Reengineering .NET
Injecting Quality, Testability, and Architecture into Existing Systems

Bradley Irby

FREE SAMPLE CHAPTER
SHARE WITH OTHERS
Reengineering .NET

Bradley Irby
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The .NET logo is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries and is used under license from Microsoft.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

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.

Copyright © 2013 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.


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!
This page intentionally left blank
# Contents

*Preface xiii*
*Acknowledgments xix*
*About the Author xxi*

## PART I  Target Architecture

1 Implementing a Service-Oriented Architecture . . . . . . . 3

- An Overview of the Service-Oriented Architecture ................. 4
- Understanding Standardized Service Contracts .......................... 6
- **Interfaces** ................................................................. 6
- Understanding Coupling .................................................... 12
- Understanding Service Abstraction ....................................... 15
- Designing Reusable Services ............................................. 18
- Understanding Service Autonomy and Service Composability ....... 18
- Understanding Service Statelessness .................................... 19
- A Service Example ......................................................... 24
- Summary ................................................................. 26
Reengineering Project Recommendations ......................... 193

Constants ......................................................... 194

Data Transfer Objects (DTO) Projects ............................. 195

Interfaces ......................................................... 196

Services ......................................................... 197

Domain Model Projects ........................................... 198

Repository Projects ............................................... 199

Controllers, View Models, and Presenters ....................... 199

Refactoring to the Solution Structure .............................. 200

Remove Unnecessary Using Clauses ............................... 200

Separate Unit Tests and Integration Tests ....................... 201

Move Classes to Appropriate Projects ............................ 202

Move Shortcuts to Libraries ..................................... 202

Refactorings that Affect Logic .................................... 203

Move Initialization Logic into
the Constructor .............................................. 204

Replace Nested IF Statements with Guards ....................... 205

Removing Access to Entity Class Constructors .................. 210

Summary ......................................................... 211

10 Establishing the Foundation ................................. 213

Adding New Projects ........................................... 213

Using Prism, Unity, and Enterprise Library Versions ......... 214

Adapting the Shell .............................................. 216

Creating the IBaseView .......................................... 217

Adapting the Current Shell ...................................... 218

Adding a Shell Controller ....................................... 220

Creating the Service Locator .................................... 220

Setting Up the BootStrapper Class .............................. 223

Creating the Winforms BootStrapper ............................. 223

Updating the Winforms Program Class ........................... 226
Creating a WPF Application and Bootstrapper ............ 228
Using Alternative Bootstrapper Configurations .......... 232
Summary .................................................. 236

11 Basic Refactoring to Services ......................... 237
Using DialogService .................................... 238
Unit Testing ........................................... 242
Refactoring for DialogService ......................... 249
Adding Unit Tests .................................... 250
Using LogWriterService ............................... 251
Refactoring for LogWriterService ..................... 254
Tracking Session Information ......................... 257
Refactoring for Session Information .................. 258
Accessing Resources the SOA Way ................... 260
Refactoring for ResourceProvider ..................... 264
Using a Message Aggregator ......................... 265
Refactoring for MessageAggregator .................. 266
Converting Static Classes ............................ 271
Refactoring Static Classes ............................ 272
Summary .................................................. 273

12 Advanced Refactoring to Services ................. 275
Using a Repository Pattern .......................... 275
Creating a Repository with a Domain Model .......... 283
Reengineering Methods to a Repository ............... 288
Converting Existing Code to
Use a Domain Model .................................. 289
Adding Data Validations to the Domain Model ....... 291
Reengineering Domain Models to Use Validations ... 296
Using a Generic Object Manager ..................... 296
Simplifying Complex Code with a Command
Dispatcher Service .................................... 303
Refactoring for CommandLineInterpreter .......... 313
Summary .................................................. 314
Contents

13 Refactoring to a Controller ........................................ 315
   Using the Legacy Approach to Form Creation ................. 316
   Preparing the View .............................................. 319
   Introducing the Controller ...................................... 320
   Enhancing the Controller ....................................... 322
   Summary .......................................................... 325

Appendix  Reengineering .NET Projects with
           Visual Studio 2012 ....................................... 327
           Examining Source Control with Visual Studio 2012 ..... 327
           Managing Parallel Development ............................. 330
           Making Changes in Isolation ............................... 334
           Unit Testing with Visual Studio 2012 ..................... 337
           Writing a Unit Test Method .................................. 338
           Running the Unit Test ....................................... 339
           Using the Edit-and-Continue Feature ....................... 341
           Using Continuous Test Runner .............................. 344
           Using Fakes to Write Unit Tests for “Untestable”
           Code ............................................................ 346
           Looking for Hard-to-Maintain Code Using
           Code Metrics .................................................. 348
           Looking for Code Duplicates ................................. 350
           Summary ........................................................ 353

