PART I
Getting Started

1

The Road to Test-Driven Development

WHAT'S IN THIS CHAPTER?

- How has software development evolved to bring us to TDD
- What an Agile methodology is and how does it differs from traditional waterfall-based technologies
- What TDD is and what the benefits of using it are

Test-Driven Development (TDD) has become one of the most important concepts and practices in modern software development. To understand why this is, consider the history of the practice of creating software. TDD was created through an almost evolutionary process. It came about as a response to the difficulties and challenges of writing software, but there was no real plan for its creation. It's a classic case of the traits of a thing that make that thing more successful and stronger being propagated and the traits that lead to failure being discarded. The practices of TDD were not created by any single company or individual; they rose from countless discussions (or, more likely, arguments) about what was done in the past, why it failed, and what could be done better. If TDD is a structure, such as a house, its foundation is created from failure. Failed projects, whose developers knew there had to be a better way, are what TDD has been built upon.

In this chapter you learn about the history of software development and how the methodology of managing software projects has moved from favoring waterfall to iterative to agile methodologies. You'll learn how the practice of Test-Driven Development is a key component of agile methodologies to ensure that quality code that addresses the business needs is being produced. I will explain the tenets of Test-Driven Development, outline its benefits, and show you an example of how Test-Driven Development is done.

THE CLASSICAL APPROACH TO SOFTWARE DEVELOPMENT

To understand the importance of TDD, it's necessary to see the road that led to it. Over the past 50 years the practice of software development has constantly evolved in an effort to find a balance between the needs of the business, the capabilities of the current technology, and the methodology in which developers are most productive. Missteps have occurred along the way, but even they were important as a means of determining which techniques and methodologies were evolutionary dead ends. This chapter reviews the road to TDD.

A Brief History of Software Engineering

Software development for business began during the age of the mainframe. Each hardware vendor seemed to have its own unique platform and paradigm for developing software. Sometimes these systems were similar enough to each other that developers could move from job to job and platform to platform with very little friction. Other times it was like starting from scratch. Although the basic concepts of computing were the same, each vendor had its own, sometimes very unique take on those concepts. Languages were archaic, often requiring many lines of code to do the simplest things that we take for granted today. And many times what worked in one implementation of a language or platform didn't work quite the same way in another.

The mainframe was a large, expensive piece of equipment. Many companies didn't own one, so the concept of the service bureau was born: Companies with a mainframe would lease time on their computer to customers. Unfortunately, this sometimes meant waiting for access to the computer. Imagine if you wrote a program today but couldn't compile it until next Monday. It would be very hard to be a productive developer with that kind of constraint. Suppose you attempted to compile on Monday but encountered an error. You could fix it, but you wouldn't know if your fix was correct for three more days. The limited access to computing resources often meant that testing, out of necessity, took a backseat to getting the product out the door.

These were also the days before the concept of *waterfall* development. Developers, left to their own devices, often worked in an iterative manner, scoping out specific pieces of a system and completing those, and then adding new features and functionality later. This method worked well, because it allowed developers to approach application development in a logical manner that kept things in terms they could understand and manage. Unfortunately, business users and what was logical and comprehensible to them often were not taken into consideration.

The second generation of mini-computers emerged in 1977, but they didn't really take off in business until 1979 with the release of VisiCalc. VisiCalc was the first spreadsheet application available for the personal computer. It demonstrated that PCs weren't just toys for the home, but machines that could provide real value to business. PCs offered many advantages over mainframes, the first one being that they were much less expensive. A business that couldn't afford even one dedicated mainframe could afford dozens of PCs. And although PCs weren't as fast as mainframes, their availability made them ideal for day-to-day tasks that didn't require the power of the mainframe. Developers could write applications for the PC and know right away if their code worked. They also didn't have to wait days to have their jobs scheduled and run.

Things got even better with third- and fourth-generation programming languages. They abstracted some of the more mundane tasks of their predecessors and allowed developers to be more productive by focusing on the business problem at hand. These languages also opened software development to a wider audience who didn't want to deal with the friction of languages such as Assembler and C. Business and the business computer industry ultimately settled on a few base languages and their derivatives. This helped developers become more attractive and marketable to business as their skills became more portable.

