the art of
UNIT TESTING
with Examples in .NET

MANNING

ROY OSHEROVE

The Art of Unit Testing
with Examples in .NET

by Roy Osherove

Chapter 1

Brief contents

Getting started

This part of the book covers the basics of unit testing.

In chapter 1, we'll define what "good" unit testing means and compare it with integration testing, and we'll take a brief look at test-driven development and its role in relation to unit testing.

Then, in chapter 2, we'll take a stab at writing our first unit test using NUnit. We'll get to know NUnit's basic API, how to assert things, and how to run the test in the NUnit test runner.

1

The basics of unit testing

This chapter covers

- *Defining unit testing and integration testing*
- *Exploring a simple unit-testing example*
- *Exploring text-driven development*

There's always a first step: the first time you wrote a program, the first time you failed a project, and the first time you succeeded in what you were trying to accomplish. You never forget your first time, and I hope you won't forget your first tests. You may have already written some tests, and you may even remember them as being bad, awkward, slow, or unmaintainable. (Most people do.) On the other hand, you may have had a great first experience with unit tests, and you're reading this to see what more you might be missing.

This chapter will first analyze the "classic" definition of a unit test and compare it to the concept of integration testing. This distinction is confusing to many. Then we'll look at the pros and cons of each approach and develop a better definition of a "good" unit test. We'll finish up with a look at test-driven development, because it's often associated with unit testing. Throughout the chapter, we'll also touch on various concepts that are explained more thoroughly elsewhere in the book.

Let's begin by defining what a unit test should be.

1.1 Unit testing—the classic definition

Unit testing isn't a new concept in software development. It's been floating around since the early days of the Smalltalk programming language in the 1970s, and it proves itself time and time again as one of the best ways a developer can improve code quality while gaining a deeper understanding of the functional requirements of a class or method.

Kent Beck introduced the concept of unit testing in Smalltalk, and it has carried on into many other programming languages, making unit testing an extremely useful practice in software programming. Before we get too far, we need to define unit testing better. Here's the classic definition, from Wikipedia.

DEFINITION A *unit test* is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed. A "unit" is a method or function.

Unit testing will be performed against a system under test (SUT).

DEFINITION *SUT* stands for *system under test*, and some people like to use *CUT* (*class under test* or *code under test*). When we test something, we refer to the thing we're testing as the SUT.

This classic definition of a unit test, although technically correct, is hardly enough to enable us to better ourselves as developers. Chances are you already know this and are getting bored reading this definition again, because it appears in any book or website that discusses unit testing. Don't worry. In this book, we'll go beyond the classic definition of unit testing by addressing maintainability, readability, correctness, and more. But this familiar definition, precisely because it is familiar, gives us a shared base from which to extend the idea of a unit test.

No matter what programming language you're using, one of the most difficult aspects of defining a unit test is defining what's meant by a "good" one.

1.1.1 The importance of writing "good" unit tests

Most people who try to unit-test their code either give up at some point or don't actually perform unit tests. Instead, they either rely on system and integration tests to be performed much later in the product lifecycle or they resort to manually testing the code via custom test applications or by using the end product they're developing to invoke their code.

There's no point in writing a bad unit test, unless you're learning how to write a good one and these are your first steps into this field. If you're going to write a unit test badly without realizing it, you may as well not write it at all, and save yourself the trouble it will cause down the road with maintainability and time schedules. By defining what a good unit test is, we can make sure we don't start off with the wrong notion of what we're trying to write.

To succeed in this delicate art of unit testing, it's essential that you not only have a *technical definition* of what unit tests are, but that you describe the *properties* of a good unit test. To understand what a good unit test is, we need to look at what developers do when they're testing something.

How do you make sure that the code works today?

1.1.2 We've all written unit tests (sort of)

You may be surprised to learn this, but you've already implemented some types of unit testing on your own. Have you ever met a developer who has not tested his code before handing it over? Well, neither have I.

You might have used a console application that called the various methods of a class or component, or perhaps some specially created WinForm or WebForm UI (user interface) that checked the functionality of that class or component, or maybe even manual tests run by performing various actions within the real application's UI. The end result is that you've made certain, to a degree, that the code works well enough to pass it on to someone else.

Figure 1.1 shows how most developers test their code. The UI may change, but the pattern is usually the same: using a manual external tool to check something repeatedly, or running the application in full and checking its behavior manually.

