MEAP Edition
Manning Early Access Program

Copyright 2007 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

Part I Getting started

1. The Basics of unit testing
2. The first unit test

Part II Core techniques

3. Using Stubs to break dependencies
4. Interaction testing using Mock Objects
5. Mock Object frameworks

Part III The test code

6. Managing tests
7. Trustworthy tests
8. Maintainable tests
9. Readable tests

Part IV Design and process

10. Software methodologies and unit tests
11. Designing for testability
12. Test Driven Development – The ultimate weapon
13. Refactoring for unit testability

Part V Integration testing with unit test frameworks

14. Testing Web applications and other UIs
15. Testing database code
16. Testing legacy systems
17. Customer/acceptance tests

Part IV Appendices

Appendix A: Alternative .NET testing frameworks and extensions

Appendix B: Unit testing resources

1

The basics of unit testing

One of the biggest failed projects I worked on had unit tests. Or so I thought. I was leading a group of programmers to create a billing application, and we were doing it in a fully test-driven manner – writing the test, then writing the code, seeing the test fail, making the test pass, refactor, rinse, repeat.

The first few months of the project were great; things were looking up, and we had tests that proved that our code worked. As time went by, requirements changed, and we were forced to change our code to fit those new requirements. Whenever we changed the code, tests broke and we had to fix them – the code was still working, but the tests we wrote were so brittle that any little change in our code broke them, even though the code was working just fine. It became a daunting task to change our code in a class or a method for fear of changing all the unit tests involved with that unit being tested.

Worse yet, some tests became unusable because the people who wrote them had left the project and no one knew how to maintain the tests, or what they were testing. The names we gave our unit test methods were not clear enough. We had tests relying on other tests. We ended up throwing away most of the tests less than 6 months into the project.

It was a miserable failure because we let the tests we wrote do more harm than good – they were taking too much time to maintain and understand than they were saving us in the long run. So we stopped using them. I moved on to other projects, where we did a better job writing our unit tests, and even had some great successes using them, saving huge amounts of debugging and integration time.

Ever since that first project that failed, I've been compiling best practices for unit tests and using them on the next project. Every time I get involved in a new project, I find a few more best practices. A solid naming guideline is just one of those. Understanding how to write unit tests, and making them maintainable, readable and trust-worthy is what this book is about– no matter what language or Integrated Development Environment (IDE) you work with.

This book will cover the basics of writing a unit test, then move on to the basics of Interaction testing, and from then we'll move to best practices for writing, managing and maintaining unit tests in the real world.

1.1 Unit testing—classic definition

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

Kent Beck is the person who introduced the concept of unit testing in Smalltalk. The concept he created has carried on into many programming languages today, which has made unit testing an extremely useful practice in software programming. The concept of unit testing as introduced in Smalltalk has been carried on into many programming languages today.

Before we get too far, we need to define unit testing better. I will start with the classic definition most of us have heard a million times.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=355

UNIT TEST—THE CLASSIC 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.

The piece of code being tested is often called *SUT* (System Under Test). The piece of code that tests the SUT will usually reside in a *Test Method*. This classic definition, while technically correct, is hardly enough to get us started down a path where we can better ourselves as developers. Chances are you already know this and are getting bored even reading this definition again, as it appears practically in any web site or book that discusses unit testing.

Don't worry; in this book, I'll take you beyond the classic definition of unit testing by addressing issues not mentioned in it: Maintainability, Readability, Correctness and more. However, precisely *because* you probably might already be familiar with the classic definition, it gives us a shared knowledge base from which to extend the idea of a unit test into something with more value than is currently defined.

No matter what programming language you are using, one of the hardest aspects of defining a unit test is defining what is meant by a "good" one.

1.1.1 Defining a "good" unit test

I firmly believe that there's no point in writing a bad unit test. If you're going to write a unit test badly, you may as well not write it all and save yourself the trouble it will cause down the road with maintainability and time schedules. Defining what a good unit test is the first step we can do to make sure we don't start off with the wrong notion of what we're trying to write.

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 lifecycle of the product they are developing or resort to manually testing the code via custom test applications or actually using the end product they are developing to invoke their code.

To succeed, it is essential that you not only have a *technical* definition of a unit test, but that you describe *the properties of a good unit test*. To understand what a good unit test is, we'll need to go Look into the present to understand what developers have been doing so far in a software project.

In other words, if you hadn't been doing unit tests, how *did* you make sure that the code works?