Ultimately business's need to plan brought about the waterfall project methodology. The concept behind waterfall was that every software project, whose average time span was about two years, should have every phase from inception to delivery planned from the start. This necessitated a long period at the beginning for requirements gathering. After all the requirements were gathered, they were "thrown over the wall" to the architects. The architects took the requirements and designed the system that would be built down to the smallest detail. When they completed this task, they threw the design over the wall to the developers, who built the system. After the developers completed their work, they threw the system to the Quality Assurance (QA) department, which tested the application. As soon as the application was validated, it was deployed to the users.

Software testing in a waterfall methodology was often a long, difficult, inefficient, and expensive process. QA testers would test applications by manually running through test scripts, which were documents that instructed the tester to carry out an action in the system and described the result the tester should observe. Sometimes these scripts ran into hundreds of pages. When a change was made to the system, it could take a tester two or more weeks to completely regression-test the system. Another issue was that often these test scripts were written by the developer who created the system. In these cases the scripts usually described how the system *would* act, not how it *should* act.

The first step toward TDD happened with the proliferation of automated QA testing tools. These recorded a series of actions a user takes on a user interface (UI) and allowed them to be played back later. This verified that the UI worked correctly. After the initial tests were recorded, the QA tools also allowed much faster regression testing than manual tests and could be run repeatedly. A large failing of many of these early tools was that the tests they created were brittle. When an aspect of the UI changed, the test usually couldn't handle the change, and the test would break. For tools that used the record/playback model, that meant the test had to be discarded and a new one created. Later versions of these tools allowed for scripting that would make some of these changes easier to absorb, but the tests still remained fragile.

From Waterfall to Iterative and Incremental

Software development doesn't happen in a void. It doesn't matter if it's an 18-month project to create an application to collate the Enterprise's Testing Procedure Specification (TPS) reports or a website that you built for your child's peewee hockey team; you are using a methodology. You have requirements, you plan features, and you build the application. After it's tested, you deploy it to a grateful user base.

A problem with the waterfall methodology is that all the requirements are gathered early on. In business, requirements often change for a variety of reasons. Changes in the law, a shift in the company's strategic direction, or even something as simple as a mistake in the requirements-gathering phase could have serious repercussions for the downstream process. The planned-out

nature of waterfall does not respond well to change. A change request to the system generally must go through the same requirements/design/development/QA process that the rest of the system did. This creates a ripple effect that causes the rest of the plan to become inaccurate.

To create the upfront plan, the work must be estimated early — sometimes years before the actual work is to be done, and usually by someone who won't actually do the work. This creates a house of cards in which one wrong estimate can again wreak havoc across the rest of the project plan.

The architects aren't blameless either. This era led to "ivory tower architects" who created designs for applications that in practice were impractical or, in some cases, impossible. Developers didn't help the case either, because many of them simply carried out the architect's design vision, whether or not it made sense. Many times what was delivered to the business (two years after it had been requested) did not remotely resemble what was wanted or needed.

In an effort to solve some of the issues with waterfall, some development shops turned to the concept of *iterative* or *incremental* development. The idea was to take a large waterfall project and divide it into several smaller waterfall projects. Each subproject would have a defined scope and delivery target and upon completion would feed into the next iteration of the larger project. This was an improvement, because it resulted in smaller projects that were easier to define and got software in front of users much faster. However, in the end this was really just several linked waterfall projects, albeit shorter ones. The individual subproject still did not have a good mechanism for dealing with the constant change of business and technology. Another step was needed.

A QUICK INTRODUCTION TO AGILE METHODOLOGIES

Unlike waterfall, which seeks to control and constrain the realities of software development, *agile* methodologies embrace them. Change in business is inevitable, and software development methodologies must be able to adapt. A key failure of the large up-front plan is that estimates by their very definition are always wrong. If they were correct, they wouldn't be estimates; they would be "the number." An iterative process shows promise, but the iterations themselves, and the methodology as a whole, must be flexible and open to change.

A Brief History of Agile Methodologies

In February 2001 several proponents of new methodologies such as Scrum, Extreme Programming (XP), Pragmatic Programming, Feature Driven Development, and others met and drafted the Agile Manifesto. It reads as follows:

> *"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> - *Individuals and interactions over processes and tools*
> - *Working software over comprehensive documentation*

- *Customer collaboration over contract negotiation*
- *Responding to change over following a plan*

That is, while there is value in the items on the right, we value the items on the left more."

