1 The basics of unit testing

This chapter covers

- Defining a unit test
- Contrasting unit testing with integration testing
- Exploring a simple unit testing example
- Understanding test-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 a few tests, and you may even remember them as being bad, awkward, slow, or unmaintainable. (Most people do.) On a more upbeat note, 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 unit testing versus integration testing and develop a better definition of a "good" unit test. We'll finish with a look at test-driven development, because it's often associated with unit testing. Throughout

the chapter, I'll also touch on concepts that are explained more thoroughly elsewhere in the book.

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

1.1 Defining unit testing, step by step

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 I go any further, I need to define unit testing better. Here's the classic definition, from Wikipedia. It'll be slowly evolving throughout this chapter, with the final definition appearing in section 1.4.

> **DEFINITION 1.0** 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.

The thing you'll write tests for is called the 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 you test something, you refer to the thing you're testing as the SUT.

I used to feel (Yes, feel. There is no science in this book. Just art.) this definition of a unit test was technically correct, but over the past couple of years, my idea of what a *unit* is has changed. To me, a *unit* stands for "unit of work" or a "use case" inside the system.

Definition

A *unit of work* is the sum of actions that take place between the invocation of a public method in the system and a single noticeable end result by a test of that system. A noticeable end result can be observed without looking at the internal state of the system and only through its public APIs and behavior. An end result is any of the following:

- The invoked public method returns a value (a function that's not void).
- There's a noticeable change to the state or behavior of the system before and after invocation that can be determined without interrogating private state. (Examples: the system can log in a previously nonexistent user, or the system's properties change if the system is a state machine.)
- There's a callout to a third-party system over which the test has no control, and that third-party system doesn't return any value, or any return value from that system is ignored. (Example: calling a third-party logging system that was not written by you and you don't have the source to.)

This idea of a unit of work means, to me, that a *unit* can span as little as a single method and up to multiple classes and functions to achieve its purpose.

You might feel that you'd like to minimize the size of a unit of work being tested. I used to feel that way. But I don't anymore. I believe if you can create a unit of work that's larger, and where its end result is more noticeable to an end user of the API, you're creating tests that are more maintainable. If you try to minimize the size of a unit of work, you end up faking things down the line that aren't really end results to the user of a public API but instead are just *train stops* on the way to the *main station*. I explain more on this in the topic of overspecification later in this book (mostly in chapter 8).

> **UPDATED DEFINITION 1.1** A *unit test* is a piece of code that invokes a unit of work and checks one specific end result of that unit of work. If the assumptions on the end result turn out to be wrong, the unit test has failed. A unit test's scope can span as little as a method or as much as multiple classes.

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

Being able to understand what a unit of work is isn't enough.

Most people who try to unit test their code either give up at some point or don't actually perform unit tests. Instead, either they 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 in 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, you can make sure you don't start off with the wrong notion of what your objective is.

To understand what a good unit test is, you 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 their 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 WinForms or Web Forms UI that checked the functionality of that class or component, or maybe even manual tests

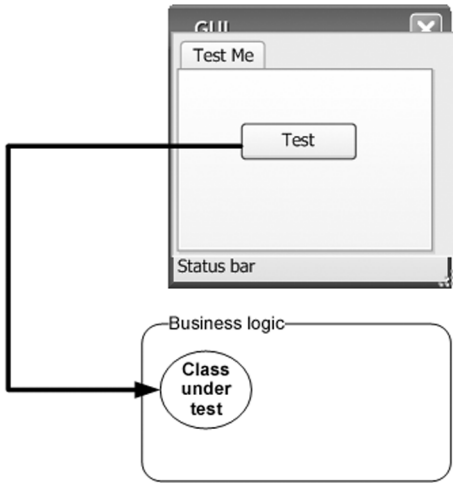

Figure 1.1 In classic testing, developers use a graphical user interface (GUI) to trigger an action on the class they want to test. Then they check the results.

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.

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 I'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.
- It should be relevant tomorrow.
- Anyone should be able to run it at the push of a button.
- It should run quickly.
- It should be consistent in its results (it always returns the same result if you don't change anything between runs).
- It should have full control of the unit under test.
- It should be fully isolated (runs independently of other tests).
- When it fails, it should be easy to detect what was expected and determine how to pinpoint the problem.