Index ................................................................. 355
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
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.
This page intentionally left blank
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.
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**

```java
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

```csharp

namespace CodeSamples.Ch03_UnitTesting.Listing2
{
    [TestClass]
    public class ExampleUnitTest
    {
        [TestMethod]
        public void ExampleTest()
        {
```
CHAPTER 3: **Unit Testing**

```csharp
Assert.IsTrue(true);
```
LISTING 3.3: A Failing Test


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.
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**

```csharp
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
        ///     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;
            // Further calculations...
        }
    }
}
```
Writing a Test

```csharp
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**

```csharp
using CodeSamples.Ch03_UnitTesting.Listing4;

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.

![Test Results]

**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**

```csharp
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>
```
CHAPTER 3: Unit Testing

/// <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.

![Test Results]

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;

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

```csharp
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>
```
Detecting Exceptions

/// <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**

```csharp
using System;
using CodeSamples.Ch03_UnitTesting.Listing8;

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);
        }
    }
}
```
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.
Comparing Unit Tests to Integration Tests

This test should run as shown in Figure 3.8.

![Test Results](image)

**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.

![Assert Options](image)

**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.

![AssemblyInfo.cs File Location](image)

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**

```csharp
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("")]
```
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**

```csharp
namespace CodeSamples.Ch03_UnitTesting.Listing11
{
    public class SampleInternalMethod
    {
        /// <summary>
        // 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("1.0.0.0")]
        [assembly: AssemblyFileVersion("1.0.0.0")]
        [assembly: InternalsVisibleTo("CodeSampleUnitTests")]
    }
}
```
/// 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
Abstract Factory Pattern, implementing, 74-79
abstraction, explained, 15-18
accessing
data (Repository pattern), 36-37, 199, 275-296. See also Dependency Inversion Principle
creating repository, 283-288
data validations in domain model, 291-296
detecting, 154-155
existing code conversion, 289-291
method reengineering, 288-289
test doubles and, 116-119
data models, 30-31
resources (ResourceProvider), 260-265
Activate method, 229
adapting shells, 216-217
adding shell controllers, 220
current shells, 218-219
IBaseView, 217-218
adding
new projects, 213-214
shell controllers, 220
advanced services, refactoring for, 166, 275-314
command dispatcher service, 303-313
generic object management, 296-303
repository pattern, 275-296
agile development management in defect tracking systems, 180
agnostic services, 4-5
alternative configurations,
  Bootstrapper, 232-236
analyzing
code, 138-142
  application controls versus form controls questions, 142
  basic architecture questions, 138-139
code structure questions, 139-140
database access questions, 140
Index

- data structure questions, 140-141
- external interface questions, 141-142
- code structure, 142-144
- Apache Subversion (SVN), 177
- application-agnostic projects, 192
- application-agnostic services, 4-5
- application controls, form controls versus, 142
- application-specific services, 5
- architectural patterns
  - choosing, 45
  - how they work, 43-44
- model access, 30-31
- MVC (Model-View-Controller), 27-29
  - form refactoring, 315
  - model access, 30
  - structures, MVVM structures versus, 193-194
- UI events, handling, 41-42
- MVP (Model-View-Presenter), 27-29
  - model access, 30
  - UI events, handling, 38-39, 42-43
- MVVM (Model-View-ViewModel), 27-29
  - form refactoring, 315
  - model access, 30
  - structures, MVC structures versus, 193-194
  - UI events, handling, 39
  - view models, 31-38
