Learning to Love Unit Testing

Improving development through the practice of testing before you code

BY DAVE THOMAS AND ANDY HUNT

WHY DON'T MORE DEVELopers use unit tests? After all, unit tests help produce better-designed systems and more accurate code. The recent rise of extreme programming (XP) and the Gamma/Beck xUnit testing framework has brought unit testing into the daily conversation of many coders (see this article's StickyNotes for references). But still, many (perhaps the majority of) programmers avoid writing them. This article is an attempt to change those developers' minds.

If you're a developer who isn't currently using unit tests, then this article will show you some reasons to start and, we hope, convince you that writing tests for everything you do will actually make your job easier and your systems better. And if you're a developer who *is* testing as you code, then maybe the article will help you feel just a little smug.

If you are a professional tester or manager, we hope to give you some insight into what it's like to write unit tests as you write code. We'll help you talk with developers about their testing and give you some ammunition to help convince them of the merits of consistent and comprehensive testing.

What qualifies us to talk about this? We're programmers who have used all of the bad developer excuses in this article. But now we know better…honest.

Let's look at unit testing by first sitting in on a development session as a programmer starts to attack a particular problem. We'll watch her use tests to give herself confidence, and to break her work into bite-sized chunks. We'll see how simple it is to write tests, and how the process of "test a little, code a little" makes our developer more productive.

Then we'll look at some of the reasons developers give for *not* writing

QUICK LOOK

- How to handle common reasons for not testing before you code
- 3 surprising benefits of unit testing

unit tests, along with our counter arguments. Finally, we'll look at some surprising benefits that come from unit testing—things that have nothing to do with testing itself.

Test a Little, Code a Little

Anne the developer looks at the whiteboard for her next task. It seems that the server folks need a class that manipulates filenames. First up is a method that will generate a filename with a specified extension given an existing name. Pass it "fred" or "fred.xml," and it generates names such as "fred.txt" or "fred.bak." Not too difficult, she thinks. As Anne programs in Java, and since this is the first of a number of filename manipulation routines, she decides to use a new package to hold the code. She makes a directory to hold the package source, then creates a subdirectory within it to hold the tests (see the sidebar below). Now she'll need to write some code.

Anne has developed a habit when creating new classes—she starts by writing a minimal source file. All this does is declare a package to hold her code, and define an empty class (in this case called **NameMangler**). She'll add the details later. Though the following code is Java specific, Anne's unit testing habits are good examples to follow in general.

```
package com.mycompany.server.fileutil;
public class NameMangler {

}
```

Putting Tests in Place

Decent tests will probably double the number of source files in a project. Clearly you need to manage all this extra code. We favor a simple set of rules (though other people do it differently):

1. Create a "test" subdirectory to hold the tests for the code in a particular directory.

2. Write a separate test file for each source code file.

3. Organize so you can run individual unit tests, all the tests in a file, all the tests in a directory, and all the tests in the system.

4. Ensure that both the building and running of the tests are automated. This means the inspection of the results must be automated, too, wherever possible. Manually checking a log file for the correct values doesn't cut it.

How does this work in practice? For our Java projects, we've started using the Ant build system to manage both our builds and tests. Typically, we give Ant a "test" target that looks for all directories called "test" and runs any programs whose filenames start with "Test" in these directories. By using a simple naming convention, we've arranged for tests to be added to the application's test suite automatically.

If you're not using Ant, look for some similar way to automate your tests. You can integrate the test tools into your favorite IDE. (If it doesn't allow you to add tools, why is it your favorite IDE?) The key is that the complete test suite should never be more than a couple of clicks or keystrokes away.

