Reengineering .NET

Injecting Quality, Testability, and Architecture into Existing Systems

Windows Development Series

Bradley **Irby**

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Reengineering .NET

Bradley Irby

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

The Library of Congress cataloging-in-publication data is on file.

ISBN-13: 978-0-321-82145-4
ISBN-10: 0-321-82145-9

Text printed in the United States on recycled paper at
R.R. Donnelley in Crawfordsville, Indiana.
First printing: October 2012

I would like to say thank you to my friends and family for their support and kind words; my children, Max and Lucas, for forgiving me for all those nights when I was writing and couldn't read them a story; and especially my wife, sweet Marcela, for her seemingly unlimited patience, understanding, faith, and encouragement. Many Chuicks!

Contents

Preface

What Is Software Reengineering?

Any developer who has been practicing his craft for more than a few years has been confronted with the task of enhancing an application that is difficult to work with. Navigating the code is difficult, figuring out where to start tracking down a defect is difficult, and making changes is difficult. Everything is difficult with these applications. Enhancements and bug fixes can be time-consuming, risky, and expensive.

One option for these legacy applications is to take them offline for a year or more to rewrite from scratch. Often these applications are so critical to the operation of the business, however, that feature development cannot be stopped for such an extended period of time. Therefore, work on the legacy system continues on, making patches and fixes to try to get through the next release cycle.

There is another option to help these legacy systems—software reengineering.

What Is Old Software?

After a software application is built, it immediately begins to age. Software engineering is a young field, and new ways of building applications are created every day. As new tools are introduced to the industry, if current applications are not retrofitted to use these tools, they become more and more difficult to maintain.

Causes of Software Aging

Software can become old for many reasons. The most obvious is the breathtakingly rapid pace of technology improvement in the world today. New software technology that was developed just a few years ago is already considered old and difficult to maintain.

The rapidity of job changes that is becoming a standard can also add to the deterioration of code. As the original developers pack their bags and move on to other companies, the original intent of much of the code is forgotten, leaving the remaining developers to pick up the pieces and hack together solutions as best they can.

By continually reengineering the system to modern technologies, this dependence on the original architects becomes less crippling. New developers can easily adapt to the system architecture because it is up to date and plenty of information can be found about it on the Web.

Warning Signs

Certain signs can tell when a system reaches the point it needs to be reengineered.

Developer Resistance to Feature Requests

If developers resist the efforts by management or users to enhance an application, it might be because the system is too difficult to work with. Over time, the software can become fragile, causing any feature development to become difficult and frustrating.

Large Bug Fixing Effort Immediately After a Release

If the development team is swamped with defect notices immediately after a new release of the software, it indicates a lack of modern quality tools. Part of the reengineering process involves introducing these automated quality tools so that the defect rate should decrease significantly.

Persistent Quality Problems

Old software can often display its age by the number of defects it contains and the effort necessary to fix them. The older and more fragile software gets, the more difficult it is to fix problems without breaking something else. If you see two defects appear for every one that is fixed, this is a sign that the application needs to be reengineered.

Legacy applications are especially prone to quality problems because they cannot support the new quality assurance approaches of Unit Testing and System Mocking. Without these tools in place, making changes to a system can result in creating a defect in an area seemingly and totally unrelated to the change made.

The Goal and Advantages of Software Reengineering

The goal of software reengineering is to incrementally improve an existing system by injecting modern architecture and software development techniques, while continuing to enhance the system with new features and while never having to take the system offline. This means we can take an existing system and slowly improve on it until it is brought up to modern software development standards without the need for a large, concerted rewrite effort. Throughout the reengineering project, the system is ready for production release. In other words, we can keep the plane in the air while we fix it.

Injecting Modern Architecture

The architecture of a software system is what determines how the many necessary details are built. Trying to use the latest approach to build a new data entry form for an old system is like trying to attach a jet engine to a biplane. You might get it off the ground, but the frame is not going to hold up for long.

The first reaction of most managers when they hear that the system architecture must be updated is to assume the application must be rewritten from scratch. This is not necessarily true. Bringing the architecture up to date can be done incrementally, as long as the new pieces are introduced in the proper order and using the proper steps.

Injecting a new architecture does not have to be a large effort by a team of people. The approaches described here can be introduced slowly by a small team of people (or even a single architect), regardless of the size of the full development team or the number of lines of code. Each step is a standalone element that can be introduced without affecting the rest of the application. Injecting a new architecture and quality measurements can be done without a large budget or dedicated team.

Adding New Features While Never Going Offline

The steps in this book are designed so they do not interfere with other development that might be going on simultaneously. The structures are introduced in such a way that they will have no detrimental effect while they are being added, but when complete can be turned on with a few lines of code. This enables the product manager to continue accepting new feature requests and pushing new versions into production, while in the background, each of these new versions is a little better than the last.

Any enhancement that takes more than a single day by a single developer has been designed to be pushed out in small steps so that the application can continue to work with some of the features converted to the new structure and some still using the old structure.

