



Undedited Draft

The Art of Unit Testing
By Roy Osherove

Copyright 2006 Manning Publications

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

[Click here for new title updates](#)

Chapter 1 - The Basics of Unit Testing

This chapter is all about introducing the concept of *unit tests*, what they are, and especially what they *aren't*. Many misconceptions about what actually makes for a good unit test abound including how you should go about writing unit tests.

This chapter introduces the basic concepts of a unit test and the various test frameworks out there for .NET (the most common ones). The discussion also sets the stage for the other chapters in the book, where we'll dive into actually writing unit tests.

1.1 Unit Testing Defined

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

If you didn't have the pleasure of programming in the 1970s, chances are you're not very familiar with Smalltalk, a very effective, object oriented language for developing software systems.

It was designed and created in part for educational use, more so for Constructivist teaching—at Xerox PARC by Alan Kay, Dan Ingalls, Ted Kaehler, Adele Goldberg, and others during the 1970s—and influenced by Sketchpad and Simula (earlier programming languages). The language was generally released as Smalltalk-80 and has been widely used since. Smalltalk is in continuing active development and has gathered a loyal community of users around it.

Kent Beck is the person who introduced the concept of unit testing in Smalltalk as described in this web page 1:[<http://www.xprogramming.com/testfram.htm>]. The notions he has created have carried on into many programming languages today, as you'll read later in this chapter.

Smalltalk is still used but has been overshadowed by more popular programming languages such as VB and VB.NET, C# and Java. Still, it is available in many flavors for most operating systems today.

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 definition given in this chapter, 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.*

[Click here for new title updates](#)

1.1.1 Defining “a good unit test”

Most people who try to unit test their code either give up at some point or don't actually perform unit tests. 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. There's no point in writing a bad unit test.

To understand what a good unit test is, we'll need to go back a little bit to understand what developers have been doing so far in a software project.

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 that they wrote before letting it out from under their hands? 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 out their code. Of course, the UI may change, but the pattern is usually the same: use a manual external tool to check something repeatedly.

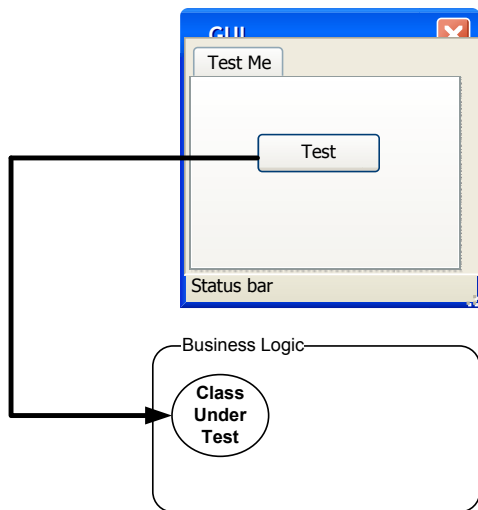


Figure 1.1 For classic testing, developers use a GUI (windows or web) and trigger an action on the class they want to test. Then they check the results somewhere (UI or other places)

While these may have been very useful, and, technically, they may be close to the “classic” definition of a unit test (and 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. We pose that question in the next section.

1.2 *What is not a unit test?*

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*.

1.2.1 *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 any one of these parts, or some 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 fail – all these software components fails as a team – it could get really hard finding out the true culprit of the failure of the final operation.