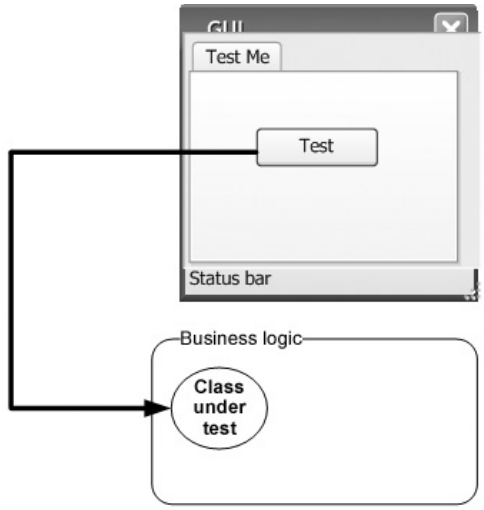

Figure 1.1 In classic testing, developers use a GUI (graphical user interface) to trigger an action on the class they want to test. Then they check the results.

These tests may have been useful, and they may come close to the classic definition of a unit test, but they're far from how we'll define a *good unit test* in this book. That brings us to the first and most important question a developer has to face when defining the qualities of a good unit test: what is a unit test, and what is not.

1.2 Properties of a good unit test

A unit test *should* have the following properties:

- It should be automated and repeatable.
- It should be easy to implement.
- Once it's written, it should remain for future use.
- Anyone should be able to run it.
- It should run at the push of a button.
- It should run quickly.

Many people confuse the act of testing their software with the concept of a unit test. To start off, ask yourself the following questions about the tests you've written up to now:

- Can I run and get results from a unit test I wrote two weeks or months or years ago?

- Can any member of my team run and get the results from unit tests I wrote two months ago?
- Can I run all the unit tests I've written in no more than a few minutes?
- Can I run all the unit tests I've written at the push of a button?
- Can I write a basic unit test in no more than a few minutes?

If you've answered "no" to any of these questions, there's a high probability that what you're implementing isn't a unit test. It's definitely *some* kind of test, and it's as important as a unit test, but it has drawbacks compared to tests that would let you answer "yes" to all of those questions.

"What was I doing until now?" you might ask. You've done *integration testing.*

1.3 Integration tests

What happens when your car breaks down? How do you learn what the problem is, let alone fix it? An engine consists of many parts working together, each relying on the others to help produce the final result: a moving car. If the car stops moving, the fault could be with any of these parts, or more than one. It's the integration of those parts that makes the car move. You could think of the car's movement as the ultimate integration test of these parts. If the test fails, all the parts fail together; if it succeeds, the parts all succeed.

The same thing happens in software. The way most developers test their functionality is through the final functionality of the user interface. Clicking some button triggers a series of events—various classes and components working together to produce the final result. If the test fails, all of these software components fail as a team, and it can be difficult to figure out what caused the failure of the overall operation. (See figure 1.2.)

As defined in *The Complete Guide to Software Testing*, by Bill Hetzel, integration testing is "an orderly progression of testing in which software and/or hardware elements *are combined and tested* until the entire system has been integrated." That definition of integration testing falls

