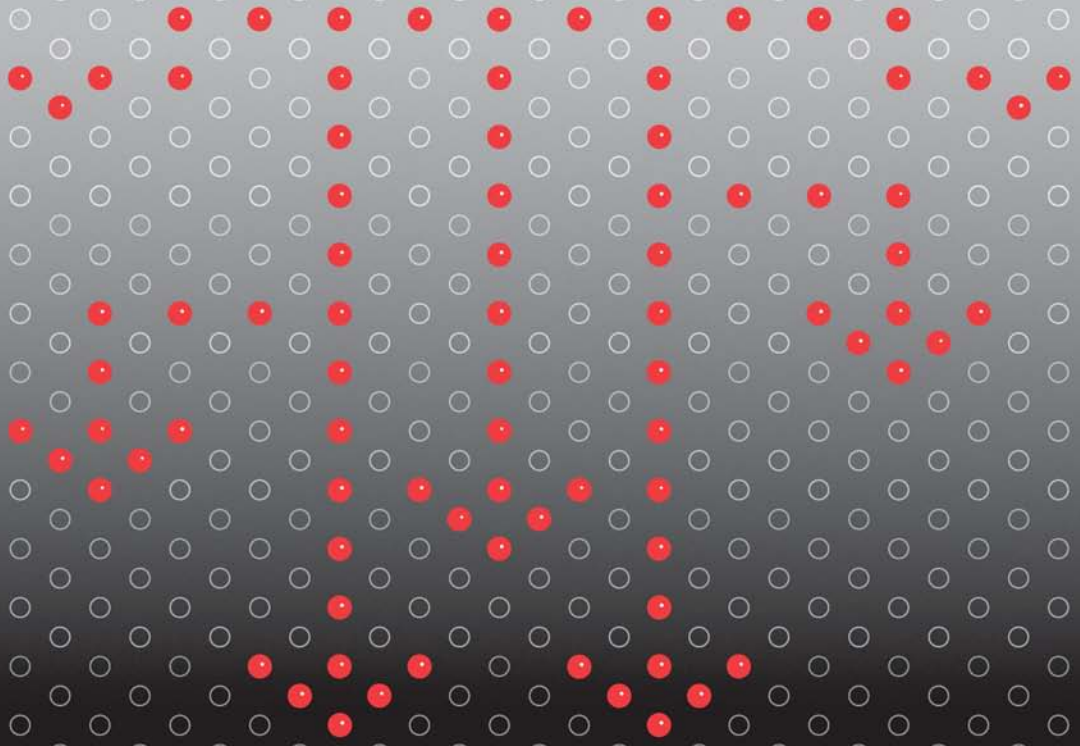
Microsoft®

MICROSOFT PROFESSIONAL



Test-Driven Development in Microsoft® .NET

James W. Newkirk
Alexei A. Vorontsov

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data pending.

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWE 8 7 6 5 4 3

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/learning/. Send comments to *mspinput@microsoft.com*.

Acquisitions Editors: Linda Engelman and Robin Van Steenburgh
Project Editors: Devon Musgrave and Kathleen Atkins
Indexer: Virginia Bess

Body Part No. X10-25670

This book is dedicated to my father,
William A. Newkirk

Practical wisdom is only to be learned in the school of experience. Precepts and instruction are useful so far as they go, but, without the discipline of real life, they remain of the nature of theory only — Samuel Smiles

JWN

This book is dedicated to my mother,
Larisa L. Vorontsova

A thousand-mile journey begins with the first step and can only be taken one step at a time. — An old saying

AAV

Contents at a Glance

Contents

Foreword

I enjoyed reading this book because it stretches the boundaries of Test-Driven Development (TDD). My original TDD book demonstrated TDD in an ideal situation, in which the programmer is just typing in code and doesn't have to worry about external systems or user interfaces. After you get into the messy realities of widgets and databases, you need new techniques to continue practicing TDD and reaping its benefits, among which is confidence in cleaner code written faster.

With this book, the pieces missing from my book are included. If you want to test drive code that includes a Web interface and a database, you will learn how to do that in these pages. Even if you aren't using the Microsoft technology, you will find ideas you can carry to your application server or database.

The strength of this book is its concreteness. The extensive examples show you exactly how expert programmers use test-driven development with realistic tasks. Following the examples will show you the techniques used and, more important, the flow between the techniques. Technique can be learned from a book, but to understand the rhythm of development, you usually need to sit down with a programmer who understands it. As you read, paying careful attention to the way the techniques fit together in this book will teach you lessons about the rhythm of programming.

I think TDD is a really valuable tool. It's inexpensive, it's easy to adopt, and it brings immediate improvement. TDD has led to fewer defects, less debugging, more confidence, better design, and higher productivity in my programming practice. More important, I sleep better at night knowing that my code works in every circumstance I can think of, and I can prove it at the push of a button. This book gives you the practical advice you need to gain the benefits of TDD.

