

Extracted from:

# Pragmatic Unit Testing

---

## in Java with JUnit

This PDF file contains pages extracted from Pragmatic Unit Testing, one of the Pragmatic Starter Kit series of books for project teams. For more information, visit [http://www.pragmaticprogrammer.com/starter\\_kit](http://www.pragmaticprogrammer.com/starter_kit).

**Note:** This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2003, 2004 The Pragmatic Programmers, LLC. All rights reserved.  
No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

## Chapter 1

# Introduction

---

There are lots of different kinds of testing that can and should be performed on a software project. Some of this testing requires extensive involvement from the end users; other forms may require teams of dedicated Quality Assurance personnel or other expensive resources.

But that's not what we're going to talk about here.

Instead, we're talking about *unit testing*: an essential, if often misunderstood, part of project and personal success. Unit testing is a relatively inexpensive, easy way to produce better code, faster.

Many organizations have grand intentions when it comes to testing, but tend to test only toward the end of a project, when the mounting schedule pressures cause testing to be curtailed or eliminated entirely.

Many programmers feel that testing is just a nuisance: an unwanted bother that merely distracts from the real business at hand—cutting code.

Everyone agrees that more testing is needed, in the same way that everyone agrees you should eat your broccoli, stop smoking, get plenty of rest, and exercise regularly. That doesn't mean that any of us actually do these things, however.

But unit testing can be much more than these—while you might consider it to be in the broccoli family, we're here to tell

you that it's more like an awesome sauce that makes everything taste better. Unit testing isn't designed to achieve some corporate quality initiative; it's not a tool for the end-users, or managers, or team leads. Unit testing is done by programmers, for programmers. It's here for our benefit alone, to make our lives easier.

Put simply, unit testing alone can mean the difference between your success and your failure. Consider the following short story.

## 1.1 Coding With Confidence

Once upon a time—maybe it was last Tuesday—there were two developers, Pat and Dale. They were both up against the same deadline, which was rapidly approaching. Pat was pumping out code pretty fast; developing class after class and method after method, stopping every so often to make sure that the code would compile.

Pat kept up this pace right until the night before the deadline, when it would be time to demonstrate all this code. Pat ran the top-level program, but didn't get any output at all. Nothing. Time to step through using the debugger. Hmm. That can't be right, thought Pat. There's no *way* that this variable could be zero by now. So Pat stepped back through the code, trying to track down the history of this elusive problem.

It was getting late now. That bug was found and fixed, but Pat found several more during the process. And still, there was no output at all. Pat couldn't understand why. It just didn't make any sense.

Dale, meanwhile, wasn't churning out code nearly as fast. Dale would write a new routine and a short test to go along with it. Nothing fancy, just a simple test to see if the routine just written actually did what it was supposed to do. It took a little longer to think of the test, and write it, but Dale refused to move on until the new routine could prove itself. Only then would Dale move up and write the next routine that called it, and so on.

Dale rarely used the debugger, if ever, and was somewhat puzzled at the picture of Pat, head in hands, muttering various evil-sounding curses at the computer with wide, bloodshot eyes staring at all those debugger windows.

The deadline came and went, and Pat didn't make it. Dale's code was integrated and ran almost perfectly. One little glitch came up, but it was pretty easy to see where the problem was. Dale fixed it in just a few minutes.

Now comes the punch line: Dale and Pat are the same age, and have roughly the same coding skills and mental prowess. The only difference is that Dale believes very strongly in unit testing, and tests every newly-crafted method before relying on it or using it from other code.

Pat does not. Pat "knows" that the code should work as written, and doesn't bother to try it until most of the code has been written. But by then it's too late, and it becomes very hard to try to locate the source of bugs, or even determine what's working and what's not.

## 1.2 What is Unit Testing?

A *unit test* is a piece of code written by a developer that exercises a very small, specific area of functionality of the code being tested. Usually a unit test exercises some particular method in a particular context. For example, you might add a large value to a sorted list, then confirm that this value appears at the end of the list. Or you might delete a pattern of characters from a string and then confirm that they are gone.

Unit tests are performed to prove that a piece of code does what the developer thinks it should do.

The question remains open as to whether that's the right thing to do according to the customer or end-user: that's what acceptance testing is for. We're not really concerned with formal validation and verification or correctness just yet. We're really not even interested in performance testing at this point. All we want to do is prove that code does what we intended, and so we want to test very small, very isolated pieces of functionality. By building up confidence that the individual pieces

work as expected, we can then proceed to assemble and test working systems.

