Professionalism and Test-Driven Development

Robert C. Martin, *Object Mentor*

Test-driven development is a discipline that helps professional software developers ship clean, flexible code that works, on time.

Professional software developers ship clean, flexible code that works—on time. It seems to me that this statement is the minimum standard for professional behavior for software developers. Yet, in my travels as a software consultant, I've met many software developers who don't set the bar this high and instead ship late, buggy, messy, and bloated code.

Actually, I don't think any software developer is truly content to ship code that falls below the bar. The truth is that most developers simply don't know how to reach that bar or don't believe reaching it is possible. All too often they commit to deadlines that they later find they can't meet, and so, through high-stress heroics, they wind up compromising quality to avoid being overly late.

In this article, I'll discuss how test-driven development can help software developers achieve a higher degree of professionalism. TDD isn't a silver bullet and won't suddenly transform you into a sterling professional. It does, however, play a significant role.

The three laws of TDD

TDD practitioners follow these three laws:

- You may not write production code unless you've first written a failing unit test.
- You may not write more of a unit test than is sufficient to fail.
- You may not write more production code than is sufficient to make the failing unit test pass.

These three laws lock you into a tight loop in which you first write a portion of a unit test that fails, and then you write just enough production code to make that test pass. This loop is perhaps two minutes long and almost always ends in success.[1]

Following these laws perfectly doesn't always make sense. Sometimes you'll write a larger test. Sometimes you'll write extra production code. Sometimes you'll write tests after you've written the code to make them pass. Sometimes it'll take more than two minutes to go around the loop. The goal isn't perfect adherence to the laws—it's to decrease the interval between writing tests and production code to a matter of minutes.

0740-7459/07/$25.00 © 2007 IEEE

Counterintuition

In my frequent lectures and courses, I've found that many developers consider TDD to be counterintuitive. The extremely short cycles between writing a test and making it pass go against their practice of writing whole modules and testing them manually afterward. They also argue that all that test writing would be too much work.

However, intuition isn't always the best guide to professional behavior. Take, for example, the case of Ignaz Semmelweis, who in 1847 achieved a six-fold drop in his maternity ward's mortality rate by simply having doctors wash their hands before examining pregnant women:

> *His observations went against the current scientific opinion of the time, which blamed diseases (among other quite odd causes) on an imbalance of the basic "four humours" in the body, a theory known as dyscrasia. It was also "argued" that even if his findings were correct, washing one's hands each time before treating a pregnant woman, as Semmelweis advised, would be too much work. Nor were doctors eager to admit that they had caused so many deaths.*[2]

Certainly the implications of TDD aren't as significant to software professionalism as sterile procedure is to medical professionalism. Nonetheless, parallels exist. TDD changes the way programmers work minute by minute, and it profoundly affects the work's results.

Impediments

Of course, not every environment is conducive to a strict interpretation of the three laws. If you must do batch compiles or download your software into an embedded device, it's difficult to get around that loop every two minutes. You might be modifying vast quantities of hard-to-test legacy code. You might be using a third-party software package that doesn't provide easy test access, such as a COTS system or one of the many Web frameworks. There's no end to the reasons why TDD isn't practiced, given the plethora of impediments.

The solutions to these impediments aren't within this article's scope. I daresay you'll find some of them in other articles in this issue. Others you'll find on the Web or in books. Still others you might need to invent for yourself. You'll always find plenty of impediments to convince you not to write tests, but TDD is an attitude as well as a discipline. The TDD attitude doesn't yield to the impediments; rather, it finds a way to deal with them and write the tests anyway.

This might seem too extreme, and of course, some tests can't or shouldn't be written. But the attitude still holds. Those of us who follow the discipline of TDD don't yield lightly.

Benefits

If you follow the TDD discipline and attitude, you'll write dozens of tests per day, hundreds of tests per month, and thousands of tests per year. Experience shows that these tests will cover more than 90 percent of the production code. They should be kept in one place and should be easy to run. Executing them shouldn't take more than a few minutes.