Kent Beck

Acknowledgments

We would like to thank our technical reviewers, Martin Fowler, Lee Holmes, and Eric Gunnerson. The feedback and guidance that they provided during the writing process was invaluable. In addition to the technical reviews, we also received much needed feedback and criticism from the following individuals: Charlie Poole, Paul Karsten, Peter Provost, Gregor Hohpe, Dragos Manolescu, Michael Two, Kent Beck, Ron Jeffries, Jonathan Wanagel, Scott Densmore, Naveen Yajaman, David Astels, Ward Cunningham, Benjamin Mitchell, Chris Colleran, David Trowbridge, Srinath Vasireddy, and Andrew Slocum. Their input has greatly influenced the content of the book. It is a pleasure for us to acknowledge their contributions and express our appreciation for their efforts. We would also like to thank the following people at Microsoft Press: Linda Engelman for her help getting us started and everything she did to get the book completed as soon as possible; Robin Van Steenburgh for taking over Linda's big job; Devon Musgrave for his advice on the first draft; Kathleen Atkins for her help in getting the book completed; and Nancy Sixsmith for converting what we wrote into English.

Every book is an activity that always takes more time than you think. I (James) need to thank my wife, Beth, and my children, Erin and Grant, for allowing me the time that I needed to work on the book. In fact, I owe them for all the nights and weekends that they have given up while I worked on my latest "scheme." Thank you. In addition, I would like to thank my coauthor Alexei. I thoroughly enjoyed the many hours we worked together trying to cobble together the thoughts and ideas first into a sample program and then into the text that became this book.

Writing this book took a great deal more time than can be explained, justified, or even be considered reasonable, but, without a doubt, it has been the most rewarding experience for me. I (Alexei) owe James a great deal of gratitude for giving me this opportunity. I have learned much in the process of working on the book. I would also like to thank my mentor and manager, Regan Stern, for recognizing the importance of my working on this book and supporting me in this effort.

Introduction

Many people think that Test-Driven Development (TDD) is all about testing software. In fact, test-driven development's main goal is not testing software, but aiding the programmer and customer during the development process with unambiguous requirements. The requirements are expressed in the form of tests, which are a support mechanism (scaffolding, you might say) that stands firmly under the participants as they undertake the hazards of software development. However, that is not the only purpose of testing. As you will see, the tests, once written, are valuable resources in their own right.

What Are the Benefits of Using Tests?

It is important during development that problems are discovered early and corrected when they are found. Often, the biggest problems occur when there is a misunderstanding of a requirement between the consumers of the software (customers) and the producers (programmers). These types of problems could be avoided if there were a way to specify these requirements unambiguously before development begins. Enter tests. The tests specify requirements in a way that does not require human interpretation to indicate success or failure. If there is a sufficient number of tests and they are present prior to development, simply running the tests and indicating success or failure helps solve the old problem of software development, "Are We Done?" The answer is no longer an interpretation; the code either passes all the tests or it does not. After it passes, it's done.

Solving this problem alone may be justification enough, but is there more? The tests that are written during development can be run and enhanced by Quality Assurance (QA) with tests of their own. Due to the code being written with testing as a primary motivation, the resulting code should be easier to test. Having a base of existing tests and code that is easier to test should allow QA to shift from a reactive mode into a more proactive mode.

The tests themselves are useful not only in the initial development of the software; if they are maintained along with the production code, they can be used in the ongoing development of the software. For example, if a problem is discovered in the production code, the first step should be to write a test to clearly identify the problem and then, after you have a failing test, correct the

problem. This new test specifies a scenario that was not identified during the prior development. If you do this consistently, the tests will evolve into how the program is used in real life, which increases their value exponentially. When adding new features, you could run this suite of tests to ensure that the new code does not break any of the existing tests. If the test coverage is sufficient; running the tests and getting a successful result should reduce your fear of moving forward. Fear of breaking existing functionality can cause you to become overly cautious, which slows you down. Think of the tests as a way of covering your back.

An Example

Let's look at an example to demonstrate how tests can describe a requirement more clearly than words can. Consider the following description of a Stack. "A *Stack* is a data structure in which you can access only the item at the top. With a computer, *Stack* just like a stack of dishes—you add items to the top and remove them from the top" (*http://www.developersdomain.com/vb/articles/stack.htm*). This is not a bad description, but it does not specify method names and it uses an analogy that might not resonate with people. In short, it leaves a great deal open to interpretation, and you would get many implementations that could satisfy this definition.

Now look at a test that specifies the same thing:

```
[Test]
public void PushPop()
{
   string name = "Name";
   Stack stack = new Stack();
   stack.Push(name);
   Assert.AreEqual(name, stack.Pop());
}
```

This code specifies the names of methods, how they are called, and what they should return. It also specifies a sequence that yields a successful result. Finally, the test is executable, meaning that you can run it on the production code, and it will inform you if your implementation passes the test. The only thing that is open to interpretation is how you should implement the *Stack*, which is exactly what you want if you are a programmer. If your job was to implement a *Stack*, would you rather have your specification described as a series of tests or as a written specification?

Organization

This book is organized into two sections, followed by three appendixes.

- **Part I: Test-Driven Development Overview** This section describes the concepts of test-driven development. It begins with Kent Beck's rules, provides some additional detail about how to use and apply these rules, defines terminology that we use throughout the book, and defines a process for doing test-driven development. In addition to the definitions, we also demonstrate how to apply them by example. The focus in these early chapters is on completeness and following the principles and practices as written.
- **Part II: The Test-Driven Development Example** This section demonstrates how to do Test-Driven Development on a realistic *n*-tier application. The application, a media library, is specified in Chapter 4, "The Media Library Example." As well as implementing the expected functionality, we also investigate important real-world application areas that are typically avoided in sample applications. For example, we demonstrate the use of TDD with concepts such as exception handling and database connectivity. By the end of the sample, you'll have a good grounding in the techniques needed to use TDD in your own enterprise projects.
- **Appendix A: NUnit Primer** This appendix contains an introduction to the tool, NUnit.
- **Appendix B: Transactions in ADO.NET** This appendix provides an overview of transaction support in the .NET Framework.
- **Appendix C: Bibliography** The bibliography lists the works by other people that we have used ourselves and referred readers to throughout this book.