Then she goes into the "test" subdirectory and writes the start of the unit test. She always writes a basic *smoke test* first, simply creating an object of the class she's testing.

```
package com.mycompany.server.fileutil.test;
import junit.framework.*;
import com.mycompany.server.fileutil.*;

public class TestNameMangler extends TestCase {
    // … standard setup code …

    // Test object creation
    public void test_smoke() {
        NameMangler n = new NameMangler();
    }
}
```

Since Anne uses the JUnit framework (see the StickyNotes for pointers), she can write a Java class containing a number of test methods. In this case, Anne had the name of the test class (**TestNameMangler**) mirror the name of the class she's testing. It contains a single test method (for now), **test_smoke**.

Anyway, that's enough coding; it's time to see if it runs. Anne uses the Ant tool to build her software, so she types "ant test" at a command prompt. It compiles the two new files and runs the entire application's unit tests, including her new one. Everything works.

Now why did she go to all that trouble? There's no real code in place, and yet she ran every test in the system. Well, Anne finds it useful for a number of reasons. First, it serves as a quick sanity check. (Did she get the directories and package names right?) Second, it confirms that the tests are running against the code they are supposed to be testing. Third, it checks that she hasn't inadvertently done something bad to the rest of the system. Finally, it breaks the ice. As a programmer, there's something scary about starting to code and being faced with a blank editor buffer. Writing these few lines takes us beyond that; now we're just adding stuff to an existing code base.

Anne gets on with the task at hand. Her class has a constructor that takes a filename (with or without an extension) as a parameter. The class also provides a method **change_ext** that returns a new filename with a different extension. The first thing she does is to alter the smoke test (passing the filename "fred" to the constructor). While she's in there, she writes another test method that exercises the **change_ext** method (which she has yet to program). The test will pass **NameMangler** a filename with no extension and check that the new extension is correctly appended.

```
public void test_with_no_ext() {
    // Create a new NameMangler to test
    NameMangler n = new NameMangler("fred");
    // and make sure it adds ".txt" to the base name
    assertEquals("fred.txt", n.change_ext("txt"));
}
```

She then compiles and runs the tests again, and the compilation fails. No surprise, because she hasn't yet changed the constructor to take an argument, and there's no method **change_ext**. Anne knew all that, but she compiled anyway, just to see it fail. That way, she'd know that the code she was about to write makes a difference—it will fix a failing test. Writing the test as

early as she did also lets her try the interface to the class she's writing before writing the class itself. In this way, the test acts like any other code that uses her software, so she can see that the interface she's proposing feels right to use.

Anyway, now that she has some failing tests, she writes code in the **NameMangler** class to fix them. She adds a name parameter to the constructor, and creates the **change_ext** method:

```
public NameMangler(String name) {
    this.name = name;
}

public String change_ext(String new_ext) {
    return name + "." + new_ext;
}
```

The **change_ext** method that she writes is trivial (all it does is append the new extension to the existing filename). That's all she needs to pass the current tests, so that's all she does. She runs the tests, and everything passes. She can't help grinning slightly—having tests pass is a real psychological boost, even tests as simple as these.

There's a key issue implicit in this way of working—your tests have to be simple to run, and they have to run fast. You can't afford to run them every minute or so if they take half an hour to complete. This isn't the place for long-running performance tests, or for tests that enumerate every possible combination of input to your lottery number guessing routine.

Humming quietly to herself, Anne continues coding up her class. She starts by writing a new test that tries to change the extension on a filename that already has an extension.

```
public void test_change_existing() {
    NameMangler n = new NameMangler("fred.xml");
    assertEquals("fred.txt", n.change_ext("txt"));
}
```

She runs the test, and it fails—her current implementation doesn't strip off any existing extension before appending the new one. She writes a little more code:

```
public String change_ext(String new_ext) {
    String old = name;
    int dot_pos = old.indexOf('.');
    if (dot_pos > 0)
        old = old.substring(0, dot_pos – 1);
    return old + "." + new_ext;
}
```

Her confident grin fades when she compiles and runs the tests. It fails on the assertion in **test_change_existing**, claiming that the routine returned "fre.txt," not "fred.txt." She quickly finds the errant "–1" and removes it. The tests pass. She sips her coffee while contemplating the next test to write. We'll leave her to it.