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 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 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 been doing integration testing.

1.3 Integration tests

I consider integration tests as any tests that aren't fast and consistent and that use one or more real dependencies of the units under test. For example, if the test uses the real system time, the real filesystem, or a real database, it has stepped into the realm of integration testing.

If a test doesn't have control of the system time, for example, and it uses the current DateTime.Now in the test code, then every time the test executes, it's essentially a different test because it uses a different time. It's no longer consistent.

That's not a bad thing per se. I think integration tests are important counterparts to unit tests, but they should be separated from them to achieve a feeling of "safe green zone," which is discussed later in this book.

If a test uses the real database, then it's no longer only running in memory, in that its actions are harder to erase than when using only in-memory fake data. The test will also run longer, again a reality that it has no control over. Unit tests should be fast. Integration tests are usually much slower. When you start having hundreds of tests, every half-second counts.

Integration tests increase the risk of another problem: testing too many things at once.

What happens when your car breaks down? How do you learn what the problem is, let alone fix it? An engine consists of many subsystems 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 subsystems—or more than one. It's the integration of those subsystems (or layers) that makes the car move. You could think of the car's movement as the ultimate integration test of these parts as the car goes down the road. If the test fails, all the parts fail together; if it succeeds, all the parts succeed.

The same thing happens in software. The way most developers test their functionality is through the final functionality of the UI. Clicking some button triggers a series of events—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 (Wiley, 1993), 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 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* is testing a unit of work without having full control over all of it and using one or more of its real dependencies, such as time, network, database, threads, random number generators, and so on.

To summarize: an integration test uses real dependencies; unit tests isolate the unit of work from its dependencies so that they're easily consistent in their results and can easily control and simulate any aspect of the unit's behavior.

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

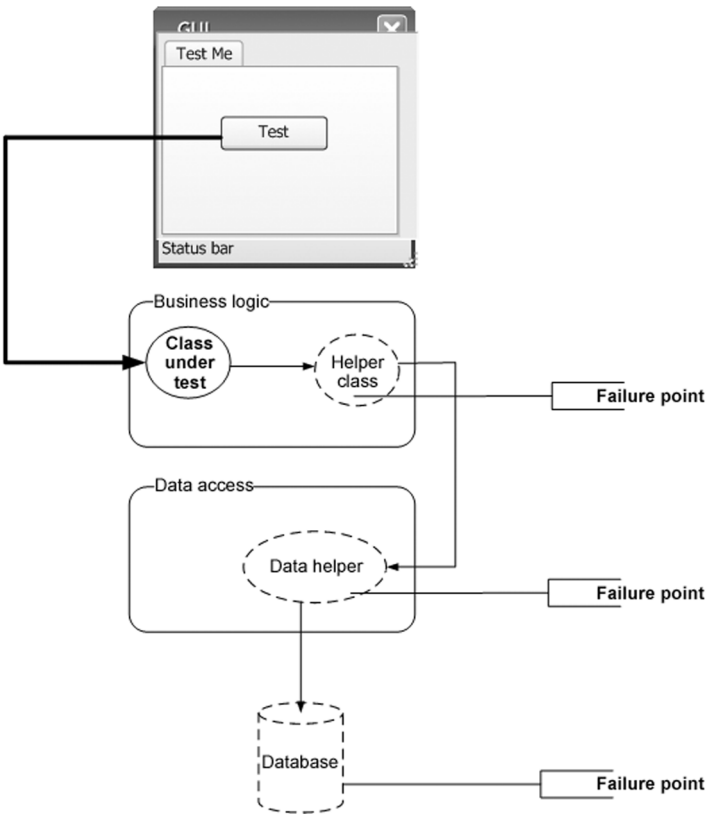

Figure 1.2 You can have many failure points in an integration test. All the units have to work together, and each could malfunction, making it harder to find the source of the bug.