How to Use This Book

This book is written primarily for experienced programmers. You will get more value from this book if you are familiar with C# syntax and understand object-oriented programming. However, even if your primary development language is not C#, you should be able to port the example to other .NET languages, such as Microsoft Visual Basic .NET. The more complicated concepts do have overview material and pointers to additional sources of information.

If You Have Never Used NUnit Before

Read Chapter 1, "Test-Driven Development Practices." Then read Appendix A, "NUnit Primer," which describes the tool that is used for technology facing or programmer tests in the text. Then you can proceed with the rest of the content.

If You Are a Manager or Business Analyst

Read Chapter 1, which introduces the concepts and the process. Then read Chapter 7, "Customer Tests: Completing the First Feature," in which we discuss ways to use tests without having to write them in C#. We use a tool named FIT (*http://fit.c2.com*) to implement the business-facing or customer tests.

Small Steps—A Personal Story

Sometimes, people ask me (James) how I got started doing test-driven development. I want to relate this story because as a result of this experience, I finally believed that a series of small steps, verified each time by tests, could actually lead to a better solution. Up until this point, I knew the rules but not how to apply them.

It was December of 1999. I was at Object Mentor, and we were in the midst of the first XP immersion class. Kent Beck, Ron Jeffries, Martin Fowler, Robert C. Martin, Michael Hill, Fred George, Alan Francis, and others were my companions. Needless to say, it was an incredible week, not so much from the perspective of a class, but from being around such an awesome array of talent all focused on this thing called Extreme Programming. Besides participating in the class, I was working on a new Java class that I would be presenting the following January. I was trying to incorporate aspects of refactoring and test-first programming using JUnit (*http://www.junit.org*) into the class. My thought was to write an awful program and then use it to teach the concepts of refactoring. (That same awful program, implemented in C#, is the basis for Chapter 3, "Refactoring—By Example.")

I was working alone, and the staging of the example was not working well because it turns out that my refactoring steps were much too large. Kent came over and asked what I was doing after noticing me working on code by myself; I told him that I was trying to work out an example of refactoring for my upcoming class. After looking at what I did, he told me he thought I should start over. Instead of walking away, he offered to sit down and help me. During the next hour or so, the whole idea of the small incremental changes leading to a better solution became a reality. This awful code was transformed into something that was very clear and easily understandable.

It is only after I spent that time working with Kent directly that I began to understand just how small the steps were that Kent, Ron, and Martin were talking about. In fact, we thought that this in itself would be a useful activity for the whole class to see the following day. So, Robert Martin went home that night and constructed some UML diagrams around the code and came up with a slightly different implementation of the same algorithm that Kent and I refactored in front of the class the next day. For a couple of hours, we walked step-by-step through the code—making the smallest of changes and then running the tests to make sure that we did not break anything. When we were finished, someone said that we had made 40 separate changes to the code. The code was so much clearer that it was remarkable. Alexei and I have used the same problem that taught me so much during the class as the sample in Chapter 3 so that you can also benefit from that experience.

Companion Web Site

Many of the code samples in this book were too long to print without interruption by explanatory text. If you prefer to see the complete code samples from the early chapters and the sample application in its entirety, you can go to *http://workspaces.gotdotnet.com/tdd*.

3

Refactoring—By Example

In Chapters 1 and 2, we briefly touched on the subject of refactoring. This chapter gives a detailed treatment of this topic because refactoring is one of the fundamental aspects of test-driven development and a very useful practice in its own right.

Refactoring is an activity aimed at improving the internal structure of existing code without making externally visible changes to the functionality. Why would such changes be useful? (After all, there is an age-old engineering adage: "If it ain't broke, don't fix it.") Are we suggesting fixing a problem that does not exist? Is refactoring just another way to waste your time and money? The simple answer is no.

> **Note** Refactoring is a long-term, cost-efficient, and responsible approach to software ownership.

We argue that refactoring is the way to make your long-term software ownership less painful. Through refactoring, design intent becomes clearer as the code evolves. Without refactoring, the code's clarity will degrade over time, eventually becoming unintelligible.

Let's look at some code to clarify the point. We will demonstrate the basic ideas behind refactoring on a simple piece of code that is in need of some maintenance.

> **More Info** For additional reading on this topic, read Martin Fowler's book: *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999). This book is the source of the refactoring names that are used in this chapter. As a side note, the examples in Martin's book are in Java but are straightforward enough to follow if you know C#.

The Sieve

The code we will refactor implements an algorithm to generate small prime numbers (say up to 10,000,000). The algorithm is called the *Sieve of Eratosthenes*. Make a list of all the integers less than or equal to *n* (and greater than one). Strike out the multiples of all primes less than or equal to the square root of *n*; the numbers that are left are the primes (*http://primes.utm.edu/glossary/page.php?sort=SieveOfEratosthenes*).

