" Well, I'm glad you asked! The following is a short (but powerful) list of places where you can go for more information on the topic of unit testing. If I've sparked an interest in developing automated unit tests, these resources will actually address the implementation of those tests. Good luck!

For an overview of test-driven development using unit tests, I suggest grabbing a copy of Kent Beck's Test-Driven Development: By Example (Addison-Wesley, 2002, ISBN 0321146530). This is an excellent book with examples that even an admittedly junior developer like me can follow.

If you just need access to a unit test framework for your development language, check the Cunningham & Cunningham wiki. This site has a wonderful amount of information on the topic and maybe the most comprehensive list of links to the different frameworks.

If you're a community-minded individual, check out TestDriven.com. Here you'll find articles, blog feeds, forums, and downloads related to everything that's remotely associated with unit testing. This is an excellent site that I visit almost daily.

NOTE

Many of the ideas presented in this article originated from the work and thoughts of people smarter than me—specifically Kent Beck, James Bach, Brian Marick, Cem Kaner, Jonathan Kohl, Bret Pettichord, Rex Black, and Dave Liebreich—to which I've added my own thoughts and experiences. In addition, this article was written using the notes from a meeting of the Indianapolis Workshop on Software Testing, held in March 2005 on the topic of "Unit Testing." Participants in the workshop included Dana Spears, Michael Kelly, Joshua Rafferty, Cheryl Westrick, Chad Campbell, Jason Halla, and Allen Stoker.

Michael Kelly is a senior consultant for Fusion Alliance, with experience in software development and testing. Mike is currently serving as the program director for the Indianapolis Quality Assurance Association and the membership chair for the Association for Software Testing. You can reach Mike by email at Mike@MichaelDKelly.com*.*

© 2005 Pearson Education, Inc. InformIT. All rights reserved.
800 East 96th Street Indianapolis, Indiana 46240