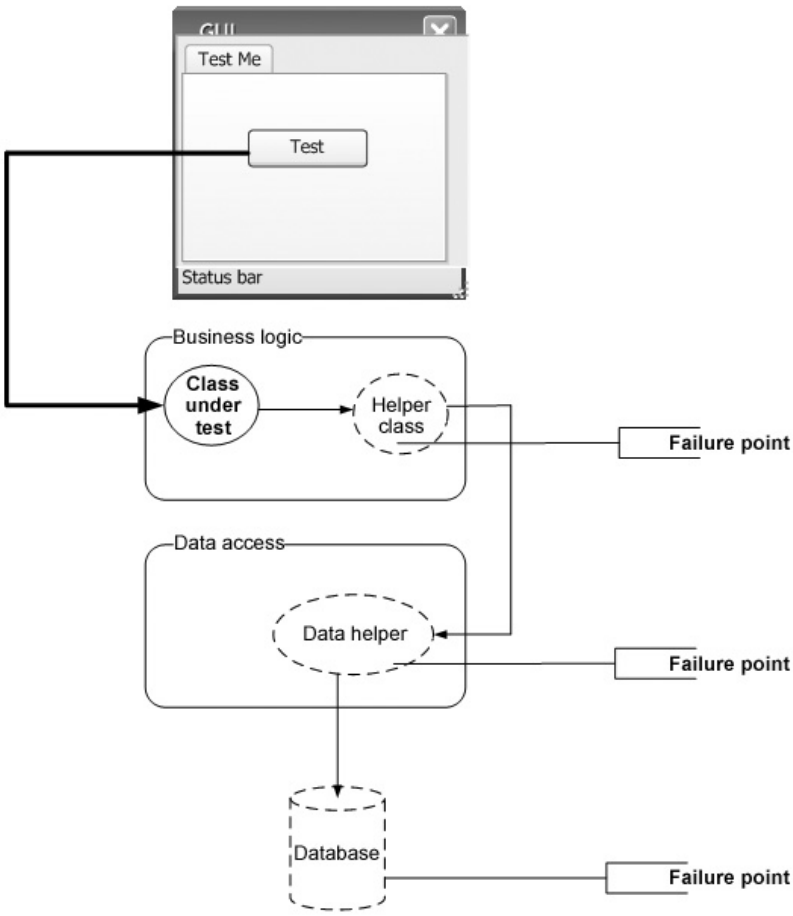

Figure 1.2 You can have many failure points in an integration test. All the units have to work together, and each of them could malfunction, making it harder to find the source of the bug.

a bit short of what many people do all the time, not as part of a system integration test, but as part of development and unit tests.

Here's a better definition of integration testing.

DEFINITION *Integration testing* means testing two or more dependent software modules as a group.

To summarize: an integration test exercises many units of code that work together to evaluate one or more expected results from the soft-

ware, whereas a unit test usually exercises and tests only a single unit in isolation.

The questions from the beginning of section 1.2 can help you recognize some of the drawbacks of integration testing. Let's look at them and try to define the qualities we're looking for in a good unit test.

1.3.1 Drawbacks of integration tests compared to automated unit tests

Let's apply the questions from section 1.2 to integration tests, and consider what we want to achieve with real-world unit tests:

- Can I run and get results from the test I wrote two weeks or months or years ago?

  If you can't do that, how would you know whether you broke a feature that you created two weeks ago? Code changes regularly during the life of an application, and if you can't (or won't) run tests for all the previous working features after changing your code, you just might break it without knowing. I call this "accidental bugging," and it seems to occur a lot near the end of a software project, when developers are under pressure to fix existing bugs. Sometimes they introduce new bugs inadvertently as they solve the old ones. Wouldn't it be great to know that you broke something within three minutes of breaking it? We'll see how that can be done later in this book.

  Good tests should be easily executed in their original form, not manually.

DEFINITION A *regression* is a feature that used to work and now doesn't.

- Can any member of my team run and get the results from tests I wrote two months ago?

  This goes with the last point but takes it up a notch. You want to make sure that you don't break someone else's code when you change something. Many developers fear changing *legacy code* in older systems for fear of not knowing what other code depends on what they're changing. In essence, they risk changing the system into an unknown state of stability.

Few things are scarier than not knowing whether the application still works, especially when you didn't write that code. If you knew you weren't breaking anything, you'd be much less afraid of taking on code you're less familiar with because you have that safety net of unit tests.

Good tests can be accessed and run by anyone.

DEFINITION *Legacy code* is defined by Wikipedia as "source code that relates to a no-longer supported or manufactured operating system or other computer technology," but many shops refer to any older version of the application currently under maintenance as legacy code. It often refers to code that's hard to work with, hard to test, and usually even hard to read.

A client of mine once defined legacy code in a down-to-earth way: "code that works." Many people like to define legacy code as "code that has no tests." The book *Working Effectively with Legacy Code* by Michael Feathers uses this as an official definition of legacy code, and it's a definition to be considered while reading this book.

- Can I run all the tests in no more than a few minutes?

  If you can't run your tests quickly (seconds are better than minutes), you'll run them less often (daily, or even weekly or monthly in some places). The problem is that, when you change code, you want to get feedback as early as possible to see if you broke something. The more time between running the tests, the more changes you make to the system, and the (many) more places to search for bugs when you find that you broke something.

  Good tests should run *quickly*.

- Can I run all the tests at the push of a button?

  If you can't, it probably means that you have to configure the machine on which the tests will run so that they run correctly (setting connection strings to the database, for example), or that your unit tests aren't fully automated. If you can't fully automate your unit tests, you'll probably avoid running them repeatedly, as will everyone else on your team.