The Agile Manifesto itself is not a development methodology. It doesn't prescribe how software should be developed. It simply states a set of key values that can be used to create and describe lighter and faster application development methodologies that are more focused on people, working software, and results than meticulous multiyear project plans and mountains of documentation.

Many branded agile development methodologies are in use today:

- Scrum
- Extreme Programming (XP)
- Feature Driven Development
- Clear Case
- Adaptive Software Development

The Principles and Practices of Test-Driven Development

These methodologies are all different in how they are implemented, but they share some characteristics:

- They all make communication across the team a high priority. Developers, business users, and testers are all encouraged to communicate frequently.
- They focus on transparency in the project. The development team does not live in a black box that obscures their actions from the rest of the team. They use very public artifacts (a Kanban board, a big visible chart, and so on) to keep the team informed.
- The members of the team are all accountable to each other. The team does not succeed or fail because of one person; they either succeed or fail as a team.
- Individual developers do not own sections of the code base. The whole team owns the entire code base, and everyone is responsible for its quality.
- Work is done in short, iterative development cycles, ideally with a release at the end of each cycle.
- The ability to handle change is a cornerstone of the methodology.
- Broad strokes of a system are defined up front, but detailed design is deferred until the feature is actually scheduled to be developed.

Agile methodologies are not a silver bullet. They are also not about chaos or "cowboy coding." In fact, agile methodologies require a larger degree of discipline to administer correctly. Furthermore, no one true agile methodology exists. Ultimately, each team needs to do what works best for them. This may mean starting with a branded agile methodology and changing it, or combining aspects

of several. You should constantly evaluate your methodology and do more of what works and less of what doesn't.

THE CONCEPTS BEHIND TDD

The history of TDD starts in 1999 with a group of developers who championed a set of concepts known as Extreme Programing (XP). XP is an agile based methodology that is based on recognizing what practices in software development are beneficial and dedicating the bulk of the developers time and effort to those practices under the philosophy "if some is good, more is better." A key component of XP is test-first programming. TDD grew out of XP as some developers found they were not ready to embrace some of the more, at the time, radical concepts, yet found the promise of improved quality that was delivered by the practice of TDD compelling.

As mentioned, agile methodologies do not incorporate a big upfront design. Business requirements are distilled into features for the system. The detailed design for each feature is done when the feature is scheduled. Features, and their resulting libraries and code, are kept short and simple.

TDD as a Design Methodology

When used as an application design methodology, TDD works best when the business user is engaged in the process to help the developer define the logic that is being created, sometimes going so far as to define a set of input and its expected output. This is necessary to ensure that the developers understand the business requirements behind the feature they are developing. TDD ensures that the final product is in line with the needs of the business. It also helps ensure that the scope of the feature is adhered to and helps the developer understand what *done* really means with respect to the current feature in development.

TDD as a Development Practice

As a development practice, TDD is deceptively simple. Unlike development you've done in the past, where you may sit down and start by creating a window, a web page, or even a class, in TDD you start by writing a test. This is known as *test first development*, and initially it might seem a bit awkward. However, by writing your test first, what you really are doing is creating the requirement you are designing for in code. As you work with the business user to define what these tests should be, you create an executable version of the requirement that is composed of your test. Until these tests pass, your code does not satisfy the business requirement.

When you write your first test, the first indication that it fails is the fact that the application does not compile. This is because your test is attempting to instantiate a class that has not been defined, or it wants to use a method on an object that does not exist. The first step is simply to create the class you are testing and define whatever method on that class you are attempting to test. At this point your test will still fail, because the class and method you just created don't do anything. The next step is to write just enough code to make your test pass. This should be the simplest code you can create that causes the test to pass. The goal is not to write code based on what might be coming in the requirement. Until that requirement changes, or a test is added to expose that lack of functionality, it doesn't get written. This prevents you from writing overly complicated code

where a simple algorithm would suffice. Remember, one of the goals of TDD is to create code that is easy to understand and maintain.

As soon as your first test is passing, add more tests. You should try to have enough tests to ensure that all the requirements of the feature being tested are being met. As part of this process, you want to ensure that you are testing your methods for multiple input combinations. This includes values that fall outside the approved range. These are called *negative tests*. If your requirement says that your interest calculation method should handle only percentage rates up to 20%, see what happens if you try to call it with 21%. Usually this should cause an exception of some sort to be thrown. If your method takes string arguments, what happens if you pass in an empty string? What happens if you pass in nulls? Although it's important to keep your tests inside the realm of reality, triangulating tests to ensure the durability of your code is important too. When the entire requirement has been expressed in tests, and all the tests pass, you're done.