The existing implementation is shown here:

```
using System;
using System.Collections;

public class Primes
{
    public static ArrayList Generate(int maxValue)
    {
        ArrayList result = new ArrayList();

        int[] primes = GenerateArray(maxValue);
        for(int i = 0; i < primes.Length; ++i)
            result.Add(primes[i]);

        return result;
    }

    [Obsolete("This method is obsolete, use Generate instead")]
    public static int[] GenerateArray(int maxValue)
    {
        if(maxValue >= 2)
        {
            // declarations
            int s = maxValue + 1; // size of array
            bool[] f = new bool[s];
            int i;
```

```
            // initialize the array to true
            for(i=0; i<s; i++)
                f[i] = true;

            // get rid of known nonprimes
            f[0] = f[1] = false;

            // sieve
            int j;
            for(i=2; i<Math.Sqrt(s)+1; i++)
            {
                for(j=2*i; j<s; j+=i)
                    f[j] = false; // multiple is not prime
            }

            // how many primes are there?
            int count = 0;
            for(i=0; i<s; i++)
                if(f[i]) // if prime
                    count++; // bump count

            int[] primes = new int[count];

            // move the primes into the result
            for(i=0, j=0; i<s; i++)
            {
                if(f[i]) // if prime
                    primes[j++] = i;
            }

            return primes;
        } // maxValue >= 2
        else
            return new int[0]; // return null array
    }
}
```

As you can see from the code, there are two methods defined to generate prime numbers. The first method, *Generate*, returns the prime numbers in an *ArrayList*. The second method, *GenerateArray*, was written to return an array of integers. The *GenerateArray* method is also marked with the *Obsolete* attribute, which is usually an indicator that the code will be removed when possible. It turns out that today is the day we will remove this function because the *GenerateArray* method is no longer called by the application code but it is still called by the *Generate* method. It looks like we won't be able to just delete it. Luckily, the code has a set of tests written using NUnit for it:

```
using System;
using System.Collections;
using NUnit.Framework;

[TestFixture]
public class PrimesFixture
{
    private int[] knownPrimes = new int[]
    { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };

    [Test]
    public void Zero()
    {
        int[] primes = Primes.GenerateArray(0);
        Assert.AreEqual(0, primes.Length);
    }

    [Test]
    public void ListZero()
    {
        ArrayList primes = Primes.Generate(0);
        Assert.AreEqual(0, primes.Count);
    }

    [Test]
    public void Single()
    {
        int[] primes = Primes.GenerateArray(2);
        Assert.AreEqual(1, primes.Length);
        Assert.AreEqual(2, primes[0]);
    }

    [Test]
    public void ListSingle()
    {
        ArrayList primes = Primes.Generate(2);
        Assert.AreEqual(1, primes.Count);
        Assert.IsTrue(primes.Contains(2));
    }

    [Test]
    public void Prime()
    {
        int[] centArray = Primes.GenerateArray(100);
        Assert.AreEqual(25, centArray.Length);
        Assert.AreEqual(97, centArray[24]);
    }
```

```
[Test]
public void ListPrime()
{
    ArrayList centList = Primes.Generate(100);
    Assert.AreEqual(25, centList.Count);
    Assert.AreEqual(97, centList[24]);
}

[Test]
public void Basic()
{
    int[] primes =
        Primes.GenerateArray(knownPrimes[knownPrimes.Length-1]);
    Assert.AreEqual(knownPrimes.Length, primes.Length);

    int i = 0;
    foreach(int prime in primes)
        Assert.AreEqual(knownPrimes[i++], prime);
}

[Test]
public void ListBasic()
{
    ArrayList primes =
        Primes.Generate(knownPrimes[knownPrimes.Length-1]);
    Assert.AreEqual(knownPrimes.Length, primes.Count);

    int i = 0;
    foreach(int prime in primes)
        Assert.AreEqual(knownPrimes[i++], prime);
}


[Test]
public void Lots()
{
    int bound = 10101;
    int[] primes = Primes.GenerateArray(bound);

    foreach(int prime in primes)
        Assert.IsTrue(IsPrime(prime), "is prime");

    foreach(int prime in primes)
    {
        if(IsPrime(prime))
            Assert.IsTrue(Contains(prime, primes),
                "contains primes");
        else
```

```
                Assert.IsFalse(Contains(prime, primes),
                    "doesn't contain composites");
            }
        }

        [Test]
        public void ListLots()
        {
            int bound = 10101;
            ArrayList primes = Primes.Generate(bound);
            foreach(int prime in primes)
                Assert.IsTrue(IsPrime(prime), "is prime");

            foreach(int prime in primes)
            {
                if(IsPrime(prime))
                    Assert.IsTrue(primes.Contains(prime),
                        "contains primes");
                else
                    Assert.IsFalse(primes.Contains(prime),
                        "doesn't contain composites");
            }
        }


        private static bool IsPrime(int n)
        {
            if(n < 2) return false;

            bool result = true;
            double x = Math.Sqrt(n);
            int i = 2;
            while(result && i <= x)
            {
                result = (0 != n % i);
                i += 1;
            }

            return result;
        }


        private static bool Contains(int value, int[] primes)
        {
                 return (Array.IndexOf(primes, value) != -1);
        }
    }
```

Before Refactoring the Code: Make Sure It All Works

It is important to remember that refactoring has to be done in conjunction with running tests for the code being refactored. After all, refactoring is not supposed to change the externally observable functionality of the code being refactored. The tests are the tools needed to verify such functionality. So the first step of the refactoring process is to run the tests before you make any code changes.