  No one likes to get bogged down with configuring details to run tests just to make sure that the system still works. As developers, we have more important things to do, like writing more features into the system.

Good tests should be easily executed in their original form, not manually.

- Can I write a basic test in no more than a few minutes?

  One of the easiest ways to spot an integration test is that it takes time to prepare correctly and to implement, not just to execute. It takes time to figure out how to write it because of all the internal and sometimes external dependencies. (A database may be considered an external dependency.) If you're not automating the test, dependencies are less of a problem, but you're losing all the benefits of an automated test. The harder it is to write a test, the less likely you are to write more tests, or to focus on anything other than the "big stuff" that you're worried about. One of the strengths of unit tests is that they tend to test every little thing that might break, not only the big stuff. People are often surprised at how many bugs they can find in code they thought was simple and bug free.

  When you concentrate only on the big tests, the logic *coverage* that your tests have is smaller. Many parts of the core logic in the code aren't tested (even though you may be covering more components), and there may be many bugs that you haven't considered.

  Good tests against the system should be easy and quick to write.

From what we've seen so far about what a unit test is not, and what features need to be present for testing to be useful, we can now start to answer the primary question this chapter poses: what is a good unit test?

1.4 Good unit test—a definition

Now that we've covered the important properties that a unit test should have, let's define unit tests once and for all.

DEFINITION A *unit test* is an automated piece of code that invokes the method or class being tested and then checks some assumptions about the logical behavior of that method or class. A unit test is almost always written using a unit-testing framework. It can be written easily and runs quickly. It's fully automated, trustworthy, readable, and maintainable.

This definition sure looks like a tall order, particularly considering how many developers I've seen implementing unit tests poorly. It makes us take a hard look at the way we, as developers, have implemented testing up until now, compared to how we'd like to implement it. ("Trustworthy, readable, and maintainable" tests are discussed in depth in chapter 7.)

DEFINITION *Logical code* is any piece of code that has some sort of logic in it, small as it may be. It's logical code if it has one or more of the following: an IF statement, a loop, switch or case statements, calculations, or any other type of decision-making code.

Properties (getters/setters in Java) are good examples of code that usually doesn't contain any logic, and so doesn't require testing. But watch out: once you add any check inside the property, you'll want to make sure that logic is being tested.

In the next section, we'll take a look at a simple unit test done entirely in code, without using any unit-testing framework. (We'll look at unit-testing frameworks in chapter 2.)

1.5 A simple unit test example

It's possible to write an automated unit test without using a test framework. In fact, as they have gotten more into the habit of automating their testing, I've seen plenty of developers doing this before discovering test frameworks. In this section, I'll show what writing such a test without a framework can look like, so that you can contrast this with using a framework in chapter 2.

Assume we have a `SimpleParser` class (shown in listing 1.1) that we'd like to test. It has a method named `ParseAndSum` that takes in a string of 0 or more comma-separated numbers. If there are no numbers, it returns 0. If there's a single number, it returns that number as an `int`. If there are multiple numbers, it adds them all up and returns the sum (although, right now, the code can only handle 0 or 1 number).

Listing 1.1 A simple parser class to test

```
public class SimpleParser
    {
        public int ParseAndSum(string numbers)
        {
            if(numbers.Length==0)
            {
                return 0;
            }
            if(!numbers.Contains(","))
            {
                return int.Parse(numbers);
            }
            else
            {
                throw new InvalidOperationException(
"I can only handle 0 or 1 numbers for now!");
            }
        }
   }
```

We can create a simple console application project that has a reference to the assembly containing this class, and we can write a `SimpleParserTests` method as shown in listing 1.2. The test method invokes the *production class* (the class to be tested) and then checks the returned value. If it's not what's expected, it writes to the console. It also catches any exception and writes it to the console.

Listing 1.2 A simple coded method that tests the `SimpleParser` class

```
class SimpleParserTests
    {
        public static void TestReturnsZeroWhenEmptyString()
        {
            try
            {
                SimpleParser p = new SimpleParser();
                int result = p.ParseAndSum(string.Empty);
                if(result!=0)
                {
```

```
Console.WriteLine(
@"***SimpleParserTests.TestReturnsZeroWhenEmptyString:
_______
Parse and sum should have returned 0 on an empty string");
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e);
            }
        }
    }
```

Next, we can invoke the tests we've written by using a simple `Main` method run inside a console application in this project, as seen in listing 1.3. The `Main` method is used here as a simple test runner, which invokes the tests one by one, letting them write out to the console. Because it's an executable, this can be run without human intervention (assuming the tests don't pop up any interactive user dialogs).