THE BENEFITS OF TDD

When describing TDD to developers, development managers, and project managers who have never experienced it, I am usually met with skepticism. On paper, creating code does seem like a long and convoluted process. The benefits cannot be ignored, however:

- TDD ensures quality code from the start. Developers are encouraged to write only the code needed to make the test pass and thus fulfill the requirement. If a method has less code, it's only logical that the code has fewer opportunities for error.
- Whether by design or by coincidence, most TDD practitioners write code that follows the SOLID principals. These are a set of practices that help developers ensure they are writing quality software. While the tests generated by the practice of TDD are extremely valuable, the quality that results as a side-effect is an incredibly important benefit of TDD. The SOLID principals will be covered in Chapter 3.
- TDD ensures a high degree of fidelity between the code and the business requirements. If your requirements are written as tests, and your tests all pass, you can say with a high degree of confidence that your code meets the needs of the business.
- TDD encourages the creation of simpler, more focused libraries and APIs. TDD turns development a bit on its head, because the developer writing the interface to the library or API is also its first consumer. This gives you a new perspective on how the interface should be written, and you know instantly if the interface makes sense.
- TDD encourages communication with the business. To create these tests, you are encouraged to interact with the business users. This way, you can make sure that the input and output combinations make sense, and you can help the users understand what they are building.
- TDD helps keep unused code out of the system. Most developers have written applications in which they designed interfaces and wrote methods based on what might happen. This leads to systems with large parts of code or functionality that are never used. This code is expensive. You expend effort writing it, and even though that code does nothing, it still has to be maintained. It also makes things cluttered, distracting you from the important working code. TDD helps keep this parasite code out of your system.

- TDD provides built-in regression testing. As changes are made to the system and your code, you always have the suite of tests you created to ensure that tomorrow's changes do not damage today's functionality.
- TDD puts a stop to recurring bugs. You've probably been in situations where you are developing a system and the same bug seems to come back from QA repeatedly. You think you've finally tracked it down and put a stop to it, only to see it return two weeks later. With TDD, as soon as a defect is reported, a new test is written to expose it. When this test passes, and continues to pass, you know the defect is gone for good.
- When developing applications with testability in mind, the result is an architecture that is open, extensible and flexible. Dependency Injection (covered in Chapter 5) is a key component of both TDD and a loosely coupled architecture. This results in a system that by virtue of its architecture is robust, easy to change, and resistant to defects.

A QUICK EXAMPLE OF THE TDD APPROACH

The following exercise takes you through an example of what it's like to develop a feature for a system using TDD. For this example, imagine you have been asked to create a feature that counts occurrences of a character in a string. Assume that you are working in an existing solution, with an existing project structure, but the class you'll implement this method on does not exist. Also assume for this example that your unit-testing frameworks have been referenced in your project. Don't worry; I cover how to do this in Chapter 6. Currently, the solution looks like Figure 1-1.

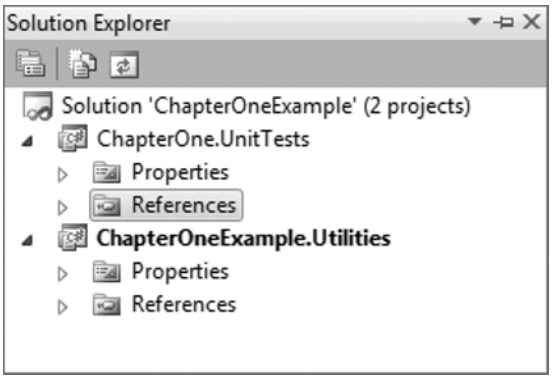

FIGURE 1-1

The ChapterOne.UnitTests project will contain our unit tests. The ChapterOneExample.Utilities project will be where our completed class will be placed. The first step is to create a class in our unit test project that will contain our unit tests, as shown in Figure 1-2.