So Why Doesn't Everyone Test?

Developers are creative people, and nowhere is their creativity used more passionately than when they try to come up with reasons *not* to test. Here are some common excuses for not doing unit testing and some convincing counter arguments to try.

Writing tests takes too much time

Ask developers why they don't write unit tests and the most common response will be "coding tests takes too much time," or (a recent favorite) "I'm too busy fixing bugs to write tests." They'll say that low-level tests are useless, and what matters is the integration of the whole system. Developers feel they are not productive unless they're coding, and they feel that they're not coding unless they're writing stuff that will execute in the final application. Clearly, this is a shortsighted view. In reality, coding is just a small part of the overall process—typing speed isn't the limiting factor to programmer productivity.

The truth is that unit testing greatly aids integration. Combining many small faulty components into a complex whole is a classic recipe for chaos. The time spent writing unit tests is repaid many times over when the components are integrated. Subsequent lower maintenance also saves time and money. The unit tests then act as a safety net, ensuring that changes do not break existing functionality.

The truth is that unit testing greatly aids **INTEGRATION**. *Combining many small faulty components into a complex whole is a classic recipe for chaos.*

How do we convince developers that testing will save them time? Reading about it clearly doesn't work. The facts have been available for a long time now with little apparent change in developer behavior. The only way we have found that works is to have developers try it. If you're a developer, show others how simple your tests are to write, and how they ultimately save you time and stress. Offer to set up the stubs of unit tests for their code, and help them get over the initial hurdles. If you're a tester, you could remind developers that the bug reports you give them are a great basis for adding new unit tests. After all, they should have a test that fails before they start attacking a bug. Without that test, how will they know that the bug is fixed? And if they don't know, really know, that the bug is fixed, then they risk repeating the whole exercise all over again, wasting everyone's time.

Sometimes, the only way to get people started is to mandate tests. For example, code has to have unit tests before it can be checked in to the source repository. Initial resentment tends to dissipate when developers realize how the tests are actually helping. Unit testing becomes a tool they rely on.

Tests break the flow

Some developers feel that incremental unit testing breaks their flow. They're writing large chunks of complex application code, and they worry that stopping to write a test will make them lose their place. This concern is misguided because once you start writing unit tests, you find that the tests in many ways define the application. Your creative work is largely wrapped up in designing and implementing the tests. The application coding becomes a series of manageable steps, incrementally adding small chunks of code to make the new tests pass. There's no need to keep vast amounts of context in your head as you program. You can see this in the way Anne worked. She made most

of her significant decisions while writing tests—how to package the functionality, the method signatures, and so on.

The difficulty comes from the fact that this isn't how programmers are taught to program. It doesn't feel natural (at first). This discomfort reinforces the feeling that tests are distracting.

The answer is to encourage developers to stick with it. It takes most developers a week to get into the swing of driving their coding from tests. Once it becomes natural, they'll find the constant cycle of test-code-test-code actually reinforces the flow of development.

Testing is unnecessary

Programmers must have a certain level of confidence—every day they face the challenge of creating something out of nothing. Too often, though, confidence translates into swagger—most code runs fine, and when it doesn't, it's easily fixed. Obviously, this four-line routine is correct, so why waste time on a test? On single-person projects, this philosophy sometimes works.

But even the best developers make mistakes. In fact, the best developers are probably responsible for the trickiest bugs. Once these bugs get into the overall application, they're free to interact with other developers' bugs, making diagnosis very difficult. Look at Anne's trivial off-by-one bug. Her code ran fine, it just produced a subtly wrong answer. If the buggy code had been integrated into the full application, it could have resulted in files getting lost, or existing files being overwritten. Other developers would waste time deciding that although it was their code that was generating these wrong files, the fault wasn't theirs.

We've worked with many good developers. Without exception, once these developers started using rigorous unit tests, they discovered that their code had more bugs than they'd realized. Because they now use unit tests, they've become better developers. If anything, their swagger can be even *more* confident.