1.1.2 We've all written unit tests

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 the code they wrote before letting it out from under their hands? Well, neither have I.

It might have been a console application that calls the various methods of a class or component, some specially created Winform or Webform UI that checks the functionality of that class or component, or even manual tests that are run by performing various actions within the real application's UI to test the end functionality. The end result is that the developer is certain to a degree that the code works well enough to give it away to someone else. Figure 1.1 shows how most developers test their code.

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=355

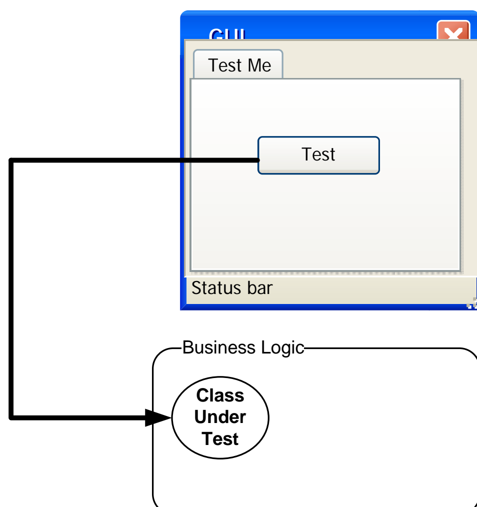

Figure 1.1 For classic testing, developers use a GUI (windows or web) and trigger an action on the class they want to test. Then they check the results somewhere (UI or other places).

Of course, the UI may change, but the pattern is usually the same: use a manual external tool to check something repeatedly.

While these may have been very useful, and, technically, they may be close to the "classic" definition of a unit test (although sometimes not even that, especially when they are done manually), they are far from the definition of a *unit test* as used in this book. That brings us to the first and most important question a developer has to face when attempting to define the qualities of a good unit test: what a unit test is and is not.

1.2 Good unit tests

We can map out what properties a unit test *should* have. After we write these out, we'll try to define what a unit test is:

- It is automated and repeatable.
- It is easy to implement.
- Once it's written, it stays on for the future.
- Anyone can run it.
- It runs at the push of a button.
- It runs quickly.

Many people confuse the act of simply 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 of a unit test I wrote two weeks/months/years ago?
- Can any member of my team run and get the results from unit tests I wrote two months ago?
- Can it take me no more than a few minutes to run all the unit tests I've written so far?
- 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 any of these questions with a "no," there's a high probability that what you're actually implementing is not really a unit test. It's *some* kind of test, absolutely, and it's *just* as important as a unit test, but it has enough drawbacks to consider writing tests that answer "yes" to all of these questions.

"So what *was* I doing until now?" you might ask. You've done *integration testing*. We'll touch on what that means in the next section.

1.3 Integration tests

What happens when your car breaks down? How do you know what the problem is, let alone how to fix it? Perhaps all you've heard is a weird rumbling sound before the car suddenly stopped moving, or some odd-colored exhaust fumes were being emitted – but where *is* the problem? An engine is made of many parts working together in partnership—each relying on the other to function properly and produce the final result: a moving car. If the car stops moving, the fault could be on any one of these parts, or more than one of them all at once. In effect, it is the integration of those parts that makes the car move – you could think of the car's eventual movement as the ultimate test of the integration of these parts.

If the test fails – all the parts fail together, and if it succeeds – they all succeed as well.

The same thing happens in software: The way most developers test their functionality is in the eventual final functionality through some sort of user interface. Clicking some button somewhere triggers a series of events – various classes and components working together, relying on each other to produce the final result – a working set of functionality. If suddenly the test fails – all of these software components fail as a team – it could get really hard finding out the true culprit of the failure of the final operation. See figure 1.2 for an example.