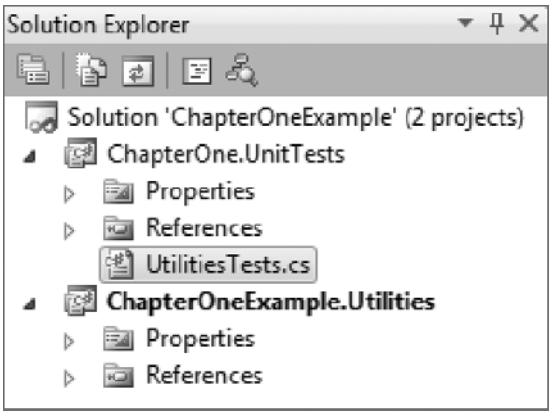

FIGURE 1-2

You have a variety of ways to arrange your unit test classes within your unit test project. Some developers prefer to place each test class in a separate code file. Some developers like to create a code file for all the test classes for a specific feature. A more common approach, which is the one taken here, is to create a code file class for all the unit test classes for a specific section of the application—in this case, the utilities project. If you had a business logic library with several business/domain-based services, you could

create a separate code file for each domain service's test classes. For this example that would be overkill, so you'll use one test class for the whole project.

When you created the `UtilitiesTest.cs` class, Visual Studio created some boilerplate code:

```
namespace ChapterOne.UnitTests
{
    public class UtilitiesTests
    {

    }
}
```

For the purposes of this example, the name UtilitiesTests is fine, but in a real business development situation it might not be descriptive enough for the other developers on your team. It definitely won't mean much to a nontechnical business user. Subsequent examples in this book will employ a method for naming and constructing tests that is more in line with a Business Driven Development style. It provides human-friendly names and makes the actual test easier to understand and follow for nontechnical people.

Now you write your first test. This can be the simplest expression of what your requirements are. This test passes in the string `mysterious` and asks the library to count the occurrences of the letter `y`:

```
using NUnit.Framework;

namespace ChapterOne.UnitTests
{
    public class UtilitiesTests
    {
        [Test]
        public void ShouldFindOneYInMysterious()
        {
            var stringToCheck = "mysterious";
            var stringToFind = "y";
            var expectedResult = 1;
            var classUnderTest = new StringUtilities();

            var actualResult =
                classUnderTest.CountOccurences(stringToCheck, stringToFind);

            Assert.AreEqual(expectedResult, actualResult);
        }
    }
}
```

The test method `ShouldFindOneYInMysterious` is decorated with the attribute `Test` to tell the unit test framework that this is a test. The test conditions are set up by defining the string to search and the character to find in it. They also define the expected result. Next the method is invoked under test and captures the actual result. Finally, an `Assert` statement determines whether the expected and actual values are the same.

The first indication that the test does not pass is the fact that the application does not compile. This tells you that no one has implemented a `StringUtilities` class in the application. That's what you must do first. To do so you simply add a new class called `StringUtilities` to your utilities project. The class that Visual Studio creates looks like this:

```
namespace ChapterOneExample.Utilities
{
    public class StringUtilities
    {

    }
}
```

The test still fails, because you haven't created a `CountOccurences` method on this class. The next step in making this test pass is adding that method:

```
using System;

namespace ChapterOneExample.Utilities
{
    public class StringUtilities
    {
        public int CountOccurences(string stringToCheck,
                                   string stringToFind)
        {
            throw new NotImplementedException();
        }
    }
}
```

This method initially throws an exception because so far the only reason this test has failed is due to a failure to compile. This might seem silly, but in TDD you don't take anything for granted; it's important to see your tests fail before you write your methods. This ensures that you are writing only enough code to make the test pass. When you run the test, it fails, as shown in Figure 1-3.

