Extracted from:

Pragmatic Unit Testing

in C# with NUnit, Second Edition

Chapter 1

Introduction

Lots of different kinds of testing 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 going to talk 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.

Unit testing is the practice of using small bits of code to exercise the code you've written. In this book, we'll be using the NUnit testing framework to help manage and run these little bits of code.

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

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.

In fact, many programmers even think testing is a nuisance—an unwanted bother that merely distracts from the real business at hand, which is cutting code.

But unit testing can be much more than a nuisance—although you might consider it to be in the broccoli family, we're here to tell you 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, managers, or team leads. Unit testing is done by programmers, for programmers. It's here for our benefit alone and can make our lives easier.

Put simply, unit testing 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 and 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. It was 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. Pat found and fixed the bug, 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. It was nothing fancy... just a simple test to see whether 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 with the other components and ran almost perfectly.[1] One little glitch came up, but it was pretty easy to see where the problem was. Dale fixed the bug 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 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 completed. 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 who exercises a very small, specific area of functionality in the code being tested. Usually a unit test exercises some particular method in a particular context. For example, we might add a large value to a sorted list and then confirm this value appears at the end of the list. Or we 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

[1]Thanks to the fact Dale had been continuously integrating via the unit tests all along.

want to do is prove that code does what we intended,[2] 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. We still need other forms of testing and perhaps much more formal testing depending on our environment. But testing, as with charity, begins at home.

1.3 Why Should We Bother with Unit Testing?

Unit testing will make our lives easier.

Please say that with us, out loud. Unit testing will make our lives easier. That's why we're here. It will make our designs better and drastically reduce the amount of time we spend debugging. We like to write code, and time wasted on debugging is time spent not writing code.

In our earlier tale, Pat got into trouble by assuming that lower-level code worked and then using 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. We fix the low-level problem, but that impacts code at higher levels, which then needs 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 we find ourselves thinking these sorts of thoughts, then it's usually a good indication that we don't have enough confidence in our code—we don't know for sure what's working and what's not.

[2]You also need to ensure you're intending the right thing; see [SH06].

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 we expect it to be. Dale's confidence doesn't come from the fact he knows the code forward and backward at all times; it comes from the fact that he has a safety net of tests that verify things work the way he thought they should.

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

1.4 What Do We Want to Accomplish?

It's easy to get carried away with unit testing because the confidence it instills makes coding 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. We want to do this to make our lives—and the lives of your teammates—easier.

And of course, executable documentation in the form of clearly written unit test code has the benefit of being self-verifiably correct without much effort beyond writing it the first time. Unlike traditional paper-based documentation, it won't drift away from the code (unless, of course, we stop running the tests or let them continuously fail).

Does It Do What We Want?

Fundamentally, we want to answer this question: "Is the code fulfilling our intent?" The code might well be doing the wrong thing as far as the requirements are concerned, but that's a separate exercise. We want the code to prove to us that it's doing exactly what *we* think it should.

Does It Do What We 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 well-known "happy 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, the effects of earthquakes, the effects of floods, and so on. Electrical engineers plan on frequency drift, voltage spikes, noise, and even problems with parts availability.

We 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, and the fact we succeeded is just a coincidence.[3] Beyond ensuring that the code does what we want, we need to ensure that the code does what we 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 We Depend on It?

Code that we can't depend on is not particularly useful. Worse, code that we *think* we can depend on (but turns out to have bugs) can cost us a lot of time to track down and debug. Few projects can afford to waste time, so we want to avoid that "one step forward, two steps back" approach at all costs and instead stick to moving forward.

No one writes perfect code, and that's OK—as long as we know where the problems exist. Many of the most spectacular software failures that strand broken spacecraft on distant planets or blow them up in midflight could have been avoided simply by knowing the limitations of the software. For instance, the Arianne 5 rocket software reused a library from an older rocket that simply couldn't handle the larger numbers of the higher-flying new rocket.[4] It exploded 40 seconds into flight, taking 500 million dollars with it into oblivion.

[3]See *Programming by Coincidence* in [HT00].

[4]For aviation geeks: The numeric overflow was because of a much larger "horizontal bias," which was in turn because of a different trajectory that increased the horizontal velocity of the rocket.

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 we've written a routine to reverse a list of numbers. As part of testing, we give it an empty list—and the code blows up. The requirements don't say we have to accept an empty list, so maybe we simply document that in the comment block for the method and throw an exception if the routine is called with an empty list. Now we 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 Our Intent?