- Arrange-Act-Assert test structure, 54

The Art of Unit Testing (Osherove), 66

- Assert keyword, 54
  - uses of, 61
  - when not to use, 60
- automated code conversion utilities, 150-151
- autonomy, 18-19

B
- backups of source controlled code, 174-175
- basic architecture questions, 138-139
- basic services, refactoring for, 164-166, 237-273
- DialogService, 238-251
- LogWriterService, 251-257
- MessageAggregator, 265-271
- ResourceProvider, 260-265
- SessionInformationService, 257-260
- static class conversion, 271-273
- Bootstrapper, 100-103, 213, 223
  - alternative configurations, 232-236
  - creating Winforms BootStrapper, 223-226
  - creating WPF applications, 228-232
  - updating Winforms program class, 226-228
- build errors, 216
- build process, selecting, 159-162, 169-185
- build servers, 181
- business entities, 152
- business logic
  - interfaces in, 9-11
  - migrating to controllers, 322-325
  - refactoring affecting, 203-211
  - testing with repository pattern, 280
C

The C# Programming Language
(Hejlsberg), 79
catch block, 228
central architecture, supporting code
versus, 142
centralized source control systems
distributed systems versus, 173-176
explained, 170-171
process example, 173
check-in frequency in source
control, 175
choosing architectural patterns, 45
CI (continuous integration)
servers, 181
introducing to team, 161-162
circular references, 189
classes
coupling
explained, 12-14
service abstraction and, 15-17
interfaces, 6-11
in business logic, 9-11
creating, 6-7
dependency inversion, 79-82
external interfaces, questions
about, 141-142
extracting, 82
implementing, 7-9
test doubles and, 116
moving to projects, 202
Clone Compare tool, 351
clone location tool, 182
code analysis, 138-142
application controls versus form
controls questions, 142
basic architecture questions, 138-139
code structure questions, 139-140
database access questions, 140
data structure questions, 140
external interface questions, 141-142
code clones, comparing, 352
Code Clones option, 350
code conversion, 149-150
automated code conversion utilities,
150-151
Code Metrics report, 348
hard-to-maintain code, 348-349
code navigation tools, 184
code snippet tools, 183
code structure, questions about,
139-140
code structure analysis, 142-144
command dispatcher service, 303-313
communication, importance of, 200
communication management,
167-168
comparing code clones, 352
composability, 18-19
ConfigureContainer method, 228
conflicts, merge, 172
Constants projects, 194-195
constructors
moving initialization logic to,
204-205
removing access in entity classes,
210-211
consumer shared code, 173-174
continuous integration (CI)
servers, 181
introducing to team, 161-162
continuous test runner, 344-345
controllers, 28
form refactoring, 315-325
business logic migration, 322-325
introducing controller, 320-322
legacy approach to form creation, 316-319
view preparation, 319-320
Controllers project, 199
custom workflow management in defect tracking systems, 180
data access (Repository pattern), 36-37, 199, 275-296. See also Dependency Inversion Principle
detecting, 154-155
domain model
creating repository, 283-288
data validations in, 291-296
existing code conversion, 289-291
method reengineering, 288-289
test doubles and, 116-119
data access patterns, detecting, 154-155
data access technologies in initial review process, 152-155
database access, questions about, 140
databases, dependencies on, 119
data model, detecting, 152-154
Data Model project, 198-199
data structure questions, 140-141
Data Transfer Objects (DTO) projects, 195-196
data validations in domain model, 291-296
dead code, removing, 145
defects, fixing in legacy code, 203
defect tracking
introducing to team, 161
selecting systems, 179-181
dependencies, 14
on databases, 119
tight coupling, problems with, 67-74
dependency injection containers, 84-87
replacing registrations, 123
Dependency Inversion Principle, 67, 103-108
Abstract Factory Pattern, implementing, 74-79
dependency injection, 103-108
interfaces, 79-82
service location, 84
advantages for reengineering, 108-113
BootStrapper class, 100-103
dependency injection containers, 84-87
example of, 90-96
OnDemand service property, 96-99
ServiceLocator factory, 88-90
unit testing advantages, 99-100
unit tests, creating, 82-84
detecting exceptions, 58-61
development, managing parallel
development, 330-334
development tools, selecting, 159-162,
169-185
continuous integration (CI)
servers, 181
defect tracking systems, 179-181
source control, 169-173
third-party refactoring tools,
183-184
Visual Studio 2010 refactoring tools,
182-183
DialogService, 238-251
on-demand service properties, 241