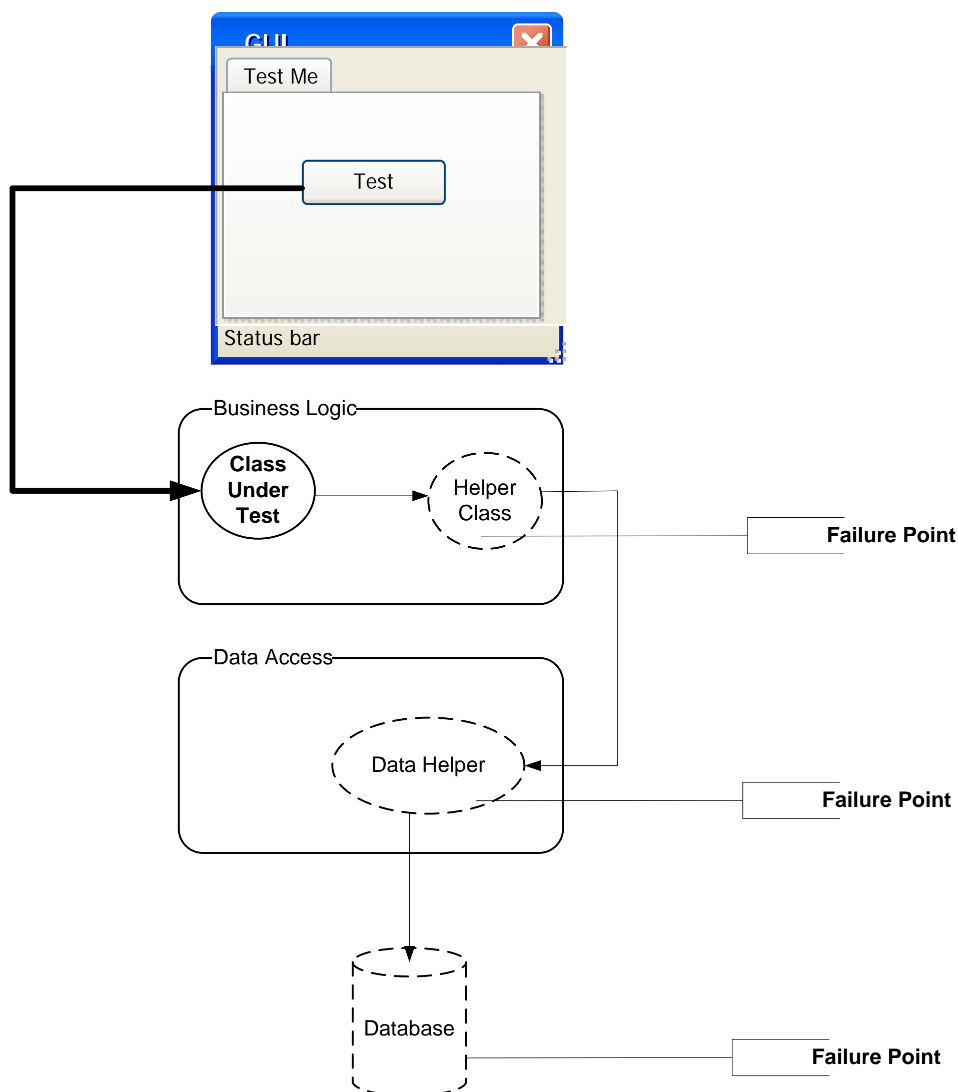

Figure 1.2 You can have many failure points in an integration test because all the units have to work together, and each of them could malfunction, making it harder to find the source of the bug. With unit tests, like the hero says in the movie *Highlander*, "There

can be only one" (culprit).

With previous description of an integration test in mind, let's look at the classic definition of Integration Tests. According to the book *The Complete Guide to Software Testing* 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 (in parallel).

INTEGRATION TESTING—A BETTER DEFINITION

Testing two or more dependent software modules as a group.

Integration testing is important enough a subject that you use it in conjunction with unit tests. That's why see part IV of this book for a deep review of integration testing techniques using the various unit testing frameworks (You'll learn more about unit testing frameworks in chapter 2 – "The First Unit Test".)

An integration test would exercise many units of code that work together to evaluate one or more results, while a unit test would usually exercise and test only a single unit in isolation.

The *questions* at the beginning of Section 1.2 can help you realize some of the drawbacks with integration testing are easily identifiable. We'll cover them next and try to define the good qualities we are looking for in a unit test based on the explanations to these questions.

1.3.1 Drawbacks of integration tests

Let's go over through those questions one more time, shall we? This time I'll list some things that we will want to achieve when implementing real world unit tests and why these questions are important for finding out whether they are achieved or not.

Can you run the same test in the future?

Question Can I run and get results of a unit test I wrote two weeks/months/years ago?

Problem If you can't do that, how would you know whether you broke a feature that you created two weeks ago (also called a "Regression")? Code changes all the time during the life of an application. When you can't (or won't) run the tests for all the previous working features after changing your code, you just might break it without knowing. I call it "accidental bugging." This "accidental bugging" seems to occur a lot near the end of a software project, when under time pressures, developers are fixing bugs and introducing new bugs inadvertently as they solve the old ones. Some places fondly call that stage in the project's lifetime "hell month."