Listing 1.3 Running *coded tests via a simple console application*

```
public static void Main(string[] args)
        {
            try
            {
                SimpleParserTests.TestReturnsZeroWhenEmptyString();
            }
            catch (Exception e)
            {
                Console.WriteLine(e);
            }
        }
```

It's the test method's responsibility to catch any exceptions that occur and write them to the console, so that they don't interfere with the running of subsequent methods. We can then add more method calls into the `Main` method as we add more and more tests to the project. Each test is responsible for writing the problem output (if there's a problem) to the console screen.

Obviously, this is an ad hoc way of writing such a test. If you were writing multiple tests like this, you might want to have a generic `ShowProblem` method that all tests could use, which would format the errors consistently. You could also add special helper methods that would help check on various things like null objects, empty strings, and so on, so that you don't need to write the same long lines of code in many tests.

Listing 1.4 shows what this test would look like with a slightly more generic `ShowProblem` method.

Listing 1.4 Using a more generic implementation of the `ShowProblem` method

```
public class TestUtil
    {
        public static void ShowProblem(string test,string message )
        {
            string msg = string.Format(@"
---{0}---
       {1}
--------------------
", test, message);
            Console.WriteLine(msg);
        }
    }

public static void TestReturnsZeroWhenEmptyString()
        {
            //use .NET's reflection API to get the current
                                                    method's name
            // it's possible to hard code this,
            //but it's a useful technique to know
            string testName = MethodBase.GetCurrentMethod().Name;
            try
            {
                SimpleParser p = new SimpleParser();
                int result = p.ParseAndSum(string.Empty);
                if(result!=0)
                {
            //Calling the helper method
                    TestUtil.ShowProblem(testName,
"Parse and sum should have returned 0 on an empty string");
```

```
            }
        }
        catch (Exception e)
        {
            TestUtil.ShowProblem(testName, e.ToString());
        }
    }
```

Unit-testing frameworks can help make helper methods more generic like this, so tests are written more easily. We'll talk about that in chapter 2. But before we get there, I'd like to discuss one important matter: not just *how* you write a unit test, but *when* during the development process you write it. That's where test-driven development comes into play.

1.6 Test-driven development

Once we know how to write structured, maintainable, and solid tests with a unit-testing framework, the next question is when to write the tests. Many people feel that the best time to write unit tests for software is after the software has been written, but a growing number of people prefer writing unit tests *before* the production code is written. This approach is called test-first or test-driven development (TDD).

NOTE There are many different views on exactly what test-driven development means. Some say it's test-first development, and some say it means you have a lot of tests. Some say it's a way of designing, and others feel it could be a way to drive your code's behavior with only some design. For a more complete look at the different views people have of TDD, see "The various meanings of TDD" on my blog (http://weblogs.asp.net/rosherove/archive/2007/10/08/the-various-meanings-of-tdd.aspx). In this book, TDD means test-first development, with design taking a secondary role in the technique (which isn't discussed in this book).

Figures 1.3 and 1.4 show the differences between traditional coding and test-driven development.

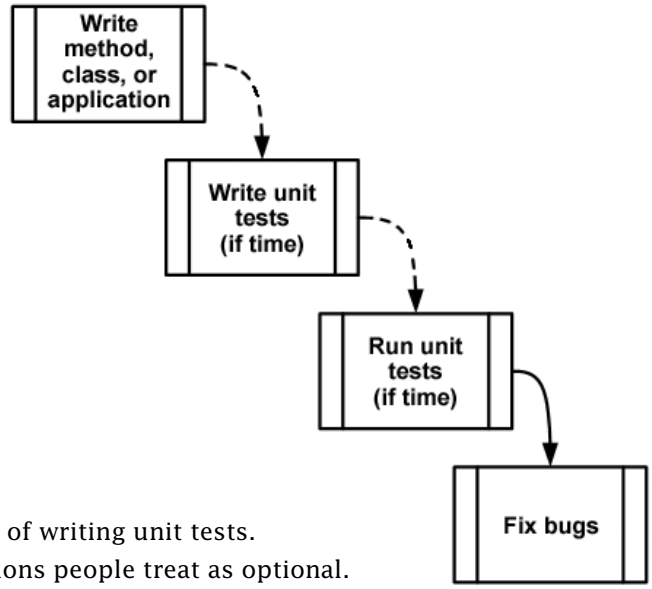

Figure 1.3 The traditional way of writing unit tests. The dotted lines represent actions people treat as optional.