unit testing, 242-251
writing, 238
dispatcher service, 303-313
Distributed Source Control (DSC)
systems
centralized systems versus, 173-176
explained, 170-171
process example, 171-173
DTO (Data Transfer Objects) projects,
195-196
duplicates, looking for, 348-350
E
edit-and-continue feature, 341-342
editing tools, 184
Enterprise Library versions, 214-215
entities, 152
entity classes, removing constructor
access, 210-211
Erl, Thomas, 3, 18
error logs (LogWriterService), 251-257
what not to test, 257
estimating time requirements for Dia-
logService refactoring, 250
evaluating hosting services, 176-179
EventAggregator, 265-271
EventArgs class, 195
exceptions, detecting, 58-61
existing code, converting to domain
model, 289-291
expectation management, 157-158
ExpectedException attribute, 58
external interfaces, questions about,
141-142
extracting interfaces, 82
F
factory classes
Abstract Factory Pattern,
implementing, 74-79
service location, 84
advantages for reengineering, 108-113
BootStrapper class, 100-103
dependency injection containers, 84-87
dependency injection versus, 104
example of, 90-96
OnDemand service property, 96-99
ServiceLocator factory, 88-90
unit testing advantages, 99-100
failing unit test projects, 51
fakes, writing unit tests for untestable
code, 346-348
Fakes mocking framework, 123
file system, organizing, 187-189
Find Matching Clones in Solution
tool, 182
Find References tool, 183
form controls, application controls
versus, 142
forms, 217
refactoring, 315-325
business logic migration, 322-325
introducing controller, 320-322
legacy approach to form creation,
316-319
view preparation, 319-320
foundation, establishing for
reengineering process, 163-164
Fowler, Martin, 162
frequency of check-in in source
control, 175
G
generic object management, 296-303
Git, 179
global variables
in initial review process, 145-149
SessionInformationService, 257-260
goals of software engineering
adding new features, xvi
injecting modern architecture,
xv-xvi
playing with agile approaches,
xvi-xvii
reducing cost, xvii-xviii
reducing risk, xvii
guard statements, replacing nested IF
statements with, 205-210
H
hard-to-maintain code (Code
Metrics), 348-349
Hayes, Dennis, 151
Hejlsberg, Anders, 79
high-level components, problems
with tight coupling, 67-74
hosting services, evaluating, 176-179
I-J
IBaseView, 217-218
IF statements, replacing with guard
statements, 205-210
images (ResourceProvider), 260-265
imitation classes
additional resources for
information, 133
explained, 115
interfaces and, 116
mocks, creating, 124-130
Repository Pattern and, 116-119
Shims, 130-133
stubs
creating, 119-123
mocks versus, 123-124
implementing interfaces, 7-9
for dependency inversion, 79-82
initialization logic, moving to constructors, 204-205
initial review process, 137
automated code conversion utilities, 150-151
code analysis, 138-142
application controls versus form controls questions, 142
basic architecture questions, 138-139
code structure questions, 139-140
database access questions, 140
data structure questions, 140-141
external interface questions, 141-142
code conversion, 149-150
code structure analysis, 142-144
data access technology usage, 152-155
dead code removal, 145
global variable usage, 145-149
language migration management, 144
integration testing
separating from unit testing, 201-202
unit testing versus, 61-62
interfaces, 6-11
in business logic, 9-11
creating, 6-7
external interfaces, questions about, 141-142
extracting, 82
implementing, 7-9
for dependency inversion, 79-82
test doubles and, 116
Interfaces project, 196-197
InternalsVisibleTo attribute, 62-65
Inversion of Control (Dependency Inversion Principle), 67, 103-108
Abstract Factory Pattern, implementing, 74-79
dependency injection, 103-108
interfaces, 79-82
service location, 84
advantages for reengineering, 108-113
BootStrapper class, 100-103
dependency injection containers, 84-87
example of, 90-96
OnDemand service property, 96-99
ServiceLocator factory, 88-90
unit testing advantages, 99-100
unit tests, creating, 82-84
Inversion of Control containers, 84-87
replacing registrations, 123

K-L
keywords, Assert, 54
uses of, 61
when not to use, 60
language migration management, 144
legacy code
converting, 149-150
automated code conversion utilities, 150-151
fixing defects in, 203
form creation, 316-319
wrapping, 149-150
legacy solution cleanup, 162-163, 187-211
business logic refactoring, 203-211
Constants project, 194-195
Controllers project, 199
Index

Domain Model projects, 198-199
DTO (Data Transfer Objects) projects, 195-196
file system organization, 187-189
Interfaces project, 196-197
MVVM structures versus MVC structures, 193-194