1.3.1 Drawbacks of nonautomated integration tests compared to automated unit tests

Let's apply the questions from section 1.2 to integration tests and consider what you 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, how would you know whether you broke a feature that you created earlier? Code changes regularly during the life of an application, and if you can't (or won't) run tests for all the previously 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 resolve the old ones. Wouldn't it be great to know that you broke something within three minutes of breaking it? You'll see how that can be done later in this book.

DEFINITION A *regression* is one or more units of work that once worked and now don't.

- *Can any member of my team run and get results from tests I wrote two months ago?*
  This goes with the previous 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 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." *Working Effectively with Legacy Code* by Michael Feathers (Prentice Hall, 2004) 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 I've written 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 I've written 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. Developers 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, once you've figured out the patterns you want to use to test your specific object model. Small warning: even experienced unit testers can find that it may take 30 minutes or more to figure out how to write the very first unit test against an object model they've never unit tested before. This is part of the work, and is expected. The second and subsequent tests on that object model should be very easy to accomplish.

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

1.4 What makes unit tests good

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

> **UPDATED AND FINAL DEFINITION 1.2** A *unit test* is an automated piece of code that invokes the unit of work being tested, and then checks some assumptions about a single end result of that unit. A unit test is almost always written using a unit testing framework. It can be written easily and runs quickly. It's trustworthy, readable, and maintainable. It's consistent in its results as long as production code hasn't changed.

This definition certainly looks like a tall order, particularly considering how many developers implement 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 8.)

In the previous edition of this book, my definition of a unit test was slightly different. I used to define a unit test as "only running against control flow code." But I no longer think that's true. Code without logic is usually used as part of a unit of work. Even properties with no logic will get used by a unit of work, so they don't have to be specifically targeted by tests.

> **DEFINITION** *Control flow code* is any piece of code that has some sort of logic in it, small as it may be. It has one or more of the following: an `if` statement, a `loop`, `switch`, or `case` statement, 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 specific targeting by the tests. It's code that will probably get used by the unit of work you're testing, but there's no need to test it directly. But watch out: once you add any check inside a property, you'll want to make sure that logic is being tested.

In the next section, we'll look at a simple unit test done entirely with 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, because developers have gotten more into the habit of automating their testing, I've seen plenty of them 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 you have a `SimpleParser` class (shown in listing 1.1) that you'd like to test. It has a method named `ParseAndSum` that takes in a string of zero 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 zero or one number). Yes, I know the `else` part isn't needed, but just because ReSharper tells you to jump off a bridge, doesn't mean you have to do it. I think the `else` adds a nice readability to it.