Let's run the tests. All of them pass, so we can begin from a known good state.

Refactoring Cycle

The cycle we will follow is straightforward: Identify a problem, select a refactoring to address the problem, apply the refactoring by making the appropriate code change; compile and run the tests; repeat. The emphasis is on the code changes being very small—and running the tests. Why small changes? We transition the system from a known good state to the next desirable state.

Think of it as climbing a wall. If the wall is high, you might break your neck attempting to climb it, but you could use a ladder to assist you. With a ladder in place if you feel tired, you can just stop and rest. The tests are your ladder—they are both your safety net and a climbing tool. So, before you start climbing, what should you do? Do yourself a favor: Make sure that your ladder is not broken. This brings us to the following rule for refactoring:

> **Important** As you refactor your code, make sure that the tests are up-to-date. If you need to change them to reflect the changing requirements, do it first.

In short, maintain your ladder. Let's take a look at the tests.

Refactoring 0: Remove Unneeded Code

There are five test methods for the *array*-based version and five test methods for the *ArrayList* version. Because the *GenerateArray* method is being removed, it appears that we can remove the tests for that method. We can do this safely because we are not losing any test coverage by removing the *array*-based tests. The *ArrayList*-based tests are exact duplicates in terms of what is being tested.

After the *array*-based tests are removed, the following tests remain:

- *ListZero*
- *ListSingle*
- *ListPrime*
- *ListBasic*
- *ListLots*

We can also get rid of the utility method *Contains* because it was used only by the *array*-based tests. After we finish removing the code, we compile and run the tests. The test method count drops to five and we have a green bar, so it is time to move on.

Refactoring 1: Rename Method

The next refactoring is still in the test code. After we remove the *array*-based tests, there is no need to preface each method with the word *List*. We need to implement the "Rename method" of refactoring. The reasoning is that you should call an apple an apple; no need to call it a "green apple" unless the greenness of the apple is of the essence. Meaningful method names are important for code readability and in turn its overall maintainability. In short, method names should convey their intentions.

Here is the test code after each method has been renamed; the contents of the methods have not changed, so they are not shown here:

```
using System;
using System.Collections;
using NUnit.Framework;

[TestFixture]
public class PrimesFixture
{
    private int[] knownPrimes = new int[]
    { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };

    [Test]
    public void Zero()
    {
      // …
    }

    [Test]
    public void Single()
    {
      // …
    }
```

```
    [Test]
    public void Prime()
    {
      // …
    }

    [Test]
    public void Basic()
    {
      // …
    }


    [Test]
    public void Lots()
    {
      // …
    }


    private static bool IsPrime(int n)
    {
      // …
    }
}
```

In this case, the renaming of the methods is straightforward. There should be no code calling the test methods. The more general case is a bit more complicated because you might have to change the callers of the method being renamed. Modern development environments make it easier to accomplish this task and pretty much take care of the process of finding and replacing the method names. If you are using a simple text editor, you might let your compiler tell you which classes you need to fix (which is crude, but it works). As always, after we make the changes, we compile and run the tests. The tests passed, so it's time to continue.

Refactoring 2: Add a Test

The *Single* test method verifies that 2 is a prime number; the *Zero* test method verifies that 0 is not a prime number. What about the number 1? We should also have a test that ensures that 1 is not a prime number.

We add a new test named *ZeroOne* and rename the *Single* method to be *ZeroTwo* to reflect the range of values being tested:

```
[Test]
public void ZeroOne()
{
    ArrayList primes = Primes.Generate(1);
    Assert.AreEqual(0, primes.Count);
}

[Test]
public void ZeroTwo()
{
    ArrayList primes = Primes.Generate(2);
    Assert.AreEqual(1, primes.Count);
    Assert.IsTrue(primes.Contains(2));
}
```

Now we have three methods that test the special cases of 0, 1, and 2. They look very similar, but it is not apparent how to factor out any commonality. When we compile and run all the tests, they succeed. We now have six tests.

When Are We Finished?

At every step of the way, an important question to ask is "Am I finished?" By the very nature of moving from a known good state to the next state, it is possible to stop at any time. What is there left to do? Why didn't we stop after removing the *array*-based tests? Because we immediately saw what we could not possibly see before: the method names could be improved. Each simple refactoring we implement opens up opportunities for further refactorings to make the code communicate its intentions more clearly.

What makes the process interesting is that it is a process of discovery. We probably don't know what refactoring we'll implement next. We also don't create a grand plan of 1001 refactorings that are needed to make this code better. We let the code itself drive the process. The code tells us which refactoring is needed at the appropriate time, and it evolves gradually into the shape it wants to take over time. The answer for now is that we are not done. We have not removed the *array*-based implementation. However, we are done with refactoring the test code.

Refactoring 3: Hide Method

Let's look at the code that generates the prime numbers. The *GenerateArray* method is used internally by the *Generate* method. There is no need to keep it public any more. We'll implement the "Hide method" refactoring, which is quite simple. In C#, it is accomplished by changing the visibility of the method from *public* to *private*. The following code

```
public static int[] GenerateArray(int maxValue)
```

now becomes

```
private static int[] GenerateArray(int maxValue)
```

Why did we do this refactoring? The less the code promises, the easier it is to deliver. The *GenerateArray* method now becomes an implementation detail. The code compiles and the tests pass, so let's move on to the next step.