project categories, 190-191
project structure, 189-190
project types, 191-193
Repository project, 199
Services project, 197-198
structure refactoring, 200-203
Levinson, Jeff, 66
libraries, moving shortcuts to, 202-203
Libraries folder, 188
listings
An Abstracted Version of Tightly Coupled Classes, 15-17
Accessing Data with Legacy Software, 276-277
An Additional Repository Test, 128
Additional Unit Tests, 56-57
All Guard Statements in a Single Line, 209-210
An ApplicationSession Service, 258-259
Assigning Values Directly to Labels, 15
The Basic View and Interface Other Views Inherit, 219
A Bootstrapper Implementation for a Winforms Application, 224-225
Broken Code That Can Be Shelved, 335
The Code for Resource Provider, 260-263
Code to Fill the Business Entity, 153-154
A Complex Nested IF Statement, 206-207
Creating an Interface, 7, 339, 347
Creating a Repository Using a Domain Model, 286-287
Creating the Shell Form Controller and Interface, 220
A Default Test Class, 337
A Dependency Injection Example, 105-106
A DisplayForm Descendant, 14
An Example Class Using Another Class, 86-87
An Example Command for the Command Dispatcher, 305-306
Example of Abstract Factory Pattern, 75
An Example of an Application Domain Model, 284-285
An Example of Generic DTO, 196
An Example of How to Use a Validation Service in a Repository, 293-295
An Example of the Form Creation Logic Typical of Legacy Applications, 316-318
An Example of Tightly Coupled Classes, 12-13
An Example Showing Methods Do Not Need Implementation Logic, 9
A Failing Test, 51
A First Simple Test, 49-50
A Fixed Unit Under Test, 55-56
Fixing the MyMath Class, 341
A Flawed Test to Check
CreateDate, 132
A Global Variable Example, 146
The Impact of Refactoring
SecurityService, 72-74
Implementation of a Generic Object
Manager, 299-302
Implementation of the Command
Dispatcher Service, 307-308
Implementation of the
Command-Line Interpreter,
309-311
Implementing a Dummy Service
and a Real Service, 234-236
Implementing an Interface, 8
Implementing the Service Locator,
221-222
An InternalsVisibleTo Example,
63-64
Introducing a Controller, 320-321
Introducing Interfaces for the
Services, 80-81
Legacy Code Refactored to Use a
Repository, 278-280
Legacy Code to Send Coupons to
New Users, 285-286
The Legacy Form Refactored to Add
the IDataEntryForm Interface,
319-320
A Legacy LogWriterService
Example, 254-255
Legacy LogWriterService Refactored
for ServiceLocator, 255-256
The Legacy Method After
Refactoring to DialogService, 240
The Legacy Method After
Refactoring to DialogService with
the OnDemand Property, 241-245
The Legacy Method Before
Refactoring to DialogService,
239-240
A Live Service Locator and a Sample
Bootstrapper, 100-103
The Main Application Class for a
WPF Application, 232
The Main Program Class with
Bootstrapper Initialization,
227-228
A Merge Conflict Demonstration-
Edit 1, 331
A Merge Conflict Demonstration-
Edit 2, 331
A MessageAggregator Example,
266-268
A Method That Throws an
Exception, 58-59
A Method Under Test, 52-53
Migrated Code from Form to Form
Controller, 322-324
Moving Initialization Code Inside
the Constructor, 205
An MVC View Exposing Events to
Controller, 39-40
An MVP View Exposing Events to
Presenter, 41-42
OnDemand Service Properties in
Base Classes, 98-99
The OnDemand Service Property,
96-97
Providing the Foundation for the
Views, 218
Pseudo Code for Unit Testing, 48
A Reengineered Coupon Distributor
Using a Repository, 287-288
Refactored Code That Uses the Local Variable, 147
Refactored Update Global Variable Using an “out” Parameter, 148
Refactoring a Static Class, 272-273
Refactoring Object Factory to Return Interfaces, 81
Refactoring to Use the ObjectFactory, 76-79
Registering a Class with Unity, 85-86
Registering a Singleton, 95-96
A Repository Example, 116-118
A Repository Mock, 124-127
A Repository Stub, 119-122
A Sample Business Entity, 152
A Sample DialogService, 238-239
A Sample Domain Model with Validation Logic, 292-293
A Sample Error Logger, 252-254
A Sample Internal Method, 64-65
A Sample Sales Process Allowing for Refunds, 20-21
A Sample Sales Process with No Refunds, 19-20
A Sample Service Implementation, 24-25
A Sample StatelessSales Process Allowing for Refunds, 22-23
A Sample Unit Test, 53, 118
A Sample View Interface, 32-33
A Sample View Interface with View Model, 34-35
A Second Implementation of the Interface, 11
A Service Locator Example, 90-93
Setting Up Logic Outside the Constructor, 204
A Simple Class Needing Unit Testing, 337
A Simple Service Locator, 88-89
A Simplified IF Statement with Guards, 207-208
Square Method with Checked Keyword, 345
Testing a Method That Throws an Exception, 59-60