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

Current and future team members can look at the tests for examples of how to use our code. If someone comes across a test case we haven't considered, we'll be alerted quickly to that fact.

1.5 How Do We Do Unit Testing?

Unit testing is basically an easy practice to adopt; we can follow some guidelines and common steps 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, we can then write the test code itself, either before or concurrently with the implementation code. If we're writing unit tests for existing code, that's fine too, but we may find we need to refactor[5] it more often than with new code in order to make things testable.

[5]Refactoring is the process of making small, deterministic changes to the code to reduce coupling and eliminate duplication, without changing the behavior of the code [FBB+99].

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 easily fix.

We sometimes 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. The root of the problem is usually some kind of inappropriate coupling, coming in forms such as global state via static variables or false singletons, circular object or class dependencies, and so on. Eliminate them early to avoid implicit dependencies on this abhorrent practice in other parts of the code.

Next, we 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. This kind of basic *regression testing* helps us avoid any collateral damage as well as any immediate, local bugs.

Every test needs to determine whether it passed—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. If you can eyeball it, you can use a code assertion to test it.

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.

"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 talk about where we spend our time when developing code.

Many people view testing of any sort as something that happens toward the end of a project. And yes, if we wait to begin unit testing until then, it will definitely take longer than it would otherwise. In fact, we 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 we start early when there's just a field of grasses, the job is easy. If we wait until later, when the field contains thick, gnarled trees and dense, tangled undergrowth, then the job becomes impossibly difficult by hand—we need bulldozers and lots of heavy equipment.

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 we go along, there's no crunch at the end, and we experience fewer overall bugs because we are generally always working with tested code. By taking a little extra time all the time, we 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.

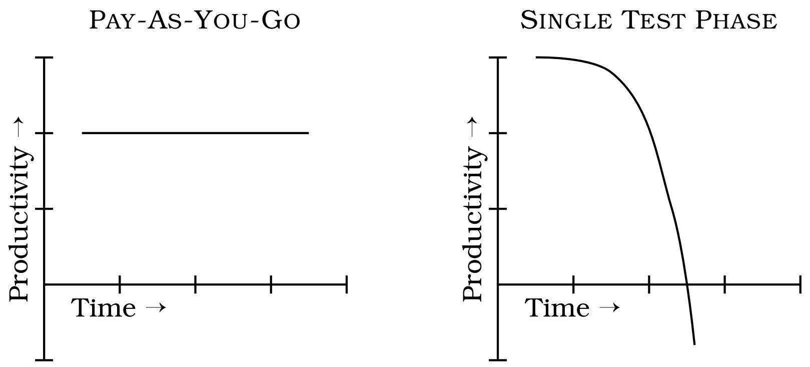


Figure 1.1: Comparison of paying-as-you-go versus having a single testing phase

Not only is the job larger and more complex, but now we have to relearn the code we wrote some weeks or months ago. All that extra work kills our productivity, as shown in Figure 1.1. These productivity losses can easily doom a project or developer to being perpetually 90% done.

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 curve on the right takes over time—straight down. Our 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:

- How much time do you spend debugging code that you or others have written?
- How much time do you spend reworking code that you thought was working but turned out to have major, crippling bugs?
- 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 span of the project. Proper unit testing can dramatically reduce these times, freeing up enough time so that we'll have the opportunity to write all the unit tests we 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 in the blink of an eye, so we should be able to run hundreds or even thousands of them in a matter of a few seconds. But sometimes that won't be possible, and we may end up with certain tests that simply take too long to run conveniently all of the time.

In that case, we'll want to separate the longer-running tests from the short ones. NUnit has functionality that handles this nicely, which we'll talk about more later. Run the long tests only in the automated build or manually at the beginning of the day while catching up on email, and run the shorter tests constantly at every significant change or before every commit to the source repository.

"My legacy code is impossible to test." Many people offer the excuse that they can't possibly do unit testing because the existing, legacy code base is such a tangled mess that it's impossible to get into the middle of it and create an individual test. Testing even a small part of the system might mean we have to drag the *entire* system along for the ride, and making any changes is a fragile, risky business.[6]

The problem isn't with unit testing, of course; the problem is with the poorly written legacy code. We'll have to *refactor*—incrementally redesign and adapt—the legacy code to untangle the mess. Note that this doesn't really qualify as making changes just for the sake of testing. The real power of unit tests is the design feedback that, when acted upon appropriately, will lead to better object-oriented designs.

Coding in a culture of fear because we are paralyzed by legacy code is not productive; it's bad for the project, bad for the programmers, and ultimately bad for business. Introducing unit testing helps break that paralysis.