As each piece of the application is converted to the new design, it becomes much more testable, and the defect rate for the application should decrease dramatically.

Playing Well with Agile Approaches

Many development shops have adopted some sort of Agile development strategy. For those not familiar with this approach, a basic tenet of Agile is short development efforts (measured in days or weeks, and called a Sprint) at the end of which the application is in a potentially releasable state.

Keeping the application in a potentially releasable state is what reengineering is about. Each change made should be complete and self-contained, so that when the code is checked in, the system still runs perfectly but does so in a slightly better way and with higher quality. This is the way the steps in this book are designed. Each can be done within a single sprint and often in a single day. The few reengineering efforts that require more time are designed to have no impact on the system if they are halfway implemented. The system continues to run, just some features run the old way and some run the new way.

Reducing Risk

After years of working with a software system, business users fall into a pattern of how they use it to get their jobs done. Rewriting a system from scratch can lose touch with these undocumented processes, forcing the users to adapt their workflow to the new system.

Reengineering maintains these undocumented business processes that are part of the normal workings of the company. By slowly injecting new architecture into the existing system, these processes are left undisturbed. If introducing some new architecture does disrupt the normal flow of business in some way, there is immediate feedback because the application can be pushed into production on a fast schedule just like normal.

Reducing Cost

Rewriting a system from scratch requires that all of the business logic also be redeveloped from scratch. Legacy systems normally have a large investment of time and knowledge that is literally thrown away and must be recreated for a rewrite.

Reengineering saves that large investment and reuses the existing business logic code, saving significant amounts of time and money. The numbers of tasks that can be skipped with a reengineering project are significant. Fewer requirement documents must be created because the existing system already embodies the requirements. This can save weeks or months of a business analyst's time in researching and documenting all of the user's needs. The business logic to implement those requirements is also already done so there is even more savings.

In 1990, W. M. Ulrich wrote an article for the October issue of *American Programmer* in which he described a commercial system with an estimate of $50 million to rewrite from scratch. The same project was successfully reengineered for a total cost of $12 million. Reengineering can be significantly less expensive than new development.

Who Is this Book For?

This book is intended to help anyone involved in keeping these systems up and running. For technical managers and product managers, it describes the process necessary to improve the reliability of the system—make it faster, easier to maintain, and with fewer defects. For architects and developers, it contains detailed descriptions of the possible choices for the new architecture and the code necessary to implement the key pieces. It contains detailed suggestions on how to incrementally improve the structure and quality of an application by adding modern architecture and automated testing approaches. Following the steps outlined in this book can improve the quality of an application and make it easier and faster to add new features.

Reengineering an existing system is easier, cheaper, and less risky than building an equivalent system from scratch. If you follow the suggestions outlined in this book, you will be able to improve both the speed of development and the resulting quality, all while "keeping the plane in the air."

I hope you find this book a great resource and time saver.

—Bradley Irby
bradirby.com

Acknowledgments

It would have been impossible for me to write this book without the help of many people. First, I would like to thank my technical editors, Joey Ebright and Peter Himschoot, who did an excellent job in making sure my code was clean, correct, and to the point. Their architectural insight and suggestions made this book much better than it could have been had I made the attempt alone. I would also like to thank my editors, Christopher Cleveland and Jeff Riley, for their valuable input on structure and flow of the book. And finally, I would like to thank Joan Murray for having the confidence in me to let me take on this challenge. I hope I get a chance to work with everyone again on another book.

About the Author

Bradley Irby is an accomplished software architect and CTO. During his 25-year professional career, he has overseen the development of highly customized internal and customer-facing applications, including a property management system to manage the repossessed properties for Bank of America, a commercial accounting system for high-net-worth individuals, a property tax prediction system for the County of San Mateo, California, and a distributed reporting system for Chevy's Restaurants. His other work includes projects for General Electric, Kashi, Wells Fargo, HP, and Adidas, in addition to many projects for medium-sized companies and startups such as OpenTable and Prosper.com.

Bradley specializes in software reengineering and software migration, injecting quality and stability into existing legacy systems. Bradley has converted many applications from VB6, ASP Classic, and early .NET versions into more modern applications with current architecture and the latest quality approaches. His recent projects include reengineering a two million-line .NET application to use modern architectures and unit testing, resulting in a near zero defect count. He is an expert at updating applications without having to shut them down or stop feature development. Using a reengineering process Bradley developed, old applications can be updated to improve quality and satisfy existing customers, while also allowing continued feature development to keep pace with competitors and attract new customers.

Bradley manages the San Francisco .NET user group and is a frequent speaker on technical software topics throughout the U.S. He holds a bachelor of Computer Science degree from the University of North Carolina and an MBA from the University of California at Berkeley.

3

Unit Testing

The primary goal of the architectural patterns we discuss is to break our application into smaller pieces that have fewer references to each other to reduce coupling. This is an advantage because it becomes much easier for a class to evolve if there are fewer hard references to it that must be updated. It also introduces more opportunities for automated testing, or unit testing.