Testing SecurityService and Object Factory, 83-84
Tests for Legacy Code Refactored to Use a Repository, 280-283
Three-Level Dependency Injection, 107-108
Tightly Coupled Code, 104, 109-111
A Tightly Coupled SalesRegistrationService, 68-70
Unit Tests Adapted to Use Mock Dialog Service, 245-247
Updating the LastUpdatedDate, 346
Using an Alternate to SecurityService, 80
Using an Interface, 10
Using a Shim to Test CreateDate, 132-133
Using Federated Security with SecurityService, 71
Using Generic Verify Parameters, 129-130
Using RegisterInstance with Unity, 94
A View Model Using Repository, 36-37
A Weak Reference DTO and a Strong Reference DTO, 298-299
WPF Application Class, 229-231
Writing a Test for an Expected Exception, 343
LogWriterService, 251-257
what not to test, 257
loose coupling, 6, 12-14
low-level components, problems with
tight coupling, 67-74

M
Manager service, 198
managing
merge conflicts, 175-176
parallel development, 330-334
merge conflicts, 172
managing, 175-176
Merge Tool, 333
MessageAggregator, 265-271
MessageBox method, DialogService
and, 238
messages
publishing, 269
subscribing to, 269
method extraction tools, 182-183
method renaming tools, 182
methods
in interfaces, 10
reengineering to repository pattern, 288-289
in unit tests, InternalsVisibleTo
attribute, 62-65
Microsoft Moles, 346
Microsoft Team Foundation Server
(TFS), 177-179
Microsoft Unit Testing framework,
unit test projects
creating, 49-52
failing, 51
running, 50
writing unit tests, 52-58
Microsoft Validation Block, 291
mocking frameworks, 123
mocks
creating, 124-130
for static classes, 130-133
stubs versus, 123-124
model access, 30-31
model agnostic projects, 193
Model project, 28, 198-199
model-specific projects, 193
Model-View-Controller. See MVC
Model-View-Presenter. See MVP
Model-View-ViewModel. See MVVM
Moles mocking framework, 123, 346
Moq library, 123, 133
moving
classes to projects, 202
initialization logic to constructors,
204-205
shortcuts to libraries, 202-203
multiple classes, implementing interfaces, 10-11
multi-targeting, 193
MVC (Model-View-Controller), 27-29
form refactoring, 315
model access, 30
structures, MVVM structures
versus, 193-194
UI events, handling, 41-42
MVP (Model-View-Presenter), 27-29
model access, 30
UI events, handling, 38-39, 42-43
MVVM (Model-View-ViewModel),
27-29
form refactoring, 315
model access, 30
structures, MVC structures versus,
193-194
UI events, handling, 39
view models, 31-38
**N–O**

naming
- project categories, 191
- unit tests, 54
- nested IF statements, replacing with guard statements, 205-210

*.NET Developer’s Journal*, 151
- new pattern projects, 191

old pattern projects, 190
- old software, xiv, xv

OnDemand service property, 96-99, 241
- organizing
  - file system, 187-189
  - project structure, 189-190

Osherove, Roy, 66
- outside DLLs, 188

**P**
- parallel development, managing, 330-334

patterns
- choosing, 45
- how they work, 43-44
- model access, 30-31

MVC (Model-View-Controller), 27-29
- form refactoring, 315
- model access, 30
- structures, MVC structures versus, 193-194
- UI events, handling, 39
- view models, 31-38
- overview, 28-29

UI events, 38-43
- view models, 31-38

planning
- initial review process, 137
- automated code conversion utilities, 150-151
- code analysis, 138-142
- code conversion, 149-150
- code structure analysis, 142-144
- data access technology usage, 152-155
- dead code removal, 145
- global variable usage, 145-149
- language migration management, 144

reengineering process
- communication and training management, 167-168
- development tool and build process identification, 159-162, 169-185
- expectation management, 157-158
- foundation establishment, 163-164
- legacy solution cleanup, 162-163, 187-211
- progress reports to stakeholders, 166-167
- refactoring for advanced services, 166, 275-314
refactoring for basic services, 164-166, 237-273
team creation, 158
preparing views in form refactoring, 319-320
presenters, 28
Presenters project, 199
Prism, 214-215
product backlogs in agile development, 180
progress reports to stakeholders, 166-167
project categories in legacy solution cleanup, 190-191
projects
adding new, 213-214
application-agnostic projects, 192
Controllers project, 199
dead-end projects, 192
Domain Model projects, 198-199
DTO (Data Transfer Objects) projects, 195-196
Interfaces project, 196-197
model-agnostic projects, 193
model-specific projects, 193
moving classes to, 202
Repository project, 199
Services project, 197-198
UI-related projects, 192-193
project structure, organizing, 189-190
project types in legacy solution cleanup, 191-193
Provider service, 197