After all, if we aren't sure the code is doing what we think, then any other forms of testing may just be a waste of time. You still need other forms of testing, and perhaps much more formal testing depending on your environment. But testing, as with charity, begins at home.

### **1.3 Why Should I Bother with Unit Testing?**

Unit testing will make your life easier. It will make your designs better and drastically reduce the amount of time you spend debugging.

In our tale above, Pat got into trouble by assuming that lower-level code worked, and then went on to use that in higher-level code, which was in turn used by more code, and so on. Without legitimate confidence in any of the code, Pat was building a “house of cards” of assumptions—one little nudge at the bottom and the whole thing falls down.

When basic, low-level code isn't reliable, the requisite fixes don't stay at the low level. You fix the low level problem, but that impacts code at higher levels, which then need fixing, and so on. Fixes begin to ripple throughout the code, getting larger and more complicated as they go. The house of cards falls down, taking the project with it.

Pat keeps saying things like “that's impossible” or “I don't understand how that could happen.” If you find yourself thinking these sorts of thoughts, then that's usually a good indication that you don't have enough confidence in your code—you don't know for sure what's working and what's not.

In order to gain the kind of code confidence that Dale has, you'll need to ask the code itself what it is doing, and check that the result is what you expect it to be.

That simple idea describes the heart of unit testing: the single most effective technique to better coding.

## 1.4 What Do I Want to Accomplish?

It's easy to get carried away with unit testing because it's so much fun, but at the end of the day we still need to produce production code for customers and end-users, so let's be clear about our goals for unit testing. First and foremost, you want to do this to make your life—and the lives of your teammates—easier.

### Does It Do What I Want?

Fundamentally, you want to answer the question: “Is the code fulfilling my intent?” The code might well be doing the wrong thing as far as the requirements are concerned, but that's a separate exercise. You want the code to prove to you that it's doing exactly what **you** think it should.

### Does It Do What I Want All of the Time?

Many developers who claim they do testing only ever write one test. That's the test that goes right down the middle, taking the “one right path” through the code where everything goes perfectly.

But of course, life is rarely that cooperative, and things don't always go perfectly: exceptions get thrown, disks get full, network lines drop, buffers overflow, and—heaven forbid—we write bugs. That's the “engineering” part of software development. Civil engineers must consider the load on bridges, the effects of high winds, of earthquakes, floods, and so on. Electrical engineers plan on frequency drift, voltage spikes, noise, even problems with parts availability.

You don't test a bridge by driving a single car over it right down the middle lane on a clear, calm day. That's not sufficient. Similarly, beyond ensuring that the code does what you want, you need to ensure that the code does what you want *all of the time*, even when the winds are high, the parameters are suspect, the disk is full, and the network is sluggish.

## Can I Depend On It?

Code that you can't depend on is useless. Worse, code that you *think* you can depend on (but turns out to have bugs) can cost you a lot of time to track down and debug. There are very few projects that can afford to waste time, so you want to avoid that “one step forward two steps back” approach at all costs, and stick to moving forward.

No one writes perfect code, and that's okay—as long you know where the problems exist. Many of the most spectacular software failures that strand broken spacecraft on distant planets or blow them up in mid-flight could have been avoided simply by knowing the limitations of the software. For instance, the Ariane 5 rocket software re-used a library from an older rocket that simply couldn't handle the larger numbers of the higher-flying new rocket.<sup>1</sup> It exploded 40 seconds into flight, taking \$500 million dollars with it into oblivion.

We want to be able to depend on the code we write, and know for certain both its strengths and its limitations.

For example, suppose you've written a routine to reverse a list of numbers. As part of testing, you give it an empty list—and the code blows up. The requirements don't say you have to accept an empty list, so maybe you simply document that fact in the comment block for the method and throw an exception if the routine is called with an empty list. Now you know the limitations of code right away, instead of finding out the hard way (often somewhere inconvenient, such as in the upper atmosphere).

## Does it Document my Intent?

One nice side-effect of unit testing is that it helps you communicate the code's intended use. In effect, a unit test behaves as executable documentation, showing how you expect the code to behave under the various conditions you've considered.

---

<sup>1</sup>For aviation geeks: The numeric overflow was due to a much larger “horizontal bias” due to a different trajectory that increased the horizontal velocity of the rocket.

Team members can look at the tests for examples of how to use your code. If someone comes across a test case that you haven't considered, they'll be alerted quickly to that fact.

And of course, executable documentation has the benefit of being correct. Unlike written documentation, it won't drift away from the code (unless, of course, you stop running the tests).