Unit testing is a way of automating much of the quality assurance function of a development department. The goal of unit testing is to isolate classes from the rest of the application so their methods can be tested via code. We write special test methods that exercise the system under test and ensure that it executes proper logic.

Unit tests are a critical part of the reengineering process. If your team does not have a unit-testing framework installed, now is the time to get one.

In this chapter, we touch on the high points of what unit tests are and how to implement them. There are many good books dedicated completely to unit testing; if you are new to the subject, read one (or more). We discuss the topic in as much depth as necessary to provide a continuous narrative. Though you learn the basics of how to create unit tests here, it is recommended that you take your skills to a higher level.

An Example of Unit Testing

Imagine you are writing a method that needs to calculate the payments to be made on a mortgage at a given interest rate. The caller can specify the initial amount of the loan, the number of years over which it is to be paid, and the interest rate. Your job is to write a method that can calculate the payments the borrower needs to make for any given loan configuration.

After you write this method, how do you test it? You can create a data entry screen that gives the user a way of entering the various values and then manually key in test data to cover the many different values for which you know what payments should be created. This approach is wasteful because it requires you to build a data entry screen that is subsequently thrown away and requires someone to manually test the results of the method before each production release of the product just to ensure that no changes are introduced that can break the calculation.

Now imagine writing code that would test the method for us. We can code something similar to the pseudo code in Listing 3.1.

LISTING 3.1: Pseudo Code for Unit Testing

```
mthlyPmt1 = CalcLoanPayment(Principal =1000, Interest=5%, NumMonths=360)
CheckForProperPayments(lst1)

mthlyPmt2 = CalcLoanPayment (Principal =1000, Interest=10%,
NumMonths=360)
CheckForProperPayments(lst2)

mthlyPmt3 = CalcLoanPayment (Principal =1000, Interest=10%,
NumMonths=240)
CheckForProperPayments(lst3)

…and more tests go here
```

If we write code to execute the tests shown previously, then we can run that test code anytime and we can ensure that the method is still working. We can even automate running the tests on each code check-in. This would virtually guarantee that the method is correct for all the scenarios we want to support and that it will never break.

Creating Unit Tests

Before we show a real example of a unit test, we need to look at how to create a unit test project. We will use the unit test project that is built into Visual Studio because it is a robust product that is well integrated into the IDE.

> **NOTE**
>
> We use the Microsoft Unit Testing framework, but there are many good frameworks available. The ideas we review here are applicable to any of these frameworks.

The Microsoft Unit Testing framework has a special project type to hold the unit tests, so the first thing we must do is add a new project. Right-click the solution in the Solution Explorer and choose **Add > New Project**. Choose a Test Project type and give it an appropriate name and then press **OK**. We use a separate test project for every standard project in the solution. Therefore, as shown in Figure 3.1, we name the test project with the name of the standard project and we append UnitTests to the end.

This creates a new project with a sample test already created and all the necessary references added. The sample test class comes with sample code that is not necessary right now, so to keep things simple, modify your sample test to look like this. This is all that is necessary for your first unit test; it simply ensures that True == True.

Listing 3.2: A First Simple Test

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace CodeSamples.Ch03_UnitTesting.Listing2
{
  [TestClass]
  public class ExampleUnitTest
  {
    [TestMethod]
    public void ExampleTest()
    {
```

```
            Assert.IsTrue(true);
        }
    }
}
```

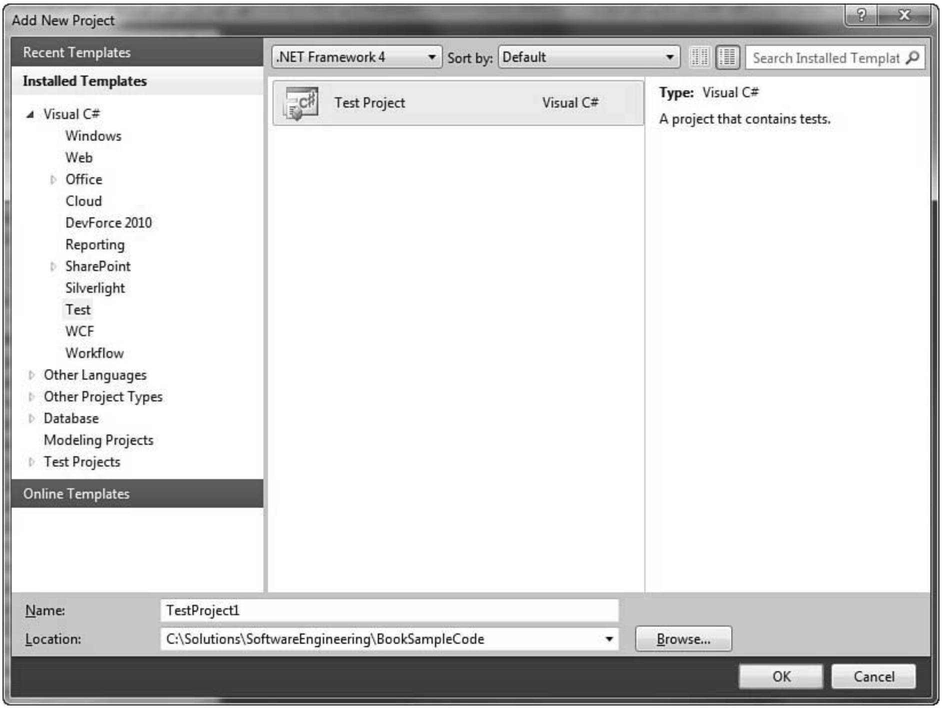

FIGURE 3.1: Adding a new test project

To run this test, right-click the name of the class and choose **Run Tests**, as demonstrated in Figure 3.2.

After running the test, the window shown in Figure 3.3 shows that the test passed.

Just so you can see what happens when a test fails, modify your code to make the test fail (see Listing 3.3).

LISTING 3.3: A Failing Test

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace CodeSamples.Ch03_UnitTesting.Listing3
{
  [TestClass]
  public class ExampleFailingUnitTest
  {
    [TestMethod]
    public void ExampleTest()
    {
      Assert.IsTrue(false);
    }
  }
}
```