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

Test Rule Tests should be easily executed in their original form, not manually.

Can other team members run the same test?

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

Problem 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 fix/change something. Many developers fear changing *legacy code* in older systems for fear of not knowing what dependencies other code has on what they are changing. In essence, they are changing the system into an unknown state of stability.

Few things are scarier than not knowing if the application still works, especially when you didn't write that code. But if you had the ability to make sure nothing broke, you'd be much less afraid of taking on code with which you are less familiar just because you had that safety net of unit tests to tell you whether you broke something anywhere in the system.

Test Rule Anyone can get and run the tests

We just introduced a new term in the last question, let's establish what *Legacy Code* means.

LEGACY CODE

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

A client of mine once defined legacy code in a very down-to-earth way, "Code that works." In many ways, although that does bring a smile to your face, you can't help but nod and say to yourself "yes, I know what he's talking about…". Many people like to define Legacy code as "code that has not tests", which is also a reasonable enough definition to be considered while reading this book.

Can you run all unit tests in minutes?

Question Does it take no more than a few minutes to run all the unit tests I've written so far?

Problem If you can't run your tests quickly (seconds is 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 longer you take between running the tests, the more changes you make to the system, and when you do find that you broke something, you'll have many more places to look for to find out where the bug resides.

Test Rule Tests should run *quickly.*

Can you run all unit tests at the push of a button?

Question Can I run all the unit tests I've written at the push of a button?

Problem If you can't, that 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, as an example), or that your unit tests are not *fully automated.* If you can't fully automate your unit tests, there's a slimmer chance that you'll avoid running them repeatedly, as well as the chance that anyone else on your team will run them.

No one likes to get bogged down with little configuration details to run tests when all they are trying to do is make sure that the system still works. As developers, we have more important things to do, like write more features into the system.

Test Rule Tests should be easily executed in their original form, not manually.

Can you write a basic test in a few minutes?

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

Problem One of the easiest ways to spot an integration test is that it takes time to prepare correctly and to implement, not only 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, that is less of a problem, but that just means 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 focus on anything else than just 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 just the big stuff – people are often surprised at just how many bugs they can find in code they considered to be totally simple and bug free.

When you concentrate only on the big tests, the *coverage* that your tests have is smaller – many parts of the core logic in the code are not tested, and you may find many bugs that you hadn't considered.

Test Rule Unit Tests against the system should be easy and quick to write

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

1.4 Unit test—final definition

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

DEFINITION: A GOOD UNIT TEST

A unit test is an automated piece of code that invokes a different method and then checks some assumptions about the *logical* behavior of that method or class under test.

A unit test is written using a *unit testing framework*. It can be written easily and runs quickly. It can be executed, repeatedly, by anyone on the development team.

That definition sounds more like something I'd like to see you implement after writing this book. But it sure looks like a tall order, especially based on how you've probably implemented unit tests so far. 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.

Here's what "Logical Code" means in the definition of a unit test:

DEFINITION: LOGICAL CODE

Logical code means any piece of code that has some sort of logic in it, small as it may be. It's logical code if the piece of code has one or more of the following:

- An IF statement
- A loop
- Swith, case
- any other type of decision making code

Properties (getters/setters in java) are a good example of code that usually does *not* contain any logic, and so does not require testing. But watch out, once you want to add any check inside the property, you'll want to make sure that logic is being tested for correctness.

In the next section we'll take a look at a simple example of a unit test done entirely in code, without using any Unit Test Framework (which you'll learn about in chapter 2).

1.5 A simple unit test example

Assume we have a SimpleParser class in our project that we'd like to test as shown in listing 1.1. It takes in a string of 0 or more numbers with a comma between them. If there are no numbers it returns zero. For a single number it returns that number as an int. For multiple numbers it sums them all up and returns the sum (right now it can only handle zero or one number though):

Listing 1.1 A simple parser class we'd like 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 add a simple console application project that has a reference to the assembly containing this class, and write a method like this in a class called SimpleParserTests, as showin in listing 1.2.