[6]See [Fea04] for details on working effectively with legacy code.

"It's not my job to test my code." Now here's an interesting excuse. Pray tell, what *is* our job exactly? Presumably our job, at least in part, is to create working, maintainable code. If we are throwing code over the wall to some testing group without any assurance that it's working, then we'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 difficult to find fault with our code, our reputation will grow rapidly—along with our job security!

"I don't really know how the code is supposed to behave, so I can't test it." If we truly don't know how the code is supposed to behave, then maybe this isn't the time to be writing it.[7] Maybe a prototype would be more appropriate as a first step to help clarify the requirements.

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

"But it compiles!" OK, 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 and that we'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 C# compiler can easily determine that this line is wrong:

```
statuc void Main() {
```

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

```
public void Addit(Object anObject) {
  List myList = new List();
  myList.Add(anObject);
  myList.Add(anObject);
  // more code...
}
```

Main.cs

[7] See [HT00] or [SH06] for more on learning requirements.

Did we really mean to add the same object to the same list twice? Maybe, maybe not. The compiler can't tell the difference; only we know what we intended the code to do.[8]

"I'm being paid to write code, not to write tests." By that same logic, we're not being paid to spend all day in the debugger either. Presumably we 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.

"I feel guilty about putting testers and QA staff out of work." Don't worry, we won't. Remember we're talking only 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. Although 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 wants to run these tests in a production or staging environment, you might be able to coordinate the technical details with that department, but realize that they are no longer unit tests in that context.

"Yeah, we unit test already." Unit testing is one of the practices that is typically marked by effusive and consistent enthusiasm. If the team isn't enthusiastic, maybe they aren't doing it right. See whether you recognize any of the following warning signs:

- Unit tests are in fact integration tests, requiring lots of setup and test code, taking a long time to run, and

[8]Automated testing tools that generate their own tests based on your existing code fall into this same trap—they can use only what we wrote, not what we meant.

accessing resources such as databases and services on the network.

- Unit tests are scarce and test only one path, don't test for exceptional conditions (no disk space and so on), and don't really express what the code is supposed to do.
- Unit tests are not maintained; tests are ignored (or deleted) forever if they start failing, or no new unit tests are added, even when bugs are encountered that illustrate holes in the coverage of the unit tests.

If you find any of these symptoms, then your team is not unit testing effectively or optimally. Have everyone read up on unit testing again, go to some training, or try pair programming to get a fresh perspective.

1.7 Road Map

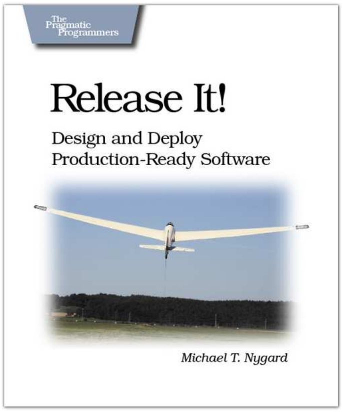
Chapter 2, *Your First Unit Tests*, contains an overview of test writing. From there we'll take a look at the specifics of writing tests in NUnit in Chapter 3. We'll then spend a few chapters on how you come up with *what* things need testing and how to test them.

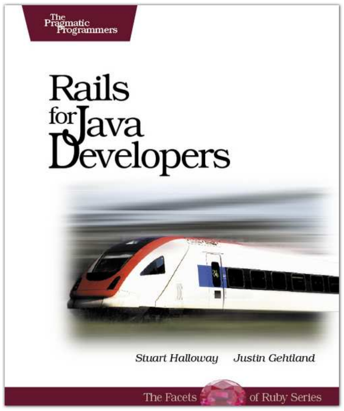
Next we'll look at the important properties of good tests in Chapter 7, followed by what we need to do to use testing effectively in projects in Chapter 8. This chapter also discusses how to handle existing projects with legacy code.

We'll then talk about how testing can influence an application's design (for the better) in Chapter 9, *Design Issues*. We'll then wrap up with an overview of GUI testing in Chapter 10.

The appendixes contain additional useful information: a look at common unit testing problems, extending NUnit itself, a note on installing NUnit, and a list of resources including the bibliography. We finish off with a summary of the book's tips and suggestions.

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


The Pragmatic Programmers
Release It!
Design and Deploy Production-Ready Software
Michael T. Nygard


The Pragmatic Programmers
Rails for Java Developers
Stuart Halloway
Justin Gehtland
The Facets of Ruby Series