Testing is too complicated

There is a group of coders who don't write unit tests because they're already at their limit. They feel that adding unit testing to their load would melt them down. These are precisely the developers who benefit most by unit testing.

Because they now use unit tests, they've become **BETTER** *developers. If anything, their swagger can be even* more *confident.*

Our experience is that many of these developers are struggling because they aren't truly in control of what they're doing. They write code by accretion, adding a little functionality here, patching a deficiency there, never really knowing when they're finished. (You can spot this style of development from the code—massive routines full of repetitive conditional statements, desperate comments, and reams of tracing spew.)

Contrast this with Anne's style of working. She may not be any better technically, but she's always in control. All her steps are small, and she always knows when she's done. Her work always follows the same basic pattern—decide on functionality, write a test, then write code to pass the test—so she never feels lost.

Anne's style of testing makes life simpler regardless of coding skill. Simplicity means control, which leads to better code.

I've got a gazillion lines of legacy code

There's no denying it. It is very, very difficult to add unit tests to a big-wad-o'-legacy code. The code probably isn't structured to make testing easy. Even if it is, there's a tremendous overhead in getting the tests in place. A well-tested system will probably have more lines of unit test code than production code, so telling your boss that you want to add a complete set of tests to a large legacy system is likely to result in your spending time testing your résumé, not your code.

Our recommendations here are pragmatic. It's unreasonable to expect a developer working on legacy code to have a complete set of tests, so don't even go there. Instead, look for the most payback. Typically, this comes when you're forced into making changes to the legacy code. As you make those changes, write test cases. If the change is in response to a bug report, first write a test that exposes the bug, then fix the code, then rerun the test to show your fix worked. Since bugs tend to breed in clusters, see if you can work out the chain of circumstances that led to the bug in the first place. If you're working on code that talks to a legacy system via an external interface, write test cases first to make sure the interface works as expected. Do not throw away any of the tests—find a way to keep them with the

What Does a Unit Test Test?

Let's start with a quick definition: Unit tests help you verify a small chunk of code (typically a particular path through a method or function). Unit tests typically do not test application-level functionality—we leave that to integration, acceptance, functional, performance, and other two-dollar-word tests.

So what do you test with a unit test? Some folks have mechanical rules (every public method needs at least one test, every exception needs to be tested, and so on). We prefer a more practical approach. Some public methods, such as attribute getters and setters in Java programs, are so trivial that writing a unit test for them would be a waste of time. (Besides, you'll probably end up using them, and hence testing them indirectly, in other tests.) Other times, a method will encapsulate so much functionality that testing it becomes effectively writing an acceptance test for the application itself, and that's too much for a unit test.

The sweet spot for unit testing is that middle ground—the worker methods that make up the bulk of an application, any methods that do something nontrivial, or methods that may fail when the environment changes (so that when the environment *does* change, you'll know what broke). Normally you'll test publicly accessible methods, but that's not a rule. If you have a private method that's complex, test it too. The earlier you find a problem, the easier it is to diagnose and fix. (Of course, if the method is so complex, perhaps it deserves a class of its own.)

legacy code. This way your set of unit tests will grow gradually over time.

Some code can't be unit tested

Having spent some time countering excuses for not doing unit testing, we have to 'fess up. There is one excuse that carries weight. What can developers do if they're interfacing with hardware, or an external system that's outside their control? Sometimes these external components just won't play ball during testing. They may be too slow, or they may not deliver the same results each time they're called (often for very good reasons—a stock ticker that gave the same price back on every call would make testing easy, but might make regulators suspicious). To overcome this, developers *could* write test harnesses that simulate these external components. But for the more complex interfaces, you have to weigh the benefits against the costs. You can bet that NASA tests its software against test rigs, both software and hardware, but their budget probably exceeds yours.