The test is simply a method, which invokes the production class (production: the actual product you're building and would like to test) and then checks the returned value. If it's not what is expected to be, 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 our SimpleParser class.

```
class SimpleParserTests
    {
        public static void TestReturnsZeroWhenEmptyString()
        {
            try
            {
                SimpleParser p = new SimpleParser();
                int result = p.ParseAndSum("1");
                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 simply invoke the tests we've written 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 for any problem. Since it's an executable, this can be run without human intervention (assuming no test pops up any interactive user dialogs).

Listing 1.3 Running our coded tests via a simple console application

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

The test also catches any exception that might occur and writes it to the console output.

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 other methods after them. We can then add more method calls into the main method as we add more and more tests into the test project. Each test is responsible writing the problem output (if there is a problem) to the console screen).

Obviously, this is a very simplistic way of doing it. You might want to have a generic "`ShowProblem`" method that all unit tests can use, which will format the errors the same way for everyone, and you could add special helper methods that help ease the checks on various things like null objects, empty strings, and so on, so that you don't write the same long lines of code in many tests.

Listing 1.4 shows what this test would look like with a little 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 just 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("1");
                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());
            }
        }
```

Making helper methods more generic to show error reports, so test are written more easily are just one of the things that unit test frameworks, which we'll talk about in chapter 2, help out with.

Before we get there, I'd like to discuss one important matter, regarding not just *how* you write a unit test, but *when* you would want to write it during the development process – that's where Test Driven Development comes into play.

1.6 Test-driven development

Even if we know how to write structured, maintainable, and solid tests with a unit test framework, we're left with the question of when we should write the tests.

Many people feel that the best time to write unit tests for software (if any at all) is after the software has been written. Still, there is a growing number of people who prefer a very different approach to writing software – writing a unit test *before* the actual production code is even written. That approach is called test driven development or TDD for short.

Figures 1.3 and 1.4 show the differences between traditional coding and Test Driven Development.

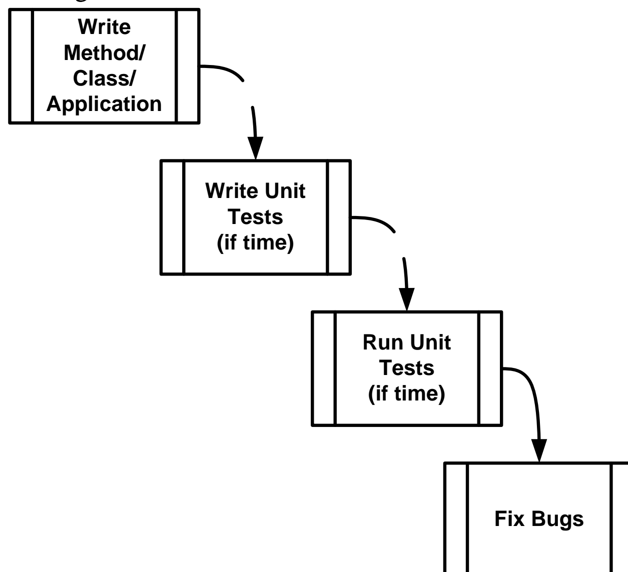

Figure 1.3 The "Traditional" way of writing unit tests. Dotted lines represented actions people treat as optional during the process.

Test Driven Development is vastly different than *classic* development as figure 1.5 shows. You actually begin by writing a test that fails, then move on to creating the actual production code, and see the test pass, and continue on to either refactor your code or to create another failing test.

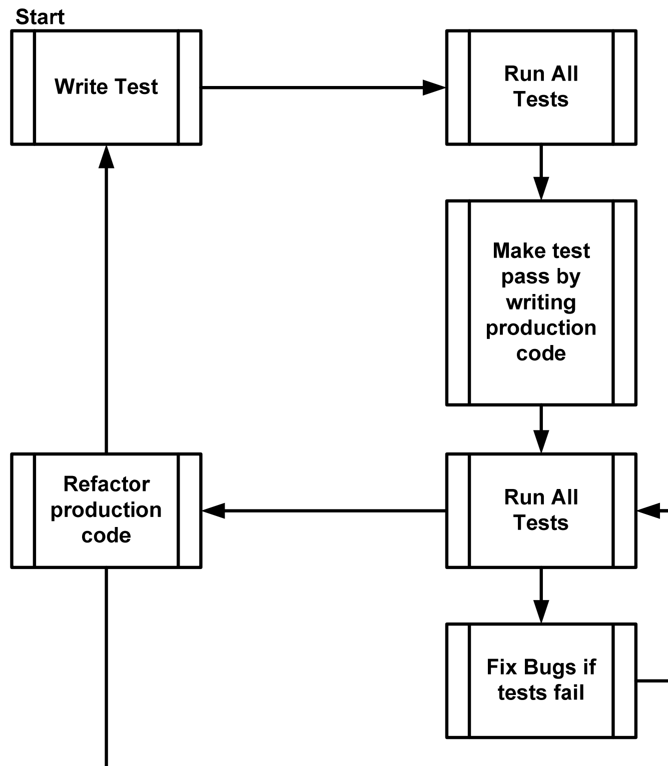

Figure 1.4 Test Driven Development - a bird's eye view. Notice the spiral nature of the process: write test, write code, refactor – write next test. It shows clearly the incremental nature of TDD: small steps lead to a quality end result.