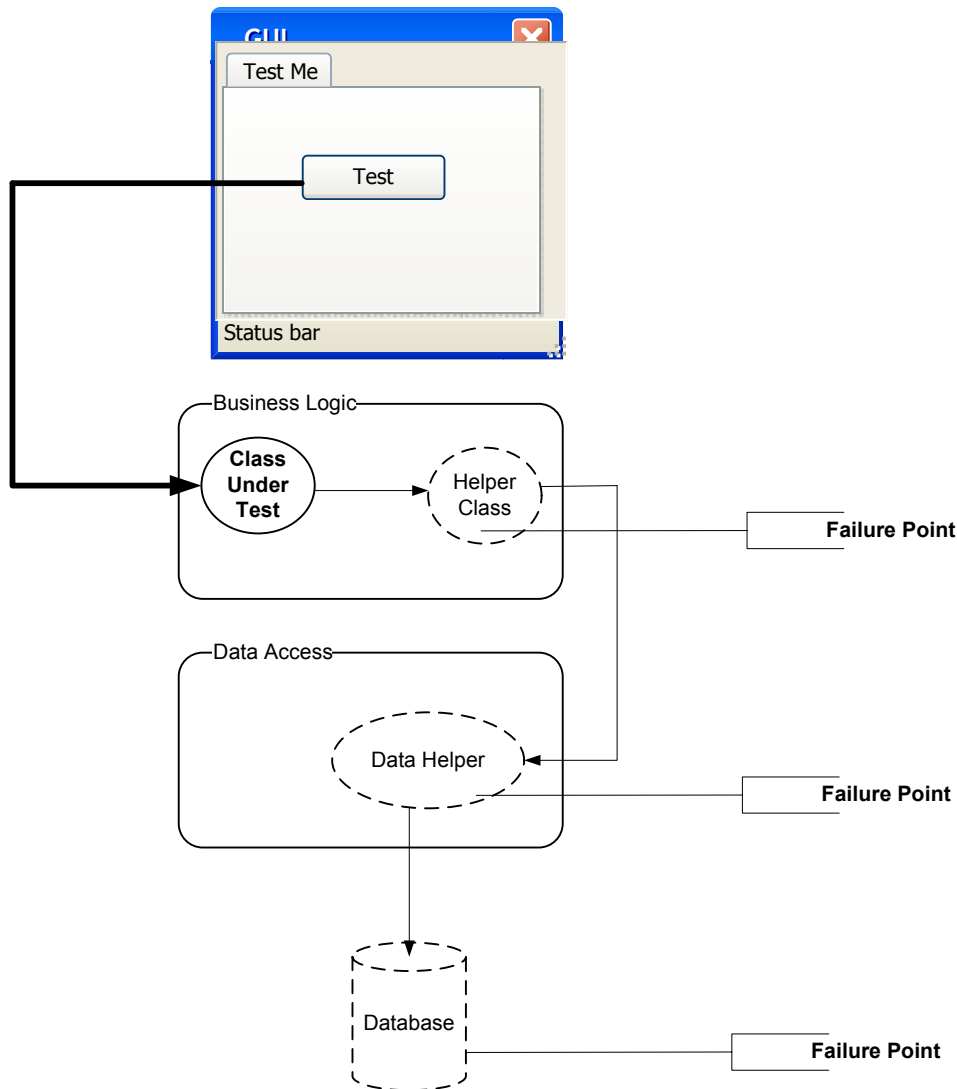


Figure 1.2 – You can have many failure points while trying to test one simple actions during an integration test because all the units have to play nicely together, and each of them could malfunction, making it harder to find the source of the bug. With unit tests, like the hero says in the movie *Highlander*, “There can be only one” (culprit).

With that description in mind, let’s look at the classic definition of Integration Tests. according to *Wikipedia*: Sometimes called Integration and testing and abbreviated I&T, integration testing is the phase of software testing in which **individual software modules are combined and tested as a group**. It follows unit testing and precedes system testing.

I’ve bolded the important words in the definition since 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).

Better Definition: Integration Testing

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 I've dedicated chapter 5 for a deep review of integration testing techniques using the various unit testing frameworks out there. (You'll learn more about unit testing frameworks in chapter 2.)

An integration test would exercise many units of code that work together to test a single result, while a unit test would test only a single unit of the whole, in isolation.

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

1.2.2 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.

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 knowing 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**.

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.

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. . ."

[Click here for new title updates](#)

Few things are scarier than not knowing whether the application still works or not, 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 that tell you whether you broke something anywhere in the system.

Test rule

Anyone can get and run the tests.

Question: Can it take me 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, 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*.