publishing messages, 269
Q
questions
application controls versus form controls questions, 142
basic architecture questions, 138-139
code structure questions, 139-140
database access questions, 140
data structure questions, 140-141
external interface questions, 141-142
R
reducing cost, xvii-xviii
reducing risk, xvii
reengineering process
form refactoring, 315-325
initial review, 137
automated code conversion utilities, 150-151
code analysis, 138-142
code conversion, 149-150
code structure analysis, 142-144
data access technology usage, 152-155
dead code removal, 145
global variable usage, 145-149
language migration management, 144
planning stage
communication and training management, 167-168
development tool and build process identification, 159-162, 169-185
expectation management, 157-158
foundation establishment, 163-164
legacy solution cleanup, 162-163, 187-211
progress reports to stakeholders, 166-167
refactoring for advanced services, 166, 275-314
refactoring for basic services, 164-166, 237-273
team creation, 158

refactoring for advanced services, 166, 275-314
command dispatcher service, 303-313
generic object management, 296-303
repository pattern, 275-296
for basic services, 164-166, 237-273
DialogService, 238-251
LogWriterService, 251-257
MessageAggregator, 265-271
ResourceProvider, 260-265
SessionInformationService, 257-260
static class conversion, 271-273
business logic, 203-211
forms, 315-325
business logic migration, 322-325
introducing controller, 320-322
legacy approach to form creation, 316-319
view preparation, 319-320
solution structure, 200-203
third-party refactoring tools, 183-184
Visual Studio 2010 refactoring tools, 182-183
references
strong object references, 297
weak object references, 297
RegionManager, 229
RegisterControlsAndControllers method, 226
registering services as singletons, 242
RegisterType method, 226
registrations, replacing, 123
removing
dead code, 145
dependency constructor access, 210-211
static variables, 257
Using clauses, 200-201
renaming tools, 182
replacing
nested IF statements, 205-210
registrations, 123
reporting management in defect tracking systems, 180-181
Repository pattern, 36-37, 199, 275-296. See also Dependency Inversion Principle
detecting, 154-155
domain model
creating repository, 283-288
data validations in, 291-296
existing code conversion, 289-291
method reengineering, 288-289
test doubles and, 116-119
Resolve Conflict window, 332
ResourceProvider, 260-265
resources for information
on test doubles, 133
unit testing, 65-66
reusable services, 18
reviewing changes to shared code, 174
rewriting versus reengineering, 144
running
unit test projects, 50
unit tests, 339-341
S
seams, 149
separating unit tests and integration
tests, 201-202
service abstraction, 15-18
service autonomy, 18-19
service composability, 18-19
service contracts, standardized, 6-11
service location, 84
  BootStrapper class, 100-103
dependency injection containers, 84-87
dependency injection versus, 104, 108-113
element of, 90-96
OnDemand service property, 96-99
ServiceLocator factory, 88-90
  unit testing advantages, 99-100
ServiceLocator, 88-90, 213
  creating, 220-223
service-oriented architecture. See SOA (service-oriented architecture)
Service-Oriented Architecture: Concepts, Technology, and Design (Erl), 18
services
  example, 24-26
  registering as singletons, 242
  reusable services, 18
types of, 4-6
Services project, 197-198
service statelessness, 19-24
SessionInformationService, 257-260
shared code
  consuming, 173-174
  reviewing changes, 174
shared new pattern projects, 191
shell controllers, adding, 220
shells, adapting, 216-217
  adding shell controllers, 220
  current shells, 218-219
IBaseView, 217-218
shelves, creating, 336
Shims mocking framework, 130-133
shortcuts, moving to libraries, 202-203
singletons, 95
  registering services as, 242
  static class conversion, 271-273
SOA (service-oriented architecture), 3-4
coupling, 12-14
reusable services, 18
service abstraction, 15-18
service autonomy, 18-19
service composability, 18-19
service example, 24-26
service statelessness, 19-24
standardized service contracts, 6-11
types of services, 4-6
software aging, xiv-xv
software engineering, goals of, xv
  adding new features, xvi
  injecting modern architecture, xv-xvi
  playing with agile approaches, xvi-xvii
  reducing cost, xvii-xviii
  reducing risk, xvii
software reengineering, xiii
Software Testing with Visual Studio 2010 (Levinson), 66
solution structure, refactoring, 200-203
source control, 169-173
centralized systems
  distributed systems versus, 173-176
process example, 173
DLLs in, 188
DSC (Distributed Source Control)