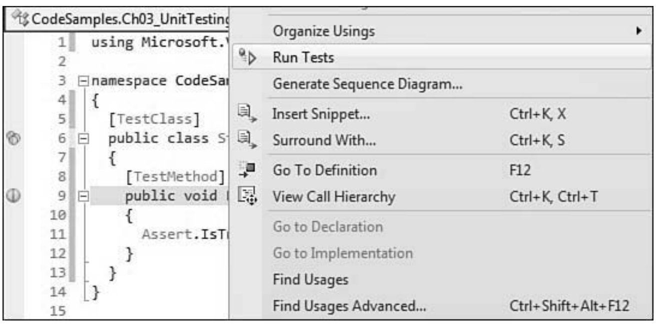

FIGURE 3.2: Running a test from within the IDE

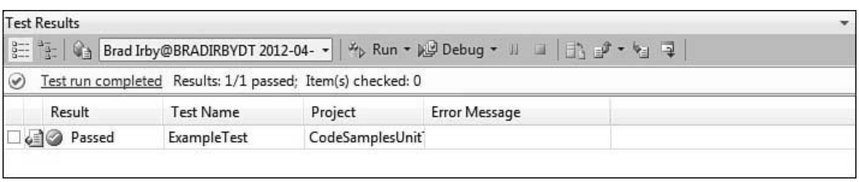

FIGURE 3.3: Test result screen

You should see something similar to Figure 3.4.

Now that we know how to create unit test files, let's get back to testing.

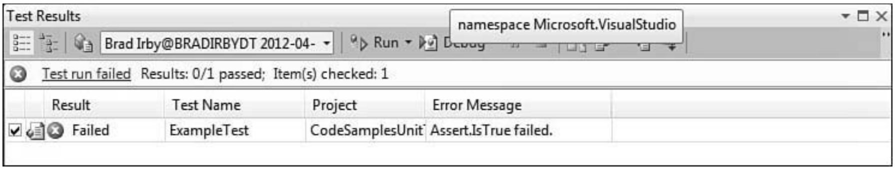

FIGURE 3.4: Failing test

Writing a Test

Let's look at an example of a class to be tested and the unit tests we build for it. The method shown in Listing 3.4 continues the previous example where we calculate the monthly payment for a loan.

LISTING 3.4: A Method Under Test

```
using System;

namespace CodeSamples.Ch03_UnitTesting.Listing4
{
  public class FinancialFunctions
  {
    /// <summary>
    /// P = (Pv*R) / [1 - (1 + R)^(-n)]
    /// where
    ///     Pv  = Present Value (beginning
    ///           value or amount of loan)
    ///     APR = Annual Percentage Rate
    ///           (one year time period)
    ///     R   = Periodic Interest Rate =
    ///           APR/ # of interest periods per year
    ///     P   = Monthly Payment
    ///     n   = # of interest periods for overall
    ///           time period (i.e., interest
    ///           periods per year * number of years)
    /// </summary>
    /// <param name="pLoanAmount">Original amount of the loan</param>
    /// <param name="pYearlyInterestRate">Yearly interest rate</param>
    /// <param name="pNumMonthlyPayments">Num of mthly payments</param>
    public double CalcLoanPayment(double pLoanAmount,
      double pYearlyInterestRate, int pNumMonthlyPayments)
    {
      var mthlyInterestRate = pYearlyInterestRate / 12;
      var numerator = pLoanAmount * mthlyInterestRate;
```

```
      var denominator = (1 - Math.Pow(1 + mthlyInterestRate,
               -pNumMonthlyPayments));
      var mthlyPayment = Math.Round(numerator / denominator, 2);
      return mthlyPayment;
    }

  }

}
```

We want to create some tests that ensure this method works for all cases that it supports. To begin, we write a test for a scenario that should produce no errors. To check the results, we can use one of the many mortgage rate calculators on the Web or Excel. Using one of these tools, we find that the payment for a $1,000 loan at 10 percent over 30 years should be $8.78 per month. In Listing 3.5, we write the test to pass in these values and `Assert` that the return value is valid.