Refactoring 4: Replace Nested Conditional with Guard Clauses

Large monitors with high resolutions allow you to see many more lines of code onscreen than you could a few years ago. But you won't make many friends if you continue to write (or tolerate) code like the *GenerateArray* method.

What is the biggest problem with the *GenerateArray* method? Let's distill it down to the essence:

```
if(maxValue >= 2)
{
      pages and pages of code that won't fit on your screen
      return primes;
} // maxValue >= 2
else
      return new int[0]; // return null array
```

The problem is that when you finally get to the *else* statement, the *if* statement has probably scrolled off the screen, so you do not have the context in which the statement is being executed. One way to correct this problem is to use the "Replace nested conditional with a guard clause" refactoring. Employing a guard clause at the beginning of the method dispenses with the bad input and focuses the method on processing the good input. Changing the method to use a guard clause looks like this:

```
if(maxValue < 2) return new int[0];

the rest of the code here.
```

Those of you who subscribe to one of the major tenets of structured programming (single entry point/single exit point) are probably jumping out of your chair. The reason this other approach is all right in this situation is because the guard clause identifies a rare situation that can be handled immediately. This frees up the rest of the code to handle the typical calling scenario without having to worry about the rare or invalid situations. In short, with the guard clause in place, the code is easier to read. After we insert the guard clause, the code compiles and the tests pass.

Refactoring 5: Inline Method

Now that the only code that calls *GenerateArray* is the *Generate* method, we can use the "Inline method" refactoring to put the method's body into the body of its caller and completely remove the method. This is not a license to create huge methods. If we intended to stop refactoring after inlining this method, we would argue to not inline the method.

The point that needs to be stressed is communication. If it makes sense to inline a method because it communicates the intent better than it did previously, you should do it. It also decreases the surface area of the code, which should improve its testability if you don't have huge methods. Because the *Generate* method returns an *ArrayList* and the *GenerateArray* method returns an *array*, we will need to slightly alter the guard clause introduced in the previous step to return an empty *ArrayList* instead of an empty *array*. Here is the *Generate* method after inlining the *GenerateArray* method (the modified guard clause is in boldface):

```
public static ArrayList Generate(int maxValue)
{
    ArrayList result = new ArrayList();
    if(maxValue < 2) return result;

    // declarations
    int s = maxValue + 1; // size of array
    bool[] f = new bool[s];
    int i;

    // initialize the array to true
    for(i=0; i<s; i++)
        f[i] = true;

    // get rid of known nonprimes
    f[0] = f[1] = false;

    // sieve
    int j;
    for(i=2; i<Math.Sqrt(s)+1; i++)
    {
        for(j=2*i; j<s; j+=i)
            f[j] = false; // multiple is not prime
    }

    // how many primes are there?
    int count = 0;
    for(i=0; i<s; i++)
        if(f[i]) // if prime
```

```
            count++; // bump count

        int[] primes = new int[count];

        // move the primes into the result
        for(i=0, j=0; i<s; i++)
        {
            if(f[i]) // if prime
                primes[j++] = i;
        }

        for(i = 0; i < primes.Length; ++i)
            result.Add(primes[i]);

        return result;
    }
```

This refactoring often requires more effort due to local variable name clashes. When performing this refactoring, you will find it useful to comment out the method that is being inlined instead of deleting it. After the code compiles and the tests pass, you can safely delete the commented-out code, which is useful to go back to in case your tests do not pass. You could also use your source-code control system to achieve the same benefit.

The code compiles, and the tests pass. The *GenerateArray* function has now been removed (or to be more exact, consumed, by the *Generate* method). Remember, this was the objective of the task. We could stop right now and be finished. However, we are still left with the legacy of the *array*-based implementation, which is filled with bad variable names and loops that iterate over the list of numbers many times. We need to do some more work to get this code in better shape.

Refactoring 6: Rename Variable

Looking at the code in the *Generate* method, we see several variables whose names do not communicate much about their intended uses, so we should give them more descriptive names. For example, what does the variable *f* mean? Does *f* indicate that the number is prime or not prime? Let's take a look at the following code snippet to demonstrate the point:

```
if(f[i]) // if prime
```

Instead of having comments in the code describing what the variable *f* means, it is better to give the variable a more descriptive name. In almost all cases in the existing program, every time the variable *f* is used there is an associated comment. Let's remove the need for the comment by providing a more

descriptive variable name. The name *isPrime* describes what the variable means in the code more clearly. After the name is changed, we can remove the comment because the variable name is descriptive enough:

```
public static ArrayList Generate(int maxValue)
{
    ArrayList result = new ArrayList();

    if(maxValue < 2) return result;

    // declarations
    int s = maxValue + 1; // size of array
    bool[] isPrime = new bool[s];
    int i;

    for(i=0; i<s; i++)
        isPrime[i] = true;

    isPrime[0] = isPrime[1] = false;

    // sieve
    int j;
    for(i=2; i<Math.Sqrt(s)+1; i++)
    {
        for(j=2*i; j<s; j+=i)
            isPrime[j] = false; // multiple is not prime
    }

    // how many primes are there?
    int count = 0;
    for(i=0; i<s; i++)
        if(isPrime[i])
            count++; // bump count

    int[] primes = new int[count];

    // move the primes into the result
    for(i=0, j=0; i<s; i++)
    {
        if(isPrime[i])
            primes[j++] = i;
    }

    for(i = 0; i < primes.Length; ++i)
        result.Add(primes[i]);

    return result;
}
```

The changes are made, the code compiles, and the tests pass. It does not look as if we are finished, however. The code still has a lot of the remnants of the *array*-based implementation and it still has many loops that seem as if they all iterate over the same elements.