## 1.5 How Do I Do Unit Testing?

Unit testing is basically an easy practice to adopt, but there are some guidelines and common steps that you can follow to make it easier and more effective.

The first step is to decide how to test the method in question—before writing the code itself. With at least a rough idea of how to proceed, you proceed to write the test code itself, either before or concurrently with the implementation code.

Next, you run the test itself, and probably all the other tests in that part of the system, or even the entire system's tests if that can be done relatively quickly. It's important that **all the tests pass**, not just the new one. You want to avoid any collateral damage as well as any immediate bugs.

Every test needs to determine whether it passed or not—it doesn't count if you or some other hapless human has to read through a pile of output and decide whether the code worked or not. You want to get into the habit of looking at the test results and telling at a glance whether it all worked. We'll talk more about that when we go over the specifics of using unit testing frameworks.

## 1.6 Excuses For Not Testing

Despite our rational and impassioned pleas, some developers will still nod their heads and agree with the need for unit testing, but will steadfastly assure us that *they* couldn't possibly do this sort of testing for one of a variety of reasons. Here are some of the most popular excuses we've heard, along with our rebuttals.



### Joe Asks...

#### What's collateral damage?

*Collateral damage* is what happens when a new feature or a bug fix in one part of the system causes a bug (damage) to another, possibly unrelated part of the system. It's an insidious problem that, if allowed to continue, can quickly render the entire system broken beyond anyone's ability to fix.

We sometime call this the "Whac-a-Mole" effect. In the carnival game of Whac-a-Mole, the player must strike the mechanical mole heads that pop up on the playing field. But they don't keep their heads up for long; as soon as you move to strike one mole, it retreats and another mole pops up on the opposite side of the field. The moles pop up and down fast enough that it can be very frustrating to try to connect with one and score. As a result, players generally flail helplessly at the field as the moles continue to pop up where you least expect them.

Widespread collateral damage to a code base can have a similar effect.

**It takes too much time to write the tests** This is the number one complaint voiced by most newcomers to unit testing. It's untrue, of course, but to see why we need to take a closer look at where you spend your time when developing code.

Many people view testing of any sort as something that happens toward the end of a project. And yes, if you wait to begin unit testing until then it will definitely take too long. In fact, you may not finish the job until the heat death of the universe itself.

At least it will feel that way: it's like trying to clear a couple of acres of land with a lawn mower. If you start early on when there's just a field of grasses, the job is easy. If you wait until later, when the field contains thick, gnarled trees and dense, tangled undergrowth, then the job becomes impossibly difficult.

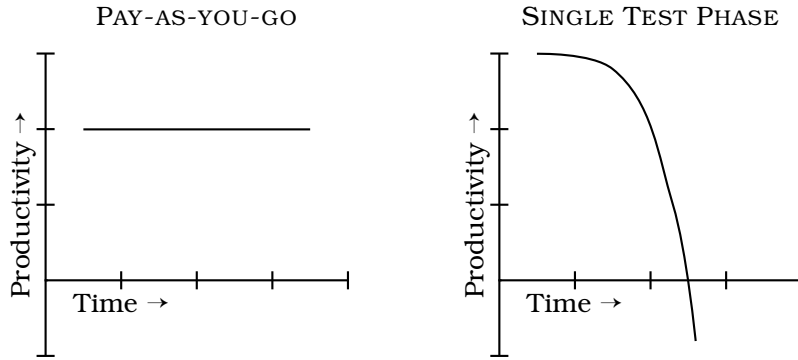


Figure 1.1: COMPARISON OF PAYING-AS-YOU-GO VS. HAVING A SINGLE TESTING PHASE

Instead of waiting until the end, it's far cheaper in the long run to adopt the “pay-as-you-go” model. By writing individual tests with the code itself as you go along, there's no crunch at the end, and you experience fewer overall bugs as you are generally always working with tested code. By taking a little extra time all the time, you minimize the risk of needing a huge amount of time at the end.

You see, the trade-off is not “test now” versus “test later.” It's linear work now versus exponential work and complexity trying to fix and rework at the end: not only is the job larger and more complex, but now you have to re-learn the code you wrote some weeks or months ago. All that extra work kills your productivity, as shown in Figure 1.1.

Notice that testing isn't free. In the pay-as-you-go model, the effort is not zero; it will cost you some amount of effort (and time and money). But look at the frightening direction the right-hand curve takes over time—straight down. Your productivity might even become negative. These productivity losses can easily doom a project.