Listing 3.5: A Sample Unit Test

```
using CodeSamples.Ch03_UnitTesting.Listing4;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace CodeSamples.Ch03_UnitTesting.Listing5
{
  [TestClass]
  public class FinancialFunctionsUnitTest
  {
    [TestMethod]
    public void CalcLoanPayment_Loan1000Int10Pmt360_
➥ReturnsProperPayment()
    {
      //Arrange
      var ex = new FinancialFunctions();

      //Act
      var pmt = ex.CalcLoanPayment(1000, 10, 360);

      //Assert
      Assert.AreEqual(8.78, pmt);
    }
  }
}
```

Let's take this test one line at a time. The first thing to notice is the `TestClass` attribute on the class. This is required to tell the unit-testing framework that this class is a test class. Each unit-testing framework has a different attribute, but they all require something similar to decorate test classes. On the test method, there is also an attribute labeling the method as a test; this is called `TestMethod`.

The name of the class itself is technically unimportant, so we typically name it the same name as the class that is tested and then append UnitTest to the end. The method name here is also technically unimportant because it will never be called by any code. The standard we use for naming unit test methods is to start with the name of the method or property being tested, add the conditions for the specific case being tested, and finally, add the results expected. In this case, we test that the `CalcLoanPayment` method, when called with a loan amount of $1,000, a yearly interest rate of 10 percent, and 360 monthly payments, it returns the proper payment, so we get the following name. The name should be descriptive enough so when you see a red "fail" indicator next to it in your test runs, you will know the scenario that has failed.

```
CalcLoanPayment_Loan1000Int10Pmt360_ReturnsProperPayment
```

The body of the test method is quite simple. We create a new instance of the class we wish to test and then call the method under test with the appropriate parameters. The only code in this method that we have not seen before is the `Assert` line. The `Assert` keyword is where we ensure that the method we test works properly. By using the `Assert` to test the results of our method, we ensure that our `CalcLoanPayment` method works properly for the given values. If the return value is what we expect, this test passes; otherwise, it fails.

This type of test structure is called Arrange-Act-Assert. The first step of the test is to arrange the classes so that they reflect the situation we want to test. We then act on the class under test to execute the code we want to test. Finally, we assert that the return value (or the actions taken) are appropriate for our test. When we run this test, we can see that our method returns the wrong value, as shown in Figure 3.5. We find our first bug!

Figure 3.5 shows the expected value and the actual returned value in the error message.

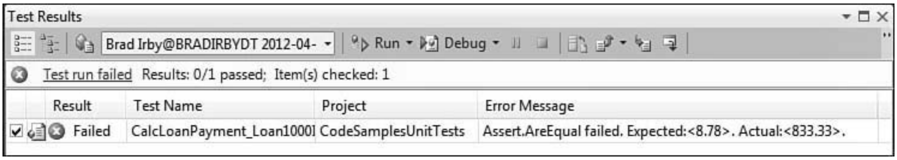

FIGURE 3.5: Failed first test

Upon examination of the code, we can see that the interest rate we pass in is a number between 0 and 100, whereas our code is expecting a number between 0 and 1. We can either change our test or change our code depending on which we think is in error. Because it is common to refer to yearly interest as 10 percent rather than 0.1, we allow users to pass in a number between 0 and 100 so that a 10 for the yearly interest is valid. That means we must update our business logic to accommodate this, as shown in Listing 3.6.

LISTING 3.6: A Fixed Unit Under Test

```
using System;

namespace CodeSamples.Ch03_UnitTesting.Listing6
{
  public class FinancialFunctions
  {
    /// <summary>
    /// P = (Pv*R) / [1 - (1 + R)^(-n)]
    /// where
    ///     Pv  = Present Value (beginning
    ///           value or amount of loan)
    ///     APR = Annual Percentage Rate
    ///           (one year time period)
    ///     R   = Periodic Interest Rate =
    ///           APR/ # of interest periods per year
    ///     P   = Monthly Payment
    ///     n   = # of interest periods for overall
    ///           time period (i.e., interest
    ///           periods per year * number of years)
    /// </summary>
    /// <param name="pLoanAmount">Original amount of the loan</param>
    /// <param name="pYearlyInterestRate">Yearly interest rate</param>
```

```
    /// <param name="pNumMonthlyPayments">Num of mthly payments</param>
    public double CalcLoanPayment(double pLoanAmount,
      double pYearlyInterestRate, int pNumMonthlyPayments)
    {
      //Change the divisor for the yearly interest rate
      var mthlyInterestRate = pYearlyInterestRate / 1200;

      var numerator = pLoanAmount * mthlyInterestRate;
      var denominator = (1 - Math.Pow(1 + mthlyInterestRate,
            -pNumMonthlyPayments));
      var mthlyPayment = Math.Round(numerator / denominator, 2);
      return mthlyPayment;
    }

  }
}
```

Running the test again proves that the code now works properly, as shown in Figure 3.6.