Refactoring 7: Collapse Loops

Looking at the last few lines of the *Generate* method, you can see two loops doing almost entirely the same thing. Here is the existing code:

```
int[] primes = new int[count];

// move the primes into the result
for(i=0, j=0; i<s; i++)
{
    if(isPrime[i])
        primes[j++] = i;
}

for(i = 0; i < primes.Length; ++i)
    result.Add(primes[i]);
```

The first loop cycles through the *isPrime* array to create a new array named *primes*. The second loop cycles through the *primes* array to build the list. This is a remnant of the *array*-based implementation returning an *array* and the *ArrayList* function converting it into an *ArrayList*. Because we no longer return an array, we can do this without creating the *primes* array, as follows:

```
for(i = 0; i < s; ++i)
{
    if(isPrime[i])
        result.Add(i);
}
```

After this change is made, the code compiles and the test passes.

Refactoring 8: Remove Dead Code

The *array*-based legacy is almost gone. Because we no longer create the *primes* array, we no longer need the *count* variable because it was just used to size the *primes* array. Therefore, we can get rid of the *count* variable and the loop that calculates it. Let's move on.

Refactoring 9: Collapse Loops (Again)

Are we done? We could be, but it appears as if a few more changes could make the code a lot clearer, so let's continue for awhile longer.

Look at this loop:

```
for(i=2; i<Math.Sqrt(s)+1; i++)
{
   for(j=2*i; j<s; j+=i)
      isPrime[j] = false; // multiple is not prime
}
```

Can we make it better? The algorithm states that you have to remove multiples only if the number is a prime number, so the code is not as efficient as it could be. Try this:

```
for(i=2; i<Math.Sqrt(s)+1; i++)
{
   if(isPrime[i])
   {
      for(j=2*i; j<s; j+=i)
         isPrime[j] = false; // multiple is not prime
   }
}
```

We make the change, compile, and run the tests. They pass, so adding this did not have an impact on the functionality, and the code is closer to the intent of the algorithm.

We Can Do Some More…

Is the code faster? Probably, but because we do not have a performance test, we do not know the answer to that. However, after we make this change, the two loops at the bottom of the program look very similar; they have the same *if* statement in them. Perhaps we can collapse the two loops together.

Here's the existing code:

```
int j;
for(i = 2; i < Math.Sqrt(s)+1; i++)
{
    if(isPrime[i])
    {
        for(j=2*i; j<s; j+=i)
            isPrime[j] = false; // multiple is not prime
    }
}

for(i = 0; i < s; ++i)
{
```

```
    if(isPrime[i])
        result.Add(i);
}
```

The boundaries of the loops are different. The first loop iterates over the *isPrime* array, beginning at 2 and continuing to *Math.Sqrt(s) + 1*. The second loop iterates over the *isPrime* array, starting at 0 and continuing all the way to *s*.

Enough about symbols. Let's look at real numbers. If *s* were equal to 100, the first loop would execute 10 times, and the second loop would execute 100 times. It looks as if it would be simple to have the second loop start at 2 instead of 0. Let's make that change. All the tests pass, so it works and the lower boundary conditions are now the same.

Now what about the upper boundary? It looks as if we could change the first loop to continue all the way to *s*. This is clearly less efficient, but (as stated previously) it is hard to say whether that is a problem because the code does not have a performance test. Let's change the code to the following and see whether it works:

```
int j;
for(i = 2; i < s; i++)
{
    if(isPrime[i])
    {
        for(j=2*i; j<s; j+=i)
            isPrime[j] = false; // multiple is not prime
    }
}

for(i = 2; i < s; i++)
{
    if(isPrime[i])
        result.Add(i);
}
```

All the tests pass, and the loops have identical boundary conditions. It is clearer now, after looking at the code and knowing that the tests run successfully, that we can safely collapse the loops into a single loop.

```
int j;
for(i = 2; i < s; i++)
{
    if(isPrime[i])
    {
        result.Add(i);
        for(j=2*i; j<s; j+=i)
            isPrime[j] = false; // multiple is not prime
    }
}
```

That works—the tests passed. It is difficult to say that the code is less efficient because we did get rid of the second loop. And we removed a couple of other loops that were used in the *array*-based implementation, so it is possible that what we have now is more efficient than it used to be. We leave it up to you to verify whether the code performs worse now than it did before we started.

Refactoring 10: Reduce Local Variable Scope

Because of all the previous refactorings, the variable *j* is now used in only one loop. We can now change its scope by moving its declaration into the loop where it is used:

```
for(i = 2; i < s; i++)
{
    if(isPrime[i])
    {
        result.Add(i);
        for(int j = 2 * i; j < s; j += i)
            isPrime[j] = false; // multiple is not prime
    }
}
```

That works just fine, and the local variable *j*'s scope is diminished.