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

If you make the tests automated, you make them easy to run and enable your team to run them whenever they want.

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 implement, not only 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.

[Click here for new title updates](#)

Test rule

Tests should be easy and quick to write

From the lessons we've learned so far, we can start to answer the question posed in the next section.

1.3 What is a unit test?

Based on the previous sections and questions, 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 in the context of this book:

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.

Definition: A Good Unit Test

A unit test is an automated piece of code that invokes a different method and then checks some assumptions on 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 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 implemented unit tests so far. It makes us take a deep hard look at the way we, as developers, have implemented testing up until now, compared to how we'd like to implement it.

In the next section we'll take a look at what unit test frameworks are, and how they help us achieve good unit tests in an easier fashion.

1.4 Unit testing frameworks

Unit testing frameworks help developers write tests faster with a set of known APIs, execute those tests automatically and review the results of those tests clearly. Let's dive in to what that means exactly.

The tests you've done up until now were:

Not structured

You had to reinvent the wheel every time you wanted to test the feature. One test looks like a console application; the other is a UI form; and another is a web form. Reinventing the wheel takes time. You don't have that time to spend, and it fails the "easy to implement" requirement.

Not repeatable

[Click here for new title updates](#)

Neither you or your team can run tests you've written in the past. That breaks the "repeatedly" requirement and prevents you from finding regression bugs.

Not on *all* your code

All the code that matters, anyway. That means all the code that has pieces of logic in it, since each and everyone of those could contain a potential bug. (Property getter and setters don't count as logic, unless you actually have some sort of logic inside them.)

In short, what you're missing is a *framework* for writing, running, and reviewing unit tests and their results. Surprisingly, that's our next subject.