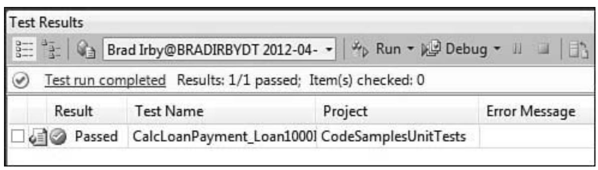

FIGURE 3.6: Fixed system under test

As shown in Listing 3.7, we can now add more methods to test different scenarios and ensure that the return values are valid.

LISTING 3.7: Additional Unit Tests

```
using CodeSamples.Ch03_UnitTesting.Listing4;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace CodeSamples.Ch03_UnitTesting.Listing7
{
  [TestClass]
  public class FinancialFunctionsUnitTest
  {
    [TestMethod]
    public void CalcLoanPayment_Loan1000Int10Pmt360_ReturnsPmt()
```

```
    {
      //Arrange
      var ex = new FinancialFunctions();

      //Act
      var pmt = ex.CalcLoanPayment(1000, 10, 360);

      //Assert
      Assert.AreEqual(8.78, pmt);
    }


    [TestMethod]
    public void CalcLoanPayment_Loan1000Int5Pmt360_ReturnsPmt()
    {
      //Arrange
      var ex = new FinancialFunctions();

      //Act
      var pmt = ex.CalcLoanPayment(1000, 15, 360);

      //Assert
      Assert.AreEqual(12.64, pmt);
    }

    [TestMethod]
    public void CalcLoanPayment_Loan1000Int10Pmt12_ReturnsPmt()
    {
      //Arrange
      var ex = new FinancialFunctions();

      //Act
      var pmt = ex.CalcLoanPayment(1000, 10, 12);

      //Assert
      Assert.AreEqual(87.92, pmt);
    }

  }
}
```

All of these tests should pass, as shown in Figure 3.7.

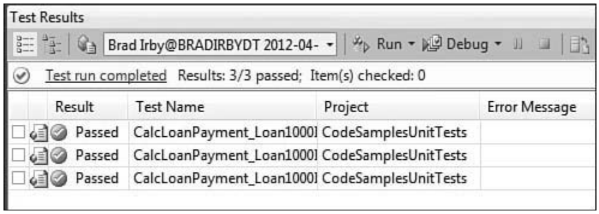

FIGURE 3.7: Successfully running unit tests

Detecting Exceptions

When writing your tests, be sure to include tests for invalid values. For instance, what happens if we pass a negative number to our method as an interest rate? This is a condition that should not be allowed, so we should throw an exception. Normally if an exception is thrown in the business logic, we want that treated as a test failure, but in this case, it should cause the test to succeed. To do this, we use the `ExpectedException` attribute.

Update the business logic to look like Listing 3.8.

LISTING 3.8: A Method That Throws an Exception

```
using System;

namespace CodeSamples.Ch03_UnitTesting.Listing8
{
  public class FinancialFunctions
  {
    /// <summary>
    /// P = (Pv*R) / [1 - (1 + R)^(-n)]
    /// where
    ///     Pv  = Present Value (beginning
    ///           value or amount of loan)
    ///     APR = Annual Percentage Rate
    ///           (one year time period)
    ///     R   = Periodic Interest Rate =
    ///           APR/ # of interest periods per year
    ///     P   = Monthly Payment
    ///     n   = # of interest periods for overall
    ///           time period (i.e., interest
    ///           periods per year * number of years)
    /// </summary>
    /// <param name="pLoanAmount">Original amount of the loan</param>
    /// <param name="pYearlyInterestRate">Yearly interest rate</param>
```

```
    /// <param name="pNumMonthlyPayments">Num of mthly payments</param>
    /// <returns></returns>
    public double CalcLoanPayment(double pLoanAmount,
      double pYearlyInterestRate, int pNumMonthlyPayments)
    {
      //start code change *******************************
      if (pYearlyInterestRate < 0)
        throw new ArgumentException(
          "pYearlyInterestRate cannot be negative");
      //end code change *******************************

      var mthlyInterestRate = pYearlyInterestRate / 1200;
      var numerator = pLoanAmount * mthlyInterestRate;
      var denominator = (1 - Math.Pow(1 + mthlyInterestRate,
            -pNumMonthlyPayments));
      var mthlyPayment = Math.Round(numerator / denominator, 2);
      return mthlyPayment;
    }

  }

}
```

With this business logic, we can write a test like this to ensure the exception is thrown. Listing 3.9 shows the new test added to the top of our existing tests.