systems, process example,
171-173
hosting services, evaluating, 176-179
introducing to team, 160-161
merge conflicts, 172
types of, 170-171
Visual Studio 2012, 327-329
specific new pattern projects, 191
stakeholders, progress reports to,
166-167
standardized service contracts, 6-11
statelessness, 19-24
static classes
converting, 271-273
factory classes as, 75
mocks for, 130-133
static variables, removing, 257
strong object references, 297
stubs
creating, 119-123
mocks versus, 123-124
subscribing to messages, 269
supporting code, core code
versus, 142
SVN (Apache Subversion), 177

T

TDD (Test Driven Development), 65
team
CI server (continuous integration
server), introducing, 161-162
creating, 158
defect tracking, introducing, 161
source control, introducing, 160-161
Team Foundation Server (TFS),
177-179
TestClass attribute, 54
test doubles
additional resources for
information, 133
explained, 115
interfaces and, 116
mocks, creating, 124-130
Repository Pattern and, 116-119
Shims, 130-133
stubs
creating, 119-123
mocks versus, 123-124
Test Driven Development (TDD), 65
testing
business logic with repository
pattern, 280
continuous test runner, 344-345
test doubles
additional resources for
information, 133
creating mocks, 124-130
creating stubs, 119-123
explained, 115
interfaces and, 116
mocks versus stubs, 123-124
Repository Pattern and, 116-119
Shims, 130-133
unit testing, 47, 337-338
additional resources for
information, 65-66
Assert keyword, uses of, 61
creating unit tests for dependency
inversion, 82-84
DialogService, 242-251
element, 48
exceptions, detecting, 58-61
integration testing versus, 61-62
InternalsVisibleTo attribute, 62-65
introducing, 165
naming unit tests, 54
running unit tests, 339-341
separating from integration tests, 201-202
service location and, 99-100
TDD (Test Driven Development), 65
using fakes to write unit tests for untestable code, 346-348
what not to test, 257
writing tests, 52-58
unit test projects
creating, 49-52
failing, 51
running, 50
TestInitialize method, 130
TestMethod attribute, 54
TFS (Microsoft Team Foundation Server), 177-179
third-party refactoring tools, 183-184
tight coupling, 12-14
problems with, 67-74
service abstraction and, 15-17
time-boxes for code structure analysis, 143
time requirements, estimating for DialogService refactoring, 250
TortoiseSVN, 177
Torvalds, Linus, 179
training management, 167-168
Try...Catch block, 228

U
UI events, handling, 38-43
UI-related projects, 192-193
Ulrich, W. M., xviii
unit testing, 47, 337-338
additional resources for information, 65-66
Assert keyword, uses of, 61
creating unit tests for dependency inversion, 82-84
DialogService, 242-251
element, 48
exceptions, detecting, 58-61
integration testing versus, 61-62
InternalsVisibleTo attribute, 62-65
introducing, 165
naming unit tests, 54
running unit tests, 339-341
separating from integration tests, 201-202
service location and, 99-100
TDD (Test Driven Development), 65
test doubles
additional resources for information, 133
creating mocks, 124-130
creating stubs, 119-123
explained, 115
interfaces and, 116
mocks versus stubs, 123-124
Repository Pattern and, 116-119
Shims, 130-133
unit test projects
creating, 49-52
failing, 51
running, 50
using fakes to write unit tests for untestable code, 346-348
what not to test, 257
writing tests, 52-58
unit test methods, writing, 338-339
unit test projects
creating, 49-52
failing, 51
running, 50
Unity dependency injection
  container, 85, 214-215
  replacing registrations, 123
untestable code, using fakes to write
  unit tests, 346-348
updating Winforms program class,
  226-228
Using clauses, removing, 200-201

V
Validate method in DTO projects, 195
validation in domain model, 291-296
variable renaming tools, 182
variables
  global variables
    in initial review process, 145-149
    SessionInformationService,
      257-260
    in interfaces, 10
VB6, converting to VB.NET, 151
VB.NET, converting VB6 to, 151
View, 28
view models, 28, 31-38. See
  also controllers
ViewModels project, 199
views, 217
  preparing in form refactoring,
    319-320
Visual Studio 2010 refactoring tools,
  182-183
Visual Studio 2012
  edit-and-continue feature, 341-342
  making changes in isolation,
    334-336
  source control, 327-329
  workspaces, creating new, 331

W–Z
weak object references, 297
Winforms BootStrapper, creating,
  223-226
Winforms program class, updating,
  226-228
workflows, custom workflow
  management in defect tracking
    systems, 180
workspaces, 328
  creating new, 331
WPF applications, creating with
  Bootstrapper, 228-232
wrappers, 149-150
writing
  DialogService, 238
  unit test methods, 338-339
  unit tests, 52-58