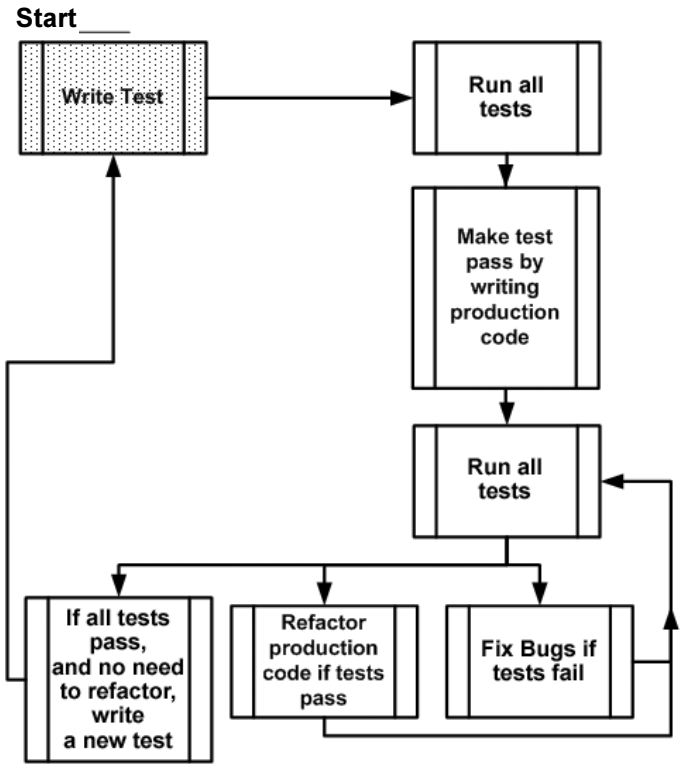

Figure 1.4 Test-driven development—a bird's-eye view. Notice the spiral nature of the process: write test, write code, refactor, write next test. It shows the incremental nature of TDD: small steps lead to a quality end result.

Test-driven development is different from traditional development, as figure 1.4 shows. You begin by writing a test that fails; then you move on to creating the production code, seeing the test pass, and continuing on to either refactor your code or to create another failing test.

This book focuses on the technique of writing good unit tests, rather than on test-driven development, but I'm a big fan of doing test-driven development. I've written several major applications and frameworks using TDD, have managed teams that utilize it, and have taught more than a hundred courses and workshops on TDD and unit-testing techniques. Throughout my career, I've found TDD to be helpful in creating quality code, quality tests, and better designs for the code I was writing. I am convinced that it can work to your benefit, but it's not without a price (time to learn, time to implement, and more). It's definitely worth the admission price, though.

It's important to realize that TDD doesn't ensure project success or tests that are robust or maintainable. It's quite easy to get caught up in the technique of TDD and not pay attention to the way unit tests are written: their naming, how maintainable or readable they are, and whether they test the right things or might have bugs. That's why I'm writing this book.

The technique of test-driven development is quite simple:

1. *Write a failing test to prove code or functionality is missing from the end product.*

   The test is written *as if* the production code were already working, so the test failing means there's a bug in the production code. For example, if I wanted to add a new feature to a calculator class that remembers the `LastSum` value, I would write a test that verifies that `LastSum` is indeed a number. The test will fail because we haven't implemented that functionality yet.

2. *Make the test pass by writing production code that meets the expectations of your test.*

   It should be written as simply as possible.

3. *Refactor your code.*

   When the test passes, you're free to move on to the next unit test or to *refactor* your code to make it more readable, to remove code duplication, and so on.

Refactoring can be done after writing several tests or after writing each test. It's an important practice, because it ensures your code gets easier to read and maintain, while still passing all of the previously written tests.

DEFINITION *Refactoring* means changing a piece of code without changing its functionality. If you've ever renamed a method, you've done refactoring. If you've ever split a large method into multiple smaller method calls, you've refactored your code. The code still does the same thing, but it becomes easier to maintain, read, debug, and change.