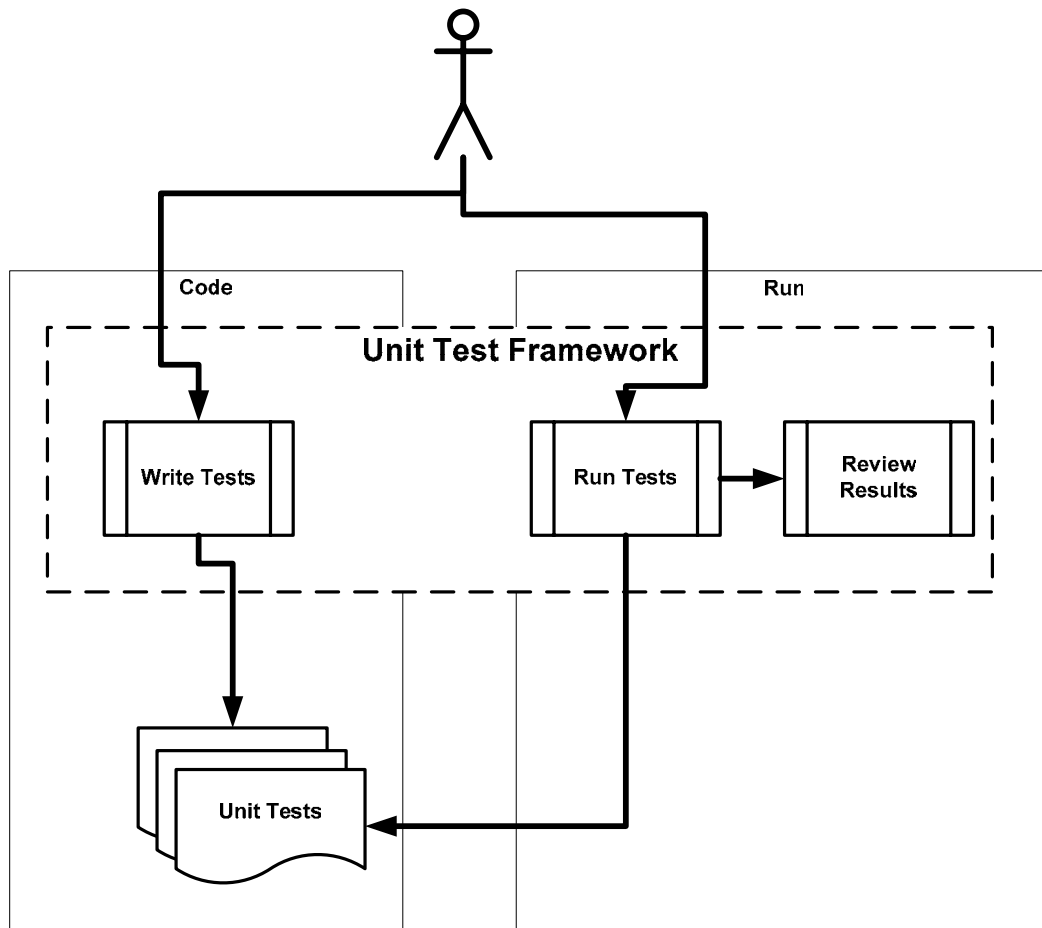


Figure 1.3 Unit tests are written as code using libraries of the unit test framework. Then the tests are run from a separate unit test tool and results are reviewed in the UI or as text results by the developer.

Unit test frameworks are code libraries and modules that help developers who would like to unit test their code do three things:

Write the tests easily and in a structured manner.

The framework usually supplies the developer with a set of method calls, classes, or interfaces that they would implement in their unit testing code. They also provide the developer with *ASSERTs*, which are special verification methods supplied by the unit test framework that provide a unified way of specifying whether the test has failed or not.

[Click here for new title updates](#)

If these ASSERTs fail, the system will collect all necessary information to let us know what, where, and why it failed.

Can execute one or all of the unit tests.

All those frameworks provide an automated way of running the unit tests that we've written using they're own supplied "Test Runner." The runner provides status as to how many tests were run, makes sure the tests are executed repeatedly and automatically.

Review the results of the test runs.

The test runners will usually let you know how many tests ran, how many didn't run, and how many failed. They will also tell you exactly which tests failed, the reason they failed (the ASSERT message), and the line number that failed. Some frameworks (including those for .NET) will give you the exact stack trace of any exceptions that have happened that caused the test to fail and will let you go to the various method calls inside the call stack. That way, you can more quickly find the reason for the test failure.

Many unit test frameworks are out there. There is practically one out there for every programming language in some form of public use. You can find a list of most of them at 1:[<http://www.xprogramming.com/software.htm>]. At the time of this writing, there are more than 150 of them, ranging in target language from C, C++, and .NET to Tcl, Ada, and AppleScript. .NET alone has at least nine different unit testing frameworks, among these, NUnit is the *de-facto* standard for writing unit tests in .NET.

Collectively, these unit testing frameworks are called "The X-Unit Framework," since their names usually start with the first letters of the language for which they were built. Hence, you might have "CppUnit" for C++, "NUnit" for .NET, and "HUnit" for the Haskell programming language; most of us are too young to have ever used it. Not all of them follow these naming guidelines, but most of them do.

In this book, I'll be using **NUnit**, the .NET flavor of a unit test framework. Although many frameworks are in .NET for unit testing, NUnit is considered the *de-facto* standard for use. Specifically, the next chapter deals with writing your first test with NUnit and explains the NUnit behavior and usage.

In the next section we'll talk about when you'd want to write the unit tests during the development process, and do a brief review of Test-Driven Development – a technique for writing unit tests while writing the source code.

1.5 *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 it's 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 to be tests is even written. That approach is called test-driven development (TDD).