So if you think you don't have time to write tests in addition to the code you're already writing, consider the following questions:

1. How much time do you spend debugging code that you or others have written?
2. How much time do you spend reworking code that you thought was working, but turned out to have major, crippling bugs?
3. How much time do you spend isolating a reported bug to its source?

For most people who work without unit tests, these numbers add up fast, and will continue to add up even faster over the life of the project. Proper unit testing dramatically reduces these times, which frees up enough time so that you'll have the opportunity to write all of the unit tests you want—and maybe even some free time to spare.

**It takes too long to run the tests** It shouldn't. Most unit tests should execute extremely quickly, so you should be able to run hundreds, even thousands of them in a matter of a few seconds. But sometimes that won't be possible, and you may end up with certain tests that simply take too long to conveniently run all of the time.

In that case, you'll want to separate out the longer-running tests from the short ones. Only run the long tests once a day, or once every few days as appropriate, and run the shorter tests constantly.

**It's not my job to test my code** Now here's an interesting excuse. Pray tell, what is your job, exactly? Presumably your job, at least in part, is to create working code. If you are throwing code over the wall to some testing group without any assurance that it's working, then you're not doing your job. It's not polite to expect others to clean up our own messes, and in extreme cases submitting large volumes of buggy code can become a "career limiting" move.

On the other hand, if the testers or QA group find it very difficult to find fault with your code, your reputation will grow rapidly—along with your job security!

**I don't really know how the code is supposed to behave so I can't test it** If you truly don't know how the code is supposed to behave, then maybe this isn't the time to be writing it. Maybe a prototype would be more appropriate as a first step to help clarify the requirements.

If you don't know what the code is supposed to do, then how will you know that it does it?

**But it compiles!** Okay, no one *really* comes out with this as an excuse, at least not out loud. But it's easy to get lulled into thinking that a successful compile is somehow a mark of approval, that you've passed some threshold of goodness.

But the compiler's blessing is a pretty shallow compliment. It can verify that your syntax is correct, but it can't figure out what your code should do. For example, the Java compiler can easily determine that this line is wrong:

```
public static void main(String args[]) {
```

It's just a simple typo, and should be `static`, not `static`. That's the easy part. But now suppose you've written the following:

```
public void addit(Object anObject){
    ArrayList myList = new ArrayList();
    myList.add(anObject);
    myList.add(anObject);
    // more code...
}
```

Main.java

Did you really mean to add the same object to the same list twice? Maybe, maybe not. The compiler can't tell the difference, only you know what you've intended the code to do.<sup>2</sup>

**I'm being paid to write code, not to write tests** By that same logic, you're not being paid to spend all day in the debugger, either. Presumably you are being paid to write *working* code, and unit tests are merely a tool toward that end, in the same fashion as an editor, an IDE, or the compiler.

---

<sup>2</sup>Automated testing tools that generate their own tests based on your existing code fall into this same trap—they can only use what you wrote, not what you meant.

**I feel guilty about putting testers and QA staff out of work**

Not to worry, you won't. Remember we're only talking about *unit testing*, here. It's the barest-bones, lowest-level testing that's designed for us, the programmers. There's plenty of other work to be done in the way of functional testing, acceptance testing, performance and environmental testing, validation and verification, formal analysis, and so on.

**My company won't let me run unit tests on the live system**

Whoa! We're talking about developer unit-testing here. While you might be able to run those same tests in other contexts (on the live, production system, for instance) *they are no longer unit tests*. Run your unit tests on your machine, using your own database, or using a mock object (see Chapter 6).

If the QA department or other testing staff want to run these tests in a production or staging environment, you might be able to coordinate the technical details with them so they can, but realize that they are no longer unit tests in that context.

## 1.7 Roadmap

Chapter 2, *Your First Unit Tests*, contains an overview of test writing. From there we'll take a look at the specifics of *Writing Tests in JUnit* in Chapter 3. We'll then spend a few chapters on how you come up with *what* things need testing, and how to test them.

Next we'll look at the important properties of good tests in Chapter 7. We then talk about what you need to do to use testing effectively in your project in Chapter 8. This chapter also discusses how to handle existing projects with lots of legacy code. Chapter 9, *Design Issues*, then looks at how testing can influence your application's design (for the better).

The appendices contain additional useful information: a look at common unit testing problems, a note on installing JUnit, a sample JUnit skeleton program, and a list of resources including the bibliography. We finish off with a summary card containing highlights of the book's tips and suggestions.

So sit back, relax, and welcome to the world of better coding.