LISTING 3.9: Testing a Method That Throws an Exception

```
using System;
using CodeSamples.Ch03_UnitTesting.Listing8;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace CodeSamples.Ch03_UnitTesting.Listing9
{
  [TestClass]
  public class FinancialFunctionsUnitTest
  {
    [TestMethod]
    [ExpectedException(typeof(ArgumentException))]
    public void CalcLoanPayment_Loan1000IntNeg10Pmt12_Throws()
    {
      //Arrange
      var ex = new FinancialFunctions();

      //Act
      ex.CalcLoanPayment(1000, -10, 12);
    }
```

```
    [TestMethod]
    public void CalcLoanPayment_Loan1000Int10Pmt360_ReturnsPmt()
    {
      //Arrange
      var ex = new FinancialFunctions();

      //Act
      var pmt = ex.CalcLoanPayment(1000, 10, 360);

      //Assert
      Assert.AreEqual(8.78, pmt);
    }


    [TestMethod]
    public void CalcLoanPayment_Loan1000Int5Pmt360_ReturnsPmt()
    {
      //Arrange
      var ex = new FinancialFunctions();

      //Act
      var pmt = ex.CalcLoanPayment(1000, 15, 360);

      //Assert
      Assert.AreEqual(12.64, pmt);
    }

    [TestMethod]
    public void CalcLoanPayment_Loan1000Int10Pmt12_ReturnsPmt()
    {
      //Arrange
      var ex = new FinancialFunctions();

      //Act
      var pmt = ex.CalcLoanPayment(1000, 10, 12);

      //Assert
      Assert.AreEqual(87.92, pmt);
    }

  }
}
```

In the first test, note the `ExpectedException` attribute just after the `TestMethod` attribute. Another item to note is the lack of an `Assert` statement. The `Assert` is not needed because execution of the test stops when the exception is thrown. If the exception is not thrown, the test fails due to the `ExpectedException` attribute.

This test should run as shown in Figure 3.8.

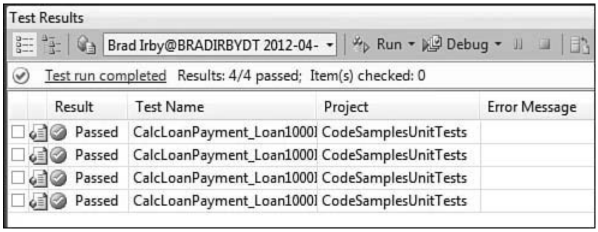

FIGURE 3.8: All four tests now pass.

Understanding the Power of Assert

The power of a unit test comes from the `Assert` keyword. There are many things that can be asserted aside from just true or false. We can test the equality of two items, whether they are null, or many other things, as shown in Figure 3.9. Of course the `Assert.IsTrue` is the most powerful because we can write any code we want to test a value and see whether that value is correct.

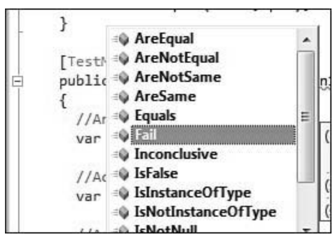

FIGURE 3.9: The many options of Assert

Comparing Unit Tests to Integration Tests

So far in this discussion, we have looked at unit tests, but there is another type of test called an integration test. A unit test can be loosely defined as a test that does not use any external resources such as the hard drive, a database, or the Web. It is completely self-sufficient for the resources it needs. In other words, it does not need to access anything but memory to execute the test, which should make a unit test fast to run. A unit test should also be repeatable, meaning that we can run the same test multiple times and always receive the same result. If a test relies on an external condition to pass and we do not have control of that external condition, it is not a unit test.

An integration test, on the other hand, uses external resources such as a database or a web service. These tests are often larger than unit tests because there are more moving pieces involved. For example, an integration test can ensure that a method can read several rows out of the database, process them properly by updating the data in the class representing the data row, and then save those rows back to the database. A unit test, on the other hand, assumes that the database read and write are working properly and it tests only that the given data was updated properly.

Integration tests are an important part of a full testing strategy, but we treat them separately because they are usually much more resource-intensive than unit tests. By using the outside resources, integration tests also normally take much longer. Unit tests for a project should run to completion in just a minute or two, but it's not uncommon to see integration test suites that run for an hour or more. The integration tests can be more thorough than a unit test, but the time commitment is so much greater that it is normally not possible to run integration tests just before a check-in like we do for unit tests. Therefore, integration tests are normally scheduled to be run once or twice a day.

Using the InternalsVisibleTo Attribute

A problem that plagues unit test writing is adding the proper references to the test project to allow access to the methods we want to test. The point of having unit tests is to exercise code and make sure it runs properly. However in a well-designed class, there are several methods that should be private to the class so they cannot be accessed by outside code. This presents a problem because this also limits the capability of our test to access the method.

