The Value of Really Dumb Tests

By Steve Berczuk

There are two claims that people who want to adopt unit testing often make when they find themselves writing few tests: Tests that will be useful are too hard and too time-consuming to write, and the code that they want to test seems too trivial to bother with when the problems can be found through inspection. People's initial view of the risks and value of testing strategies isn't always accurate. Unit Tests—by which we mean tests that can be run quickly in a development environment—can be a valuable tool for improving quality. By erring on the side of testing small, testing more, and testing around the items that are difficult to test directly, you can build reliable software and develop the discipline to test better.

Moving Toward Unit Testing

On the surface, unit testing seems like a simple idea. By testing early and in small parts, you can reduce cycle time for development, improve quality by catching problems early, and enable a whole suite of agile practices, such as refactoring and continuously working software. Like any new practice, unit testing effectively is difficult, and it may take time to demonstrate the value of the additional effort.

If you are starting a new project with a team that is experienced with unit testing and disciplined in the art of test-driven development, seeing the benefits to unit testing is easy. Many teams adopt agile practices while working on legacy code. Unit testing code that has not been designed in a manner that makes unit testing easy can be a challenge. (Michael Feathers, in his book *Working Effectively with Legacy Code*, defines legacy code as "code without tests.")

Unit testing can also be a cultural challenge. While the team and management might acknowledge the value of unit testing, it is extra work and extra time. At the beginning, unit testing, like any new skill, can slow the team down as team members learn how to test effectively. Once you've overcome the learning curve, the value of early testing manifests itself in fewer problems later on in the process and not seeing a problem is often something people take for granted.

Even if you overcome the cultural hurdles and agree to try unit testing, you may still find yourself trying to decide how best to test. Too much analysis about where to test before you have experience can lead you to test less than you should to realize the benefits of unit testing.

To successfully adopt unit testing as a practice, start with the tests that seem simple, obvious, or even trivial, and you will find that you benefit more that you expect, create a testing culture, and develop instincts that will allow you test more effectively as time goes on.

Start Simple

When faced with the prospect of testing legacy code, you may find yourself with a dilemma. Often, code written without testing in mind is poorly decomposed and the code to set up a test can be very complex, making it difficult to write tests without external resources such as databases. In these cases, it's often useful to start by testing whatever method on a class seems easy to test, and using those tests to give you the confidence to refactor other parts of code to be more testable.

No method is too trivial to start with. I once was on a project where we spent the better part of a day on a problem that was traced to a class that violated something as basic as the equals/hash code contract on a C++ class. Everyone assumed that this was too simple to be worth a test, easy enough to validate by inspection, and not likely to be incorrect as the methods were (initially) generated by the IDE. But, as code evolved, people forgot to review the code to be sure that the methods were in sync. While code reviews, whether through pair programming or a more structured process, can be valuable for identifying unanticipated problems, an automated test is quicker and more reliable than visual inspection for validating that your code honors low-level contracts that you

already know about. Tests like these are simple to write and can more than pay for themselves if they prevent one day of your team puzzling over an obscure problem.

Think Outside the Code
Another issue, especially with frameworks that rely heavily on configuration, is that the configuration you use for developer testing will, of necessity, differ from that you use in a deployment situation. For example, a developer test may use a simplified database interface, while code that is deployed in a web container will use a real database connection. Unit tests may pass with flying colors, but the application will not run. While it is good practice to start up the complete application before committing code, it is still helpful to have an automated way of identifying common configuration mistakes.

In one project I worked on, we had a good suite of unit tests with excellent code coverage and a good record of builds passing. Yet, every so often, a build that passed the unit tests would not start up when deployed as part of a web application. The failure message was obscure, and often a number of people were blocked when the problem occurred. We traced the problem to errors in the deployment configuration (either a typo or, more often, a reference to a class that was incorrect). While we could have written an integration test to make sure that the application worked, this would have been difficult to run in the context of our integration build. We wrote a simple test that identified the problem by validating that the Spring configuration loaded successfully. After this seemingly low-value, low-effort test, we rarely had problems.

In another case, we traced recurring application startup errors to syntax errors in a large configuration file that was edited by hand. A test that validated the XML file using a validating parser allowed whoever made a change to quickly identify errors and fix them.

In both cases, a simple test caught a problem that had stopped the application from running. These sorts of errors slow down anyone who updates his code and cause people to be reluctant to work with the latest code, which is a barrier to continuous integration. Both of these problems could have been easily caught by "inspection" or "being careful," but programmers are human and mistakes slip by even the most detail-oriented developer, especially when there are a few moving parts. In the end, programmer time is better spent identifying design issues than checking code for syntax errors.

Some of these cases also pointed to a fragile configuration mechanism, which you might want to improve. By keeping track of how often the tests failed during precommit testing, you have data to identify how much effort you can justify spending on architectural approaches to minimizing the risk of configuration errors.

Testing the Trivial
In some cases, unit testing what you need to test can be difficult because of lack of framework support. While I was working on a VXML application, the team did not have access to a good mechanism to write automated tests for the voice response code, so we did most of the functional testing manually by dialing into a voice server. While this was slow, it worked. Interruptions in testing caused by server syntax errors caused the greatest frustration. Since we couldn't test the VXML interaction, we decided that there might be value in validating that the VXML response was always syntactically correct. We used a testing framework to call methods on the server API with a variety of parameters and checked that the returned VXML validated to the DTD for VXML.

While this seems trivial and low value, the practical effect was that those testing the more complicated functionality did not waste time on mistakes that the automated tests caught. This did not solve the problem of testing the voice application in a repeatable way, but it made the manual testing more effective.

Next Steps
Unit testing can be a challenge, both technically and culturally, especially when you have an existing code base to work with. You need to learn both to write tests and to architect code that is both testable and efficient. You want to maximize the effectiveness of the effort that you spend writing tests. Like performance optimization and many other technical aspects of programming, our instincts on where to best spend effort fixing a problem are often incorrect before we gather data on the problem at hand. Start your testing with the attitude that no test is too trivial to write, and when you encounter a problem that seems difficult to test, think about testing configuration and other aspects of the code rather than the code that you think is problematic.

Once you begin to develop an understanding of the value of various tests, you can remove tests that seem like maintenance or of low value and err on the side of not testing things so basic that they really will not fail. But, don't make these decisions until you have enough experience to develop a sense of the cost and benefit of the tests. The quickest way to fail at change is to assume that you know more about the new techniques than you do.

Some resources that will help you to understand how to approach testing problems include XUnit Patterns for ideas on how to test in specific situations, *Continuous Delivery* for understanding the role of unit testing in the larger testing context, and *Agile Architecture* for understanding how to develop testable architectures.

Don't discount the value of simple tests. Most developers can point to a wasted afternoon spent on a coding error that "should not have happened." By starting simple and adding configuration validation to your test suites, you can be more adaptive and agile.

About the Author
Steve Berczuk is an engineer and ScrumMaster at Humedica where he's helping to build next-generation SaaS-based clinical

informatics applications. The author of *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*, he is a recognized expert in software configuration management and agile software development. Steve is passionate about helping teams work effectively to produce quality software. He has an M.S. in operations research from Stanford University and an S.B. in Electrical Engineering from MIT, and is a certified, practicing ScrumMaster. Contact Steve at steve@berczuk.com or visit berczuk.com and follow his blog at steveberczuk.blogspot.com.

StickyMinds.com Weekly Column From 2/7/2011

Close This Window