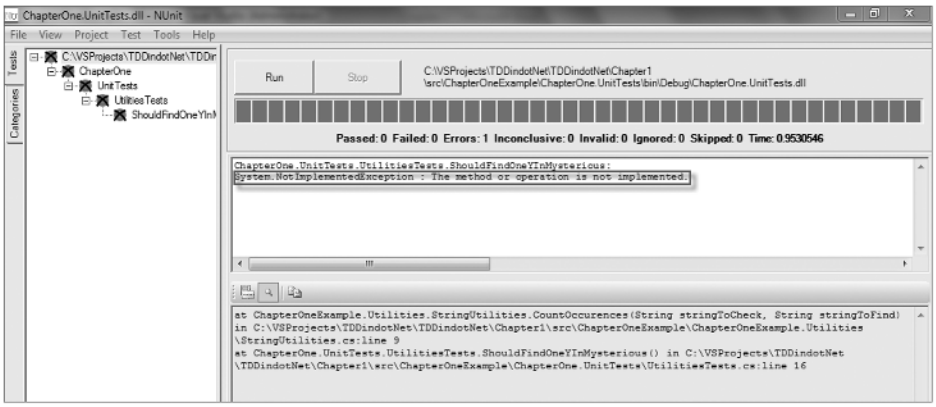

FIGURE 1-3

The reason for the test failure (as shown by the highlighted text) is that you have not implemented the method. The next step is to write code to make this test pass:

```
using System;

namespace ChapterOneExample.Utilities
{
    public class StringUtilities
    {
        public int CountOccurences(string stringToCheck,
                                   string stringToFind)
        {
            var stringAsCharArray = stringToCheck.ToCharArray();
            var stringToCheckForAsChar =
                                        stringToFind.ToCharArray()[0];
            var occuranceCount = 0;

            for (var characterIndex = 0;
                 characterIndex < stringAsCharArray.GetUpperBound(0);
                 characterIndex++)
            {
                if (stringAsCharArray[characterIndex] ==
                                             stringToCheckForAsChar)
                {
                    occuranceCount++;
                }
            }

            return occuranceCount;
        }
    }
}
```

This may or may not be the best way to implement this method, but if you run the test you can see in Figure 1-4 that it's enough to satisfy this requirement.

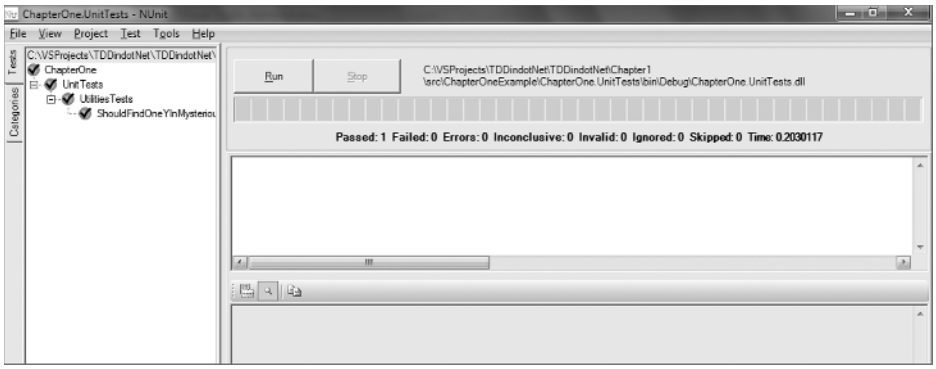

FIGURE 1-4

So, right now you know it works when there is one instance of the character you're looking for in the target word. In the interest of triangulating tests, you need to write another one to verify that it finds multiple instances:

```
[Test]
public void ShouldFindTwoSInMysterious()
{
    var stringToCheck = "mysterious";
    var stringToFind = "s";
    var expectedResult = 2;
    var classUnderTest = new StringUtilities();

    var actualResult = classUnderTest.CountOccurences(stringToCheck, stringToFind);

    Assert.AreEqual(expectedResult, actualResult);
}
```

When you run both tests, you can see that the code has a problem, as shown in Figure 1-5.

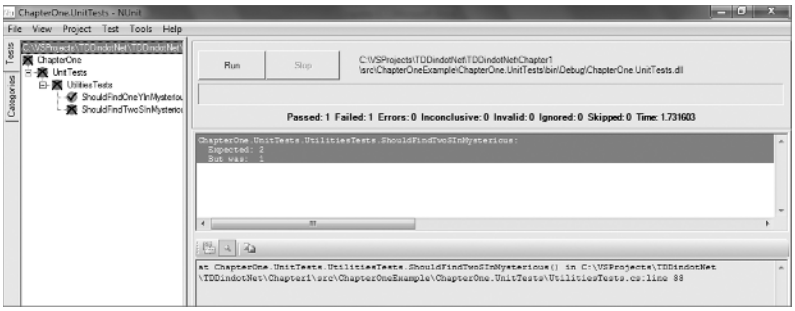

FIGURE 1-5

The test has uncovered a bug in the code. Specifically, the `for` loop is looping through the target string one fewer time than is needed (`string length - 1`). After the defect has been found, you can fix the code:

```
for (var characterIndex = 0;
     characterIndex <= stringAsCharArray.GetUpperBound(0);
     characterIndex++)
```

Now when you run the test, the code behaves the way it should, as shown in Figure 1-6.