Over the last few years, Micah Martin and I've been working on an application named FitNesse (www.fitnesse.org). FitNesse is a Web-based application using a Front Controller that defers to servlets that direct views. Downloaded tens of thousands of times, FitNesse consists of 45,000 lines of Java code in nearly 600 files. Almost 20,000 of those lines and 200 of those files are unit tests. Over 1,100 specific test cases contain many thousands of assertions, giving FitNesse test coverage of over 90 percent (as measured by Clover, http://cenqua.com/clover). These tests execute at the touch of an IDE (integrated development environment) button in approximately 30 seconds.

These tests, along with the convenience of easily executing them, have benefits that far exceed simple software verification.

Flexibility

Why don't developers clean up code? They're afraid that they'll break it. The old maxim, "If it ain't broke, don't fix it!" is a common attitude among software developers.

TDD alleviates this fear because you can check any change to the software almost instantly. If the tests all pass, it's unlikely that the change broke something unexpected. The tests make small changes virtually risk free. We found with FitNesse that even large, sweeping changes were low risk. This was likely because we implemented large changes as a sequence of small changes with test-runs in between.

So, TDD lets us clean up messy code without the fear that we'll break something. Using TDD also means that we can change the behavior of one part of the system without risking side effects in other parts. In short, having a quick and

Why don't developers clean up code? They're afraid that they'll break it.

Tests must be kept to the same level of code quality as the production code.

convenient high-coverage test suite enhances our ability to create flexible designs.

In our experience with FitNesse, we've made sweeping architectural and requirements changes with little impact on the rest of the system. By breaking changes into a sequence of tiny steps and keeping tests passing between each step, we've added and changed major features, and altered existing architectural structures with virtually no backlash from the user community due to hidden defects or side effects. The tests have made us fearless about cleaning up and improving our code.

Documentation

How do you learn a third-party framework such as Spring or Hibernate? The manuals certainly help, but the real learning comes by studying (and using) the code examples and sample projects that are typically in the back of those manuals.

The unit tests that TDD causes us to write are much like the material in the back of a manual. Each unit test is an isolated snippet of code that explains how some part of the overall system works. If you want to know how to create a certain object, there's a unit test that creates it. If you want to know how to call a certain API function, there's a unit test that calls it. You describe anything you want to do with the system in a test.

These tests are unambiguous documents, written in a language that programmers understand. They're so formal that they execute, and because they're executed all the time, they can't get out of sync with the application. They're low-level design documents that describe the system's structure and interactions so completely that if you lost the production code, you could recreate it by examining the unit tests and writing the code that makes them pass.

Our experience with FitNesse has justified this view, although not without some issues. We do find that reading the unit tests is a good way to understand a module. The tests are generally short, easy-to-understand, and descriptive snippets of code.

Unfortunately, some FitNesse tests suffer from nonlocal-isms. They derive from base classes that derive from other base classes, each containing utility functions that the tests find useful. Although this is generally good design and reduces redundancy, it makes the tests harder to read.

Some FitNesse tests suffer from huge setup functions that try to be all things to all test cases. These setups make individual tests harder to understand because keeping track of everything that a setup function did is difficult.

Some FitNesse tests are downright bloated. They try to test too much in a single function or to move the system through too many state changes in one session. These tests are tough to read and understand. Their documentation ability is limited.

In short, FitNesse taught us that you need to design tests to be good documents. You need to expend effort to keep them local and readable. You can't treat tests as second-class code. They must be kept to the same level of code quality as the production code.

Minimal debugging

If you're following the three laws within reason and you're operating in a cycle that's just a few minutes long, then you're never more than a few minutes from seeing all the tests run. This means that you'll detect almost any bug that creeps into the system within minutes. You don't need a debugger to find the bug. You don't really need to do much debugging at all. You know where the bug is because you just added it!