You can read a more in-depth look at TDD in chapter 12 – "Test Driven Development", which is dedicated to this technique. That chapter also references other books you might want to read on this subject.. This book focuses on the technique of writing a good unit test, rather than test driven development, but the subject is so important it cannot go unwritten in any book that talks about unit testing.

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

It is important to realize TDD does not ensure project success or tests that are robust or maintainable. It is quite easy to get caught up in the technique of TDD and not pay attention to the way the unit test is written: its naming, how maintainable or readable it is, or whether it really does test the right thing or might have a bug. That's why I'm writing this book. But let's turn our focus for a moment to *test driven development.*

The technique itself 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 is already working, so the test failing means there is a bug in the production code. So, 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 some number as if it was working already. It will fail because we have not implemented that functionality yet.

2. Make the test pass	By writing production code that fits the reality that your test is expecting. It should be written as simply as possible.
3. Refactor your code	When the test passes, you are free to move on to the next unit test or refactor your code to make it more readable, remove code duplication, and more.

Refactoring can be done after writing several tests, or after each test. In any case, it is a very important practice, because it makes sure your code gets easier to read and maintain, while still passing all of the previously written tests.

REFACTORING

Refactoring is the act of 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. It still does the same thing; it's just easier to maintain, read, debug, and change.

These outlined steps sound very technical, but there is a lot of wisdom behind them. In chapter 21, I'll be looking closely at those reasons, and will be explaining all the benefits. For now, I can tell you the following without spoiling it for that chapter: done correctly, TDD can make your code quality soar, decrease the amount of bugs, raise your confidence in the code, shorten the time to find bugs, improve your code's design, and keep your manager happier. If TDD is done incorrectly, it can make your project schedule slip, waste your time, lower your motivation, and decrease/worsen? Your code quality. It's a double-edged sword, which many people only find out the hard way. In the rest of this book, we'll see how to make sure only the first part of this paragraph is true for your projects.

1.7 Summary

In this chapter we discussed the origins of unit tests. We then defined a good unit test as an automated piece of code that invokes a different method and then checks some assumptions on the logical behavior of that method or class, is written using a *unit testing framework*, can be written easily, runs quickly, and can be executed repeatedly by anyone on the development team.

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

I cannot stress enough the importance of realizing the difference between unit tests and integration tests. You will be using that knowledge in your day to day life as developers when deciding when to place your tests, what kind of test to write when, and which option is better for specific problems you will be dealing with. It will also help in identifying how to fix problems with tests that are already causing you some headaches. I will be talking later in the book even more about integration testing and best practices for those kinds of test.

We also talked about the cons of doing just integration testing: hard to write, slow to run, needs configuration, hard to automate, and more. Then we talked about the notion of unit test frameworks, talked about the XUnit frameworks that exist today and how they help us write, run, and review unit test results.

.NET in particular has several "competing" frameworks for it. It's up to you to choose which one to use in your day to day work. My recommendation would be to start off simple and advance to a more elaborate

framework later on, and only if you need it. Simplicity of the framework is key to actually having people use it. If the framework has many APIs and many ways to achieve one thing, it will take longer to teach and longer to start working with (as well as the tests will be harder to read and maintain, possibly).

Lastly, we talked about test driven development, how it's different than traditional coding, and the basic benefits of TDD. I've found TDD helps out in making sure that your code coverage in your test code is very high (close to 100% of *logical* code) and it helps make sure that your test can be trusted with a little more confidence by making sure that it indeed fails when the production code is not there, and passes when the production code actually works.

By writing tests *after* writing the code, you assume the test is OK just because it's passing. Trust me – finding bugs in your *tests* is one of the most frustrating things you can imagine. It's important you don't let your tests get to that state, and TDD is one of the best ways I know to make sure that possibility is close to zero.

In the next chapter we'll dive in and start writing our first unit tests using NUnit – the *de facto* unit test framework for .NET developers.