IEEE Software
www.computer.org/software

Design for Test

Rebecca J. Wirfs-Brock

Vol. 26, No. 5
Sept./Oct. 2009

Editor: Rebecca J. Wirfs-Brock ■ Wirfs-Brock Associates ■ rebecca@wirfs-brock.com

Design for Test

Rebecca J. Wirfs-Brock

Ideas must be put to the test. That's why we make things; otherwise they would be no more than ideas. There is often a huge difference between an idea and its realisation. —Andy Goldsworthy

As developers, we're expected to turn out implementations proven by tests that we or others have written. Doing otherwise is considered unprofessional. But does code that's designed to be testable differ fundamentally from code that isn't? What does it mean to design for test?

Making Code Testable

Advocates of test-driven development (TDD) write tests before implementing any other code. They take to heart Tom Peters' credo, "Test fast, fail fast, adjust fast." Testing guides their design as they implement in short, rapid-fire "write test code—fail the test—write enough code to pass—then pass the test" cycles. Regardless of whether you adhere to TDD design rhythms, writing unit tests forces you to articulate pesky edge cases and clean up your design.

Michael Feathers has cheekily defined a legacy system as code that doesn't have tests. He says that to be testable, code needs appropriate *seams*. In *Working Effectively with Legacy Code* (Prentice Hall, 2005), Michael defines a seam as a place where you can alter your program's behavior without having to rewrite it. Every seam has an enabling point—a place where you can decide to use one behavior over another. There are two main reasons to include these seams:

- so that you can insert test code that probes the state of your running software and
- to isolate code under test from its production environment so that you can exercise it in a controlled testing context.

You can design seams in different ways, ranging from preprocessing flags and conditionals to adjusting class paths and dynamically injecting dependencies between collaborators. You also need to isolate and encapsulate dependencies on the external environment. All these techniques let you insert code that exercises your software without altering the code being tested.

In addition to inserting appropriate test hooks, you should write your code so that it doesn't have unnecessary dependencies on concrete class names, values, and variables—anything that you might want to replace in a test environment. You can do this in many ways—for example, by

- using configurable factories to retrieve service providers,
- declaring and passing along parameters instead of hardwiring references to service providers,
- declaring interfaces that can be implemented by test classes,
- declaring methods as overridable by test methods,
- avoiding references to literal values, and
- shortening lengthy methods by making calls to replaceable helper methods.

In short, you need to provide appropriate test affordances—factoring your design in a way that lets test code interrogate and control the running system.

On the surface, this sounds like nothing more than good design practice. And largely, this is true.

0740-7459/09/$26.00 © 2009 IEEE

But adding the capability to transparently insert test code has consequences. It can add extra wiring, assembly, and interaction steps to your software. Understanding how collaborations are established can become slightly more difficult because wiring and assembly steps are often accomplished by indirect dependency-injection techniques.

This approach can also involve a lot of fiddling and rework if your code wasn't designed this way from the start. My colleague Don Birkley observes,

> *One critical aspect of design for test is to keep classes designed so that in vitro tests are even possible. This involves not only clean well-factored design, but also creating the context objects to supply the "nutrients" and "oxygen" for the objects under test.*

Code that's designed for test must continually be tested. If it isn't, any test affordances you add are purely speculative.

Balancing Test and Product Code

So far we've been talking about testing from the developer's perspective on writing unit tests. But what do testers need from a design? Performance-test engineers need the ability to predictably set up, control, and measure software execution. Sometimes this requires designing extra hooks that let them precisely configure and control characteristics affecting software performance. And sometimes these extensions work their way into products, because sophisticated customers also find performance-tuning capabilities useful.

Testers also like to automate their tests. For this to be feasible, they need predictable, stable behavior at points where they stimulate software and measure its behavior. It can be disastrous when gratuitous design changes break a lot of tests. Unless I know how tests exercise my design (and how they verify its behavior), as a designer I won't know what's fair game to change and what behavior I must preserve. But whatever my software updates, logs, or reports is fair game for a test to examine. However, I need to know what the tests want to examine and whether I think it's reasonable for test code to do so. To reduce both design and test rework, the contractual agreement between what a design produces and what tests consume should be established early. It's much harder to wedge in consistent error-messaging and logging strategies as a design afterthought.

An agile development team's manager was frustrated by the increasing difficulty her team had making any significant design changes to the production system. After building extensive test suites, they were confronted with a tough choice: either make desired changes to product code and break quite a few tests, or preserve the tests and create an awkward design solution. Her team took their best shot at defining what they expected to be the "stable" contract between test and product code. Sometimes you just can't make nontransparent design changes to support new product features. This isn't just specific to tests. Part of evaluating any design change is understanding its impact on the overall system. Tests are just one of the system parts that might be impacted. You can't realistically expect that tests or the software's design won't have to change to accommodate new requirements. To make rational decisions about how to support changing requirements, you need to manage the design of production code, unit tests, and acceptance tests as codependent assets.

Promoting Repeatable Behavior

Tests help define and constrain behaviors. Still, they don't guarantee that software works predictably. But as any experienced designer of complex software knows, the more you can pin down your design and make it exhibit repeatable behavior, the easier it will be to maintain. Nondeterministic behavior makes reproducing certain bugs nearly impossible. So, compiler writers know it's important that, given the same source files, compilers should generate identical, not equivalent, code. Doing something as innocuous as using a nondeterministic hash value, such as object identity, for a table lookup can throw a monkey wrench into the works.

That's also why designers of real-time systems have their own grab bag of established design techniques to make their software behave more deterministically. It's also why those who write random-number generators know to design their code to return the same sequence, given the same seed. And designers of complex calculations take the time to design consistent, stylized code. Reproducible behavior is inherently easier to test. If tests place constraints on a design, so too, should the necessity to reproduce, isolate, and fix bugs.

If you design for testing, debugging ease, and repeatable behavior, does it ever get easier and more intuitive? Or does it always take significant time and effort? Designing for test involves discipline and vigilance. I'm not sure it ever becomes easy. But it can become more routine, especially if you treat the design of tests and of the code that satisfies them as complementary parts of your development process.

Rebecca J. Wirfs-Brock is president of Wirfs-Brock Associates. Contact her at rebecca@wirfs-brock.com; www.wirfs-brock.com.

The more you can pin down your design and make it exhibit repeatable behavior, the easier it will be to maintain.