Any practice that makes development **MORE ACCURATE**, *more maintainable, easier to understand, and more fun must be worth a look.*

In these cases, we again suggest taking a pragmatic approach. Design the software so that as much as possible can be unit tested without relying on these external interfaces. (This decoupling is good programming practice anyway.) Then put in place a solid set of integration tests that do the equivalent of unit testing on the remainder of the code.

User interface code is a special case. Technologies exist that let you perform unit tests right out to the screen, keyboard, and mouse level, but they tend to be inconvenient. Our advice is to design a thin GUI layer, with a well-defined interface to the rest of the application. Test up to this layer, and then leave the GUI testing to the test team.

The Benefits beyond Testing

Many developers don't realize that unit testing does far more for them than simply test code.

First, tests aren't a *tool* to help you get some code right. Tests are an *environment* in which the only code that runs is code that is correct (or at least as correct as the tests are complete). The tests are a continuously available sanity check and safety net. The comedian Steven Wright says, "You know how it feels when you lean back in a chair so far that you start to fall over, and then you catch yourself? That's how I feel all the time." We suspect that many coders share that feeling of instability and imminent danger every time they alter code. Unit tests cure this—you know you haven't botched the system because the whole system's tests still run.

Second, tests affect the *design* of your code. When developers get into the habit of unit testing everything, they discover that some of what they write is hard to test. Sometimes they find themselves having to construct elaborate frameworks to test a single method. Sometimes they find that the method is hard to test because it handles many different cases. And sometimes things are hard to test because the code that does the real work is buried deep in the program's guts. Are these reasons not to test? On the contrary, this kind of discovery is one of the most valuable effects of testing, because the tests are telling you secrets about the structure of the code. Listen to the tests, and refactor (a fancy way of saying restructure) accordingly. The code will be better for it. Methods will have clearly defined functions. The class hierarchies are flatter, and they more accurately reflect the business value of the application. There's less coupling between classes. All of this makes the code easier to write and simpler to maintain.

Third, tests reduce panic. Panic is the developer's worst enemy, particularly when you're looking at client delivery in a week and your system seems to be about as stable as a Hollywood marriage. Every bug fix breaks three other things, and bugs that you could swear you'd fixed two days ago suddenly pop up again. However, you *can* break the cycle of glitches. How? Stop fixing bugs, and start writing tests. You'll discover something surprising. The tests will find many little problems in code that you thought were correct. And as you fix each of the little problems, your overall system stability will improve. Many small bugs lead to chaotic system behavior, and the easiest way to find them is one at a time.

Even if the project sky isn't falling, the pacing that tests impose is a wonderful cure for the frenzied code-like-crazy disease. Testers who use a discipline of unit testing tend to lose that manic look. They appear more confident and changes don't disorient them. They even appear to be having fun.

Any practice that makes development more accurate, more maintainable, easier to understand, and more fun must be worth a look. So if you're a developer, download a copy of JUnit (or a testing framework for whatever language you're using) and start writing tests for your current project.

Don't be put off if things seem hard or unnatural at first. After all, there are things in life that suddenly click—one minute you don't get it, the next you do. As a kid, learning to ride a bike can be a couple of days of frustration and skinned knees. Then suddenly, you just start pedaling and everything works.

Unit testing can be like that. You do it for a while somewhat grudgingly. It works, but what's the big deal? Then suddenly you realize that testing has become part of your coding style—it has entered your blood and settled into your routine. You have become (as Kent Beck and Erich Gamma would say) *test infected*. There is no cure. STQE

Dave Thomas and Andy Hunt are partners in The Pragmatic Programmers, a consultancy specializing in agile development and training. They are authors of the books The Pragmatic Programmer *and* Programming Ruby, *and founding signatories of the Agile Alliance. Their philosophy is that the competence and attitude of individual developers is the single most important component of a successful project. They give talks and offer courses based on this idea worldwide. See more of their articles at* www.pragmaticprogrammer.com

STQE **magazine is produced by STQE Publishing, a division of Software Quality Engineering.**