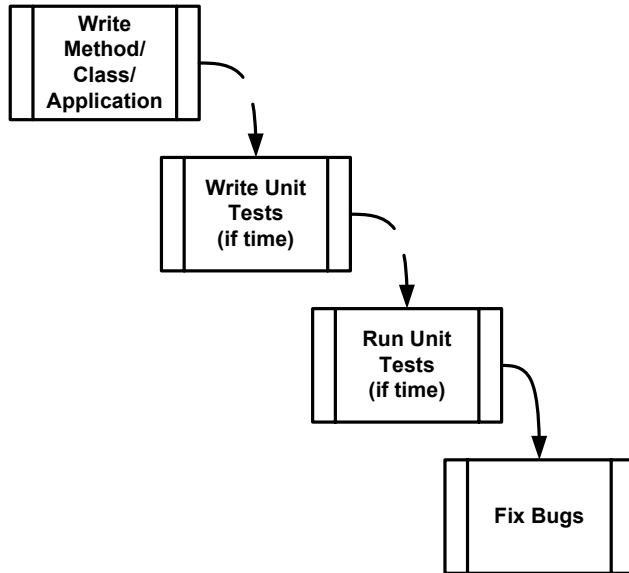


Figure 1.4 The “Traditional” way of writing unit tests. Dotted lines represented actions which people treat as optional during the process.

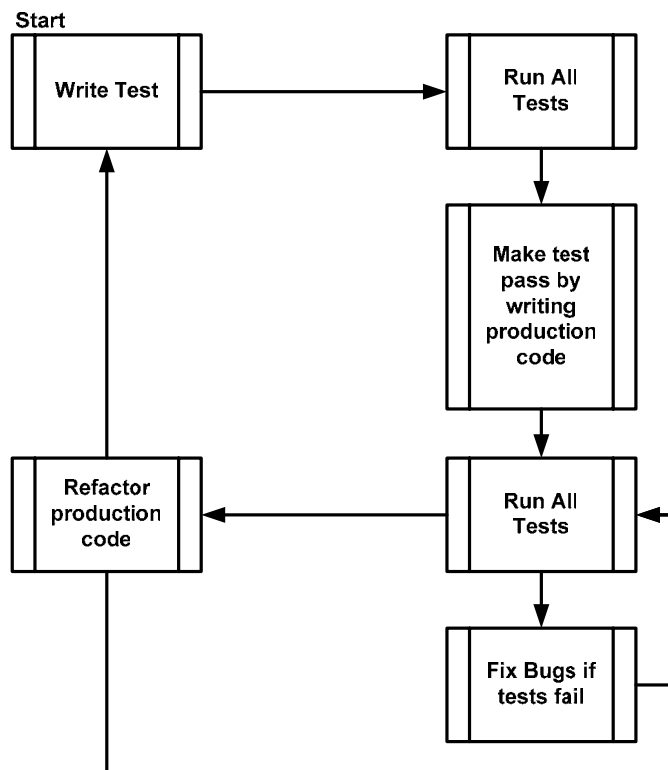


Figure 1.5 Test Driven Development - a bird’s eye view. Notice the spiral nature of the process: write test, write code, refactor – write next test. It shows clearly the incremental nature of TDD: small steps lead to a quality end result.

You can read a more in-depth look at test-driven development in chapter 21 which is dedicated to this technique, and many books out there deal with it specifically. 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. 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 test driven 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: it's naming, how maintainable or readable it is or whether it really does test the right thing or it might have a bug. That's why I'm writing this book. But let's turn our focus on to *test driven development*.

The technique itself is quite simple :

1. You write a test to prove that a feature or piece of code is missing or not working.
2. When you see the test fail, you can implement that piece of code, as simply as possible (harder than you think!)
3. 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 Definition

Refactoring is the act of changing a piece of code without changing its functionality. If you're 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.

Those steps sound very technical, but there is a lot of wisdom behind them. In chapter 4, I'll be looking closely at those reasons, and explain 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, the amount of bugs much lower, your confidence in the code higher, time to find bugs shorter, your code's design better, and your manager happier. If TDD is done incorrectly, it can make your project schedule slip, your time wasted, your motivation lower, and your code quality worse. It's a double-edged sword, which many people only find out the hard way. During the rest of this book, we'll see how to make sure only the first part of this paragraph will be true for your projects.

Summary

In this chapter we talked about 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 what was 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.

[Click here for new title updates](#)

We 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.

Lastly, we talked about test driven development, how it's different than traditional coding, and the basic benefits of TDD.

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.

[Click here for new title updates](#)