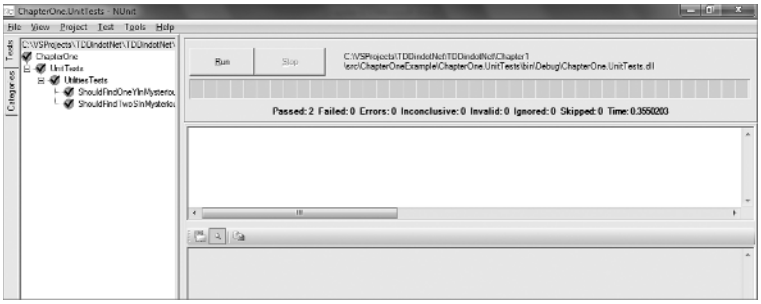

FIGURE 1-6

Now imagine that as you continue to develop your character counter method, you are given a new requirement. Your business user wants the search to be case-insensitive. That is, the algorithm shouldn't care if letters are uppercase or lowercase. Your first step is to write a test that expresses this new requirement:

```
public void SearchShouldBeCaseSenstive()
        {
            var stringToCheck = "mySterious";
            var stringToFind = "s";
            var expectedResult = 2;
            var classUnderTest = new StringUtilities();

            var actualResult =
                classUnderTest.CountOccurences(stringToCheck,
                                               stringToFind);

            Assert.AreEqual(expectedResult, actualResult);
        }
```

Figure 1-7 shows that when you run this test, the current implementation does not meet this new requirement.

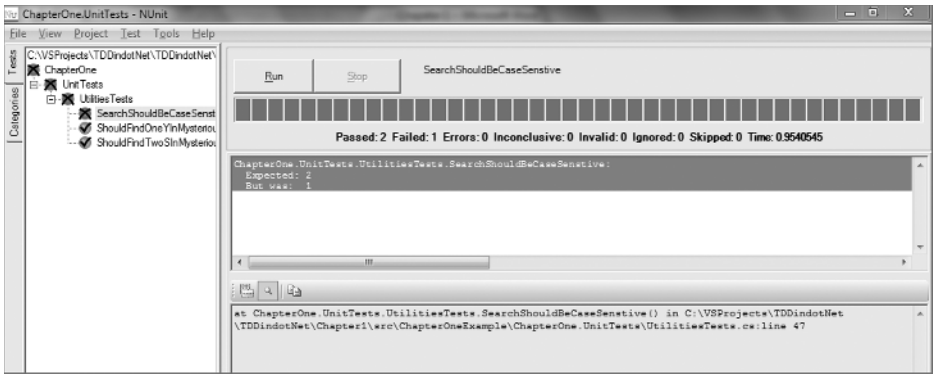

FIGURE 1-7

The next step is to update your method to make this test pass while making sure that the other two tests do not start to fail. This change is easy enough; you simply convert both the string you are searching and the character you are searching for to uppercase before you run the search algorithm:

```
var stringAsCharArray = stringToCheck.ToUpper().ToCharArray();
var stringToCheckForAsChar = stringToFind.ToUpper().ToCharArray()[0];
```

Figure 1-8 shows the results of running this test again. This change was all that was needed to make the new test pass, without causing the existing tests to fail, so this requirement is complete.