Of course, it's not really that easy. Some tests take longer to run than others, forcing a partitioning strategy for running the tests. For example, even the 30 seconds it takes to run FitNesse tests is too long to wait during the normal TDD cycle, so I usually restrict the tests I run to the package I'm modifying. This shrinks the test time to 1 or 2 seconds but means that I only run the whole suite every half-hour or so. So, some bugs could wait 30 minutes before they're discovered.

Even so, in the last three years, Micah and I haven't used a debugger with FitNesse more than a few dozen times. When we do, it's almost always in the context of a failing unit test, making the cause of the problem localized and repeatable, so the session ends quickly. I can't remember a single instance of casting a net of break points or blindly single-stepping through the code in the hope that we'd get a clue to what was going on.

Indeed, in FitNesse's entire history there hasn't been a release that was delayed for quality issues. It's never had the kind of haunting bug that thwarts delivery schedules and makes hash out of project plans. In 45,000 lines of code,

we've never released a fatal, or even a very significant, bug.

Other companies have had similar results. Both Sabre and Workshare have reported a 10× reduction in defects to the field and significantly increased productivity.[3,4]

Joel Spolsky once said that debugging is hard to estimate and can take 100 to 200 percent of the coding time.[5] Whether or not his figures are accurate, we can all agree that debugging is a huge unknown in any software schedule. TDD reduces that unknown by making sure that we discover defects early and that the causes are localized. In short, TDD makes our schedules more predictable by reducing debugging's variability.

Better design

If we follow the three laws within reason, we'll write most of our code in response to a failing unit test. This means that most of our code will be, by definition, testable. But you can't test a function if that function calls others that have unknown or deleterious effects. So, you must decouple the function you're calling and test it independently of the other functions it calls.

This might sound hard to do, but it's really just good object-oriented design. OOD lets you decouple functions from each other, and TDD enforces that decoupling. This results in a highly modular and deeply decoupled software structure—in other words, a good design.

Good designs aren't free, and TDD doesn't guarantee good designs. However, TDD provides powerful impetus to decouple, forcing developers to think through their designs in ways that they otherwise might not.

Early in FitNesse's development, we were certain we'd need a back-end database. We didn't know if it would be MySQL or something else, and we didn't want to decide too early. So, we avoided the issue by creating simple mock data-access functions. Our unit tests used these mocks by passing them to the core FitNesse modules. We hid the mocks behind an interface so that FitNesse wouldn't know that the data-accessors were mocked. Later we needed to test things that simple mocks couldn't support. So, we created a simple in-memory database that hid behind the same interface. FitNesse was and is able to use this without knowing it. (FitNesse still uses an in-memory database for most of its test cases, which is a reason why the unit tests run in 30 seconds.) Still later, we needed true persistence, so we created a simple flat-file database and hid it behind the same interface.

It turned out that the flat-file system sufficed, delivering high performance and good reliability. It was also convenient having flat files to inspect and manage. So in the end, we didn't implement the back-end MySQL database.

A year or so later, a FitNesse user needed to put his FitNesse pages into a MySQL database. FitNesse's highly decoupled design, necessitated by the mocks, made it easy for him to implement the data access interfaces with a back-end database. He got FitNesse up and running with MySQL in less than a day! So much for the notion that you need to decide on databases up front and that they're hard to put in later. FitNesse still ships with a plug-in for MySQL based on that user's work.

Make no mistake; FitNesse has design issues that I think need improvement. TDD didn't turn FitNesse into a design utopia. On the other hand, the deep commitment we made to TDD at the project's start has resulted in highly decoupled code. Peter Kriens blogged, "When I inspected the code, my hands itched… . The code looked so clean and so easy to bundlefy that I could not resist. Turning it into a bundle was indeed trivial; the makers of FitNesse provided a very clean configuration and start/stop interface" (see the full text of Peter's blog to see some of his complaints about the design).[6]

TDD and professionalism

Let's think back on my definition of a software professional: professional software developers ship clean, flexible code that works—on time.