One solution is to make public all methods that we test, but this can lead to problems later when internal methods are used inappropriately, making a class unstable. Imagine a new developer coming aboard who is unaware a certain method was made public simply to test it. He might think all public methods are available for use and then use this method, potentially introducing serious defects into the code.

Another solution is to always place the test classes in the same project as the live code. If we do this and set the methods to Internal, then we get past the problem; however, now the test code gets compiled into the same DLL as the live code, which causes the files to be bloated.

The .NET framework has addressed this problem with the `InternalsVisibleTo` project attribute. By adding this to the AssemblyInfo.cs file, we can expose an internal property or method only to a specific project in the solution. This enables the test class to access any methods that are necessary while still maintaining the proper encapsulation for the business logic.

Figure 3.10 shows where you can find the `AssemblyInfo.cs` file.

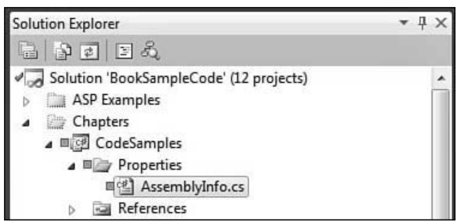

FIGURE 3.10: AssemblyInfo.cs File Location

Listing 3.10 shows what the `InternalsVisibleTo` attribute looks like. See the last line of the listing for an example.

LISTING 3.10: An InternalsVisibleTo Example

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// General Information about an assembly is controlled through the
➥following
// set of attributes. Change these attribute values to modify the
➥information
// associated with an assembly.
[assembly: AssemblyTitle("CodeSamples")]
[assembly: AssemblyDescription("")]
```

```
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Reengineering .NET")]
[assembly: AssemblyProduct("CodeSamples")]
[assembly: AssemblyCopyright("Copyright © Brad Irby")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this assembly not
➥visible
// to COM components. If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(false)]

// The following GUID is for the ID of the typelib if
// this project is exposed to COM.
[assembly: Guid("693b33bf-7738-42b0-b743-ed05a8d28d8e")]

// Version information for an assembly consists of the following four
➥values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
// You can specify all the values or you can default the
// Build and Revision Numbers
// by using the '*' as shown below:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]

[assembly: InternalsVisibleTo("CodeSampleUnitTests")]
```

The string is the project name you want to grant use of internal methods to (that is, the test project). This line should be added to the `AssemblyInfo.cs` file in the live-code project that contains the logic to be tested. Listing 3.11 shows an example of a private method that is marked `Internal` to allow access by the test.

LISTING 3.11: A Sample Internal Method

```
namespace CodeSamples.Ch03_UnitTesting.Listing11
{
  public class SampleInternalMethod
  {
    /// <summary>
```

```
    /// Sample method with the internal setting
    /// </summary>
    internal void InternalMethod(string pMessage)
    {
      //code goes here
    }

  }
}
```

Understanding Test Driven Development

Our discussion of unit testing would not be complete without addressing the Test Driven Development (TDD) movement. TDD has recently become popular as a way of building high-quality software by writing unit tests before writing the production code. Extreme TDD practitioners write the first test before coding a single line of business logic, in which case the test "fails" because the application won't build.

The assumption in TDD is that the developer should know the requirements for the code (or they shouldn't be writing code yet), so these requirements should be enforced via unit tests and the business logic written to satisfy the tests. Then the cycle repeats where another unit test is written and the production code is updated to make the test pass.

This approach has advantages when building new code, but when reengineering an existing system, TDD is a difficult approach to use. Because we are taking an existing code base and trying to enhance it with new architecture, it is impossible to write the test before writing the code. Also, because most systems that require reengineering are in such a state due to tight coupling, adding unit tests to this code is difficult or impossible. For these reasons, we do not take a TDD approach to our tests.

Learning More About Unit Testing

This chapter provides a brief introduction to unit testing, intended to give enough background so anyone who is not familiar with this process can continue reading the book and understand what is discussed. There are many books on the market that do an excellent job of explaining this topic in more detail. For example, *Software Testing with Visual Studio 2010* by Jeff Levinson is an excellent choice for .NET-focused examples. Also, *The Art of Unit Testing* by Roy Osherove is thorough. If you are new to unit testing, it is well worth the time to read one of these books.

Summary

This chapter introduced methods of ensuring the quality of your application in an automated fashion. By creating unit tests, we are able to write code that tests code, introducing dramatic improvements in quality because we can prove that code is working as it should be. Unit tests also enable us to thoroughly test code more often because these tests are now available at the push of a button.

This chapter discussed how to create your first unit tests, how to structure the tests to properly exercise your application, and how to detect exceptions that your code might generate in response to error conditions. We discussed the important distinction between integration tests and unit tests, and what that difference means to developers and to the build master. Finally, we introduced some Visual Studio-specific features of unit testing that can make your test-writing life much easier.

In the next chapter on the Dependency Inversion Principle, we begin to delve into automated ways of creating services and other objects so consumers can be insulated from the details of how to create those services.

Index

A

B

C

D

K–L

N–O

P

S

T

U