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!");
        }
    }
}
```

You can create a simple console application project that has a reference to the assembly containing this class, and you can write a `SimpleParserTests` method as shown in the following listing. 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, the test method 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, you can invoke the tests you've written by using a simple `Main` method run inside a console application in this project, as shown in the next listing. 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. You can then add more method calls into the `Main` method as you 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 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.

The following listing 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 make helper methods more generic like this, so tests are written more easily. I'll talk about that in chapter 2. 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 you 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 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 views people have of TDD, see "The various meanings of TDD" on my blog (http://osherove.com/blog/2007/10/8/the-various-meanings-of-tdd.html). 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 TDD.

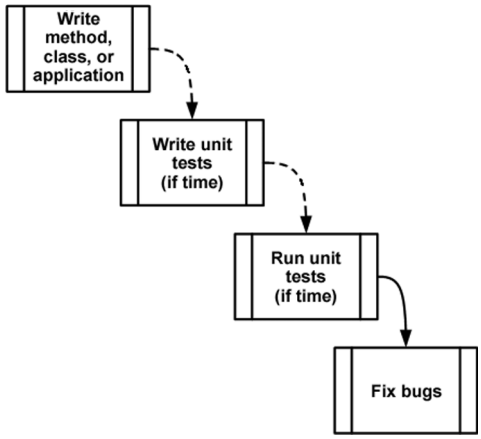

Figure 1.3 The traditional way of writing unit tests. The broken lines represent actions people treat as optional.

TDD 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 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 TDD. 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'm convinced that it can work to your

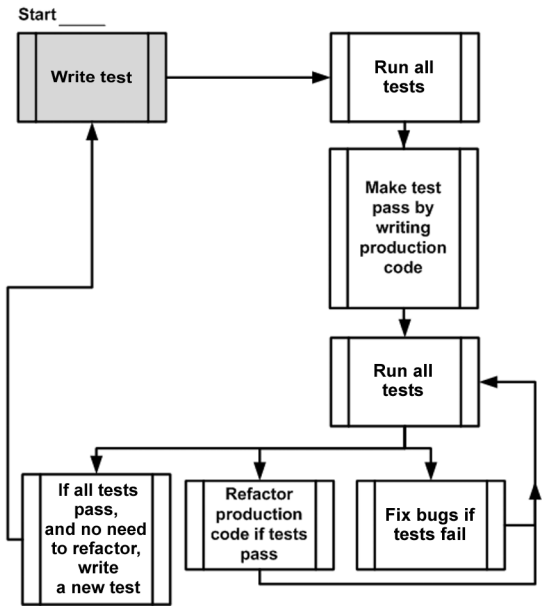

Figure 1.4 Test-driven development—a bird's-eye view. Notice the spiral nature of the process: write test, write code, refactor, write next test. It shows the incremental nature of TDD: small steps lead to a quality end result.

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 TDD 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. If I wanted to add a new feature to a calculator class that remembers the `LastSum` value, I'd write a test that verifies that `LastSum` is indeed the correct value. The test will fail to compile, and after adding only the needed code to make it compile (without the real functionality to remember the number), the test will now run, and fail, because I haven't implemented that functionality yet.
2. *Make the test pass by writing production code that meets the expectations of your test.* The production code should be kept as simple 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.

Technically, one of the biggest benefits of TDD nobody tells you about is that by seeing a test fail, and then seeing it pass without changing the test, you're basically testing the test itself. If you expect it to fail and it passes, you might have a bug in your test or you're testing the wrong thing. If the test failed and now you expect it to pass, and it still fails, your test could have a bug, or it's expecting the wrong thing to happen.

This book deals with readable, maintainable, and trustworthy tests, but the greatest affirmation you'll get from your tests comes when you see them fail and pass when they should. TDD helps with that a lot, and that's one of the reasons developers do far less debugging when TDD-ing their code than when they're simply unit testing it after the fact. If they trust the test, they don't feel a need to debug it "just in case." And that's the kind of trust you can only gain by seeing both sides of the test—failing and passing when it should.

1.7 The three core skills of successful TDD

To be successful in test-driven development you need three different skill sets: knowing how to write good tests, writing them test-first, and designing them well.

- *Just because you write your tests first doesn't mean they're maintainable, readable, or trustworthy.* Good unit testing skills are what the book you're currently reading is all about.
- *Just because you write readable, maintainable tests doesn't mean you get the same benefits as when writing them test-first.* Test-first skills are what most of the TDD books out there teach, without teaching the skills of good testing. I would especially recommend Kent Beck's *Test-Driven Development: by Example* (Addison-Wesley Professional, 2002).
- *Just because you write your tests first, and they're readable and maintainable, doesn't mean you'll end up with a well-designed system.* Design skills are what make your code beautiful and maintainable. I recommend *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce (Addison-Wesley Professional, 2009) and *Clean Code* by Robert C. Martin (Prentice Hall, 2008) as good books on the subject.

A pragmatic approach to learning TDD is to learn each of these three aspects separately; that is, to focus on one skill at a time, ignoring the others in the meantime. The reason I recommend this approach is that I often see people trying to learn all three skill sets at the same time, having a really hard time in the process, and finally giving up because the wall is too high to climb.

By taking a more incremental approach to learning this field, you relieve yourself of the constant fear that you're getting it wrong in a different area than you're currently focusing on.

In regard to the order of the learning approach, I don't have a specific scheme in mind. I'd love to hear from you about your experience and recommendations when learning these skills. You can find contact links at http://osherove.com.

1.8 Summary

In this chapter, I 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, you had to figure out what sort of testing you've done until now. You 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.

Last, we looked at 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). TDD helps you make sure that your tests can be trusted. TDD "tests your tests" in that it lets you see them fail and pass when they should. 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 successfully over time without knowing how to write good tests.

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