Clean code

TDD can't force you to write clean code, but it can help eliminate the fear that cleaning your code will break it. The ability to quickly run a full suite of tests means that you can feel safe cleaning up any messes that you find—or make. One quick test will tell you if your improvements are doing more harm than good. Following TDD means that you're freer to follow another professional discipline: "Always leave the code in a better state than when you found it."

Flexible code

Nothing, in my experience, makes code more flexible than the sure knowledge that when you

About the Authors

Robert C. Martin (Uncle Bob) is founder and president of Object Mentor in Gurnee, Illinois. His research interests include process improvement and object-oriented software design. He's published more than 100 articles and is a regular speaker at international conferences and trade shows. He served three years as editor in chief of the *C++ Report*, and he was the first chairman of the Agile Alliance. Contact him at unclebob@objectmentor.com.

modify it, it won't break. TDD provides that sure knowledge. That surety is certainly not complete. Bugs can still slip through the net of tests. Indeed, you often hear that you can't prove code correct by testing it. Although this argument is correct, it misses the point that surety is a continuum. When you run a suite of tests that covers 90+ percent of the code and you see all those tests pass, you're very sure that the code works. It's that level of surety, far more than any design or architectural structure, that makes the code flexible.

Code that works

Of all the statements in the definition of professionalism, this is the most important. Again, you can't be completely sure it works, but a professional will perform due diligence to the best of his or her ability. Later changes can inadvertently break code that worked once, but TDD significantly mitigates this problem without greatly increasing the due diligence.

When I run unit tests (and acceptance tests) on FitNesse, I know it works. Indeed, passing these tests is the only criterion for release. We don't have a quality-assurance cycle after "code complete." We continuously run the unit and acceptance tests during development. So long as they keep passing, we know we can make a release at any time.

Not all systems can depend solely on automated tests the way FitNesse does. Some manual, exploratory, system, and live-data testing is generally necessary. Even so, programmers should expect QA to find zero defects. No professional programmer should ever release code to QA and expect a list of bugs back.

On time

TDD can't guarantee that you'll make all your dates. What it can do is eliminate variables. It's much easier to accurately estimate a task when you know you won't be spending much time debugging. It's much easier to know and communicate a task's status when you can see a fraction of the tests passing. It's much easier to know when you can release if you keep all your tests passing at all times. In short, TDD can give you the surety and the information to make being on time a realistic goal.

I wear a green band on my wrist that says "Test First" (see http://butunclebob.com/ArticleS.UncleBob.GreenWristBand). It's there to remind me that I have obligations to my customers and my coworkers:

- A professional doesn't ship code he or she is uncertain of.
- A professional writes clean, flexible code that works.
- A professional is on time.

My green band reminds me that TDD's disciplines are a huge help in meeting professionalism's requirements and that it would therefore be unprofessional of me not to follow them.

References

1. R. Martin, "The Bowling Game Kata," June 2005; www.butunclebob.com/files/downloads/Bowling%20Game%20Kata.ppt.
2. "Ignaz Semmelwiess," Wikipedia entry, Mar. 2007; http://en.wikipedia.org/wiki/Ignaz_Semmelweis.
3. G. Anthes, "Sabre Takes Extreme Measures," *Computerworld*, 29 Mar. 2004; www.computerworld.com/developmenttopics/development/story/0,10801,91646,00.html.
4. D. Putman, "Workshare Technology and eXtreme Programming (XP)," 2000; www.objectmentor.com/resources/articles/workshare.zip.
6. J. Spolsky, "Painless Software Schedules," *Joel on Software*, Mar. 2000; www.joelonsoftware.com/articles/fog0000000245.html.
7. P. Kriens, "Why is Software So Brittle?" OSGi Alliance blog, Aug. 2006; www.osgi.org/blog/2006/08/why-is-software-so-brittle.html.