The preceding steps sound technical, but there's a lot of wisdom behind them. Done correctly, TDD can make your code quality soar, decrease the number of bugs, raise your confidence in the code, shorten the time it takes to find bugs, improve your code's design, and keep your manager happier. If TDD is done incorrectly, it can cause your project schedule to slip, waste your time, lower your motivation, and lower your code quality. It's a double-edged sword, and many people find this out the hard way.

1.7 Summary

In this chapter, we defined a good unit test as one that has these qualities:

- It's an automated piece of code that invokes a different method and then checks some assumptions on the logical behavior of that method or class.
- It's written using a unit-testing framework.
- It can be written easily.
- It runs quickly.
- It can be executed repeatedly by anyone on the development team.

To understand what a *unit* is, we had to figure out what sort of testing we've done until now. We identified that type of testing as integration testing because it tests a set of units that depend on each other.

The difference between unit tests and integration tests is important to recognize. You'll be using that knowledge in your day-to-day life as a

developer when deciding where to place your tests, what kind of tests to write when, and which option is better for a specific problem. It will also help you identify how to fix problems with tests that are already causing you headaches.

We also looked at the cons of doing integration testing without a framework behind it: this kind of testing is hard to write and automate, slow to run, and needs configuration. Although you do want to have integration tests in a project, unit tests can provide a lot of value earlier in the process, when bugs are smaller and easier to find, and there's less code to skim through.

Lastly, we talked about test-driven development, how it's different from traditional coding, and what its basic benefits are. TDD helps you make sure that the code coverage of your test code (how much of the code your tests exercise) is very high (close to 100 percent of *logical* code). It helps you make sure that your tests can be trusted by making sure that they fail when the production code isn't there, and that they pass when the production code works. TDD also has many other benefits, such as aiding in design, reducing complexity, and helping you tackle hard problems step by step. But you can't do TDD without knowing how to write good tests.

If you write tests *after* writing the code, you assume the test is OK because it passes, when it could be that you have bugs in your tests. Trust me—finding bugs in your *tests* is one of the most frustrating things you can imagine. It's important that you don't let your tests get to that state, and TDD is one of the best ways I know to keep that possibility close to zero.

In the next chapter, we'll start writing our first unit tests using NUnit, the de facto unit-testing framework for .NET developers.

.NET PROGRAMMING

the art of UNIT TESTING

Roy Osherove
Foreword by Michael Feathers

Unit testing, done right, can mean the difference between a failed project and a successful one, between a maintainable code base and a code base that no one dares touch, and between getting home at 2 AM or getting home in time for dinner, even before a release deadline.

The **Art of Unit Testing** builds on top of what's already been written about this important topic. It guides you step by step from simple tests to tests that are maintainable, readable, and trustworthy. It covers advanced subjects like mocks, stubs, and frameworks such as Typemock Isolator and Rhino Mocks.

And you'll learn about advanced test patterns and organization, working with legacy code and even untestable code. The book discusses tools you need when testing databases and other technologies. It's written for .NET developers but others will also benefit from this book.

What's Inside

- How to create readable, maintainable, trustworthy tests
- Stubs, mock objects, and automated frameworks
- Working with .NET tools, including NUnit, Rhino Mocks and Typemock Isolator

The chief architect at Typemock, **Roy Osherove** is one of the original ALT.NET organizers. He consults and trains teams worldwide on the gentle art of unit testing and test-driven development. He frequently speaks at international conferences such as TechEd and JAOO. Roy's blog is at ISerializable.com.

For online access to the author, code samples, and a free ebook for owners of this book, go to www.manning.com/TheArtofUnitTesting

Free ebook
SEE INSERT

"An important book that should have been written years ago."
—From the Foreword by Michael Feathers, Object Mentor

"Every book on unit testing ever written has been for amateurs. This is one is for professionals."
—Josh Cronemeyer
ThoughtWorks

"Serious about software craftsmanship? This book is a must-have for everyone who is."
—Dave Nicolette
Independent Agile Coach

"State of the art!"
—Gabor Paller, OnRelay Ltd

ISBN 13: 978-1-933988-27-6
ISBN 10: 1-933988-27-4
53999
9 781933 988276

MANNING $39.99 / Can $49.99 [INCLUDING eBOOK]