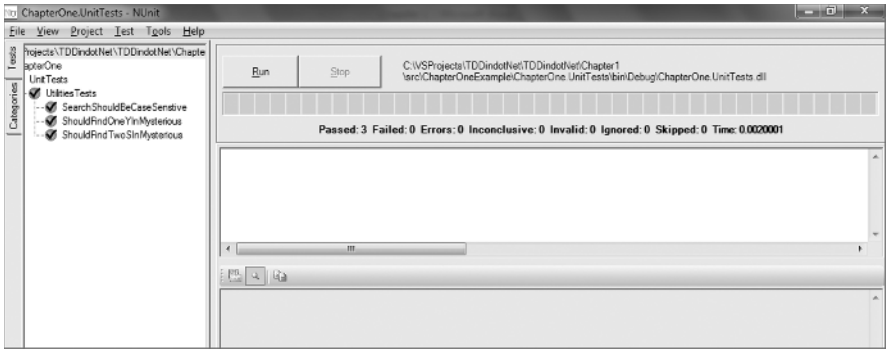

FIGURE 1-8

You deploy version one of your string utility class, and before long you have your first defect. When a user passes in a null as the string to be searched, a null reference exception is thrown. You can question the responsibility of the calling code to check its values before making the call, or argue that a null reference exception is appropriate; the string *is* null, after all. But the truth is that good developers realize that all input is evil and must be validated independently. And in the end, the business user would rather have the value –1 returned. You write a test to demonstrate this defect:

```
public void ShouldBeAbleToHandleNulls()
{
    string stringToCheck = null;
    var stringToFind = "s";
    var expectedResult = -1;
    var classUnderTest = new StringUtilities();

    var actualResult = classUnderTest.CountOccurences(stringToCheck, stringToFind);

    Assert.AreEqual(expectedResult, actualResult);
}
```

As expected, you can see in Figure 1-9 that this test fails when it is run.

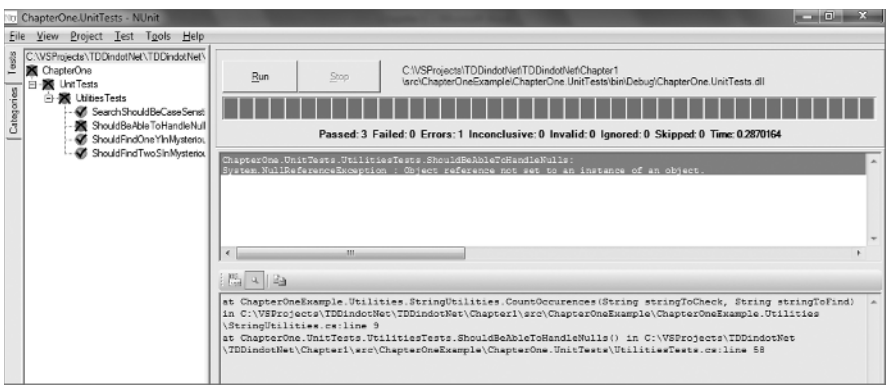

FIGURE 1-9

Another code change is needed, this time to validate the incoming arguments and return the appropriate response if the data fails validation:

```
public int CountOccurences(string stringToCheck, string stringToFind)
{
    if (stringToCheck == null) return -1;
    var stringAsCharArray = stringToCheck.ToUpper().ToCharArray();
```

When you run the test again, the code change corrects the defect, as shown in Figure 1-10.

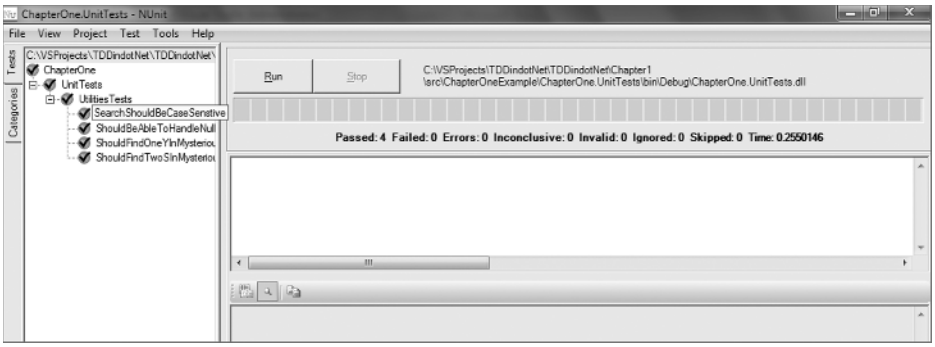

FIGURE 1-10

In addition to ensuring that you have fixed the defect, this test ensures that the defect doesn't reappear in the future.

SUMMARY

In this chapter you have seen how the history of software development has come full circle to a preference for iterative development. You also saw how the Agile Manifesto has created a framework for today's new breed of iterative methodologies. Software developers have also had to learn the value of change and find ways to adapt their work to the pace of change in the rest of the business. You saw a basic example of how Test-Driven Development (TDD) can be used to write robust software that is simple to implement and easy to maintain. You also learned how these tests can insulate you from introducing new defects while providing a framework for you to add new features without disrupting current ones. Finally, you learned what tools you need to start working with TDD.