Refactoring 11: Replace Temp with Query

The next step is to replace the temporary variable *s* because it does not communicate what it actually means:

```
int s = maxValue + 1; // size of array
```

Instead of a temporary variable, we can replace the variable entirely by using the expression *isPrime.Length,* which communicates what we really mean and is already provided by the array implementation. The changes are in boldface as follows:

```
public static ArrayList Generate(int maxValue)
{
    ArrayList result = new ArrayList();

    if(maxValue < 2) return result;

    bool[] isPrime = new bool[maxValue+1];
    int i;

    for(i = 0; i < isPrime.Length; i++)
```

```
            isPrime[i] = true;

        isPrime[0] = isPrime[1] = false;

        // sieve
        for(i = 2; i < isPrime.Length; i++)
        {
            if(isPrime[i])
            {
                result.Add(i);
                for(int j = 2 * i; j < isPrime.Length; j += i)
                    isPrime[j] = false; // multiple is not prime
            }
        }

        return result;
    }
```

Refactoring 12: Remove Dead Code

There still is some code that is not used any more due to the collapse of loops done a few refactorings ago. Because the loop that does the sieve process starts at 2 and we load the list from within that loop, we no longer need to initialize 0 and 1 to *false* because they are never accessed. We can safely remove the following line:

```
isPrime[0] = isPrime[1] = false;
```

The tests pass when we compile and run them, so it was probably safe to assume that we could remove the line.

Refactoring 13: Extract Method

Even though the code has come a long way, there is still room for improvement, especially for making the code much more explicit about what it is doing. For example, look at the boldface code in the following snippet:

```
for(i = 2; i < isPrime.Length; i++)
{
   if(isPrime[i])
   {
      result.Add(i);
      for(int j = 2 * i; j < isPrime.Length; j += i)
          isPrime[j] = false; // multiple is not prime
   }
}
```

What does the highlighted loop do? It is clear what the loop does; there is a code comment explaining what it does. The comment is a good indicator that the code does not communicate its intent directly. It needs the comment to say what it does.

> **Note** When you see a block of code with a comment attached to it, it is often a good idea to extract that code into a method and make sure that the method's name conveys the meaning specified by the comment.

Let's extract the boldface code into its own method named *RemoveMultiples*:

```
private static void RemoveMultiples(int prime, bool[] isPrime)
{
    for(int j = 2 * prime; j < isPrime.Length; j += prime)
        isPrime[j] = false;
}
```

After the method is extracted, we need to modify the code to use it. Here is the modified code:

```
    for(i = 2; i < isPrime.Length; i++)
    {
        if(isPrime[i])
        {
            result.Add(i);
            RemoveMultiples(i, isPrime);
        }
      }
```

Instead of needing the comment, the method name communicates exactly what it is doing.

Refactoring 14: Extract Method (Again)

The code is getting smaller and smaller with more explicitly named methods and variables; in fact, we can now see that there are two stages in the algorithm: initialization and elimination. Let's extract the elimination portion into a method called *Sieve* using the "Extract method" refactoring (the changes are boldface):

```
public static ArrayList Generate(int maxValue)
{
    ArrayList result = new ArrayList();
```

```
    if(maxValue < 2) return result;

    bool[] isPrime = new bool[maxValue+1];
    int i;

    for(i = 0; i < isPrime.Length; i++)
        isPrime[i] = true;

    Sieve(isPrime, result);

    return result;
}

private static void Sieve(bool[] isPrime, ArrayList result)
{
    for(int i = 2; i < isPrime.Length; i++)
    {
        if(isPrime[i])
        {
            result.Add(i);
            RemoveMultiples(i, isPrime);
        }
    }
}

private static void RemoveMultiples(int prime, bool[] isPrime)
{
    for(int j = 2 * prime; j < isPrime.Length; j += prime)
        isPrime[j] = false;
}
```

The code is much more explicit. Before we go on, however, let's make one more change. The *Sieve* function can return the *ArrayList* instead of getting it passed to it; as you see here:

```
public static ArrayList Generate(int maxValue)
{
    if(maxValue < 2) return new ArrayList();

    bool[] isPrime = new bool[maxValue+1];
    int i;

    for(i = 0; i < isPrime.Length; i++)
        isPrime[i] = true;

    return Sieve(isPrime);
}
```

```
    private static ArrayList Sieve(bool[] isPrime)
    {
        ArrayList result = new ArrayList();

        for(int i = 2; i < isPrime.Length; i++)
        {
            if(isPrime[i])
            {
                result.Add(i);
                RemoveMultiples(i, isPrime);
            }
        }

        return result;
    }

    private static void RemoveMultiples(int prime, bool[] isPrime)
    {
        for(int j = 2 * prime; j < isPrime.Length; j += prime)
            isPrime[j] = false;
    }
```

Refactoring 15: Reduce Local Variable Scope

Because we extracted a method that used the variable *i*, we can reduce the scope of the variable in the *Generate* method. The following code

```
int i;
for(i=0; i < isPrime.Length; i++)
   isPrime[i] = true;
```

now becomes

```
for(int i=0; i < isPrime.Length; i++)
      isPrime[i] = true;
```

Even though the step is small, it is still important to compile the code and run the tests. If you don't, you could have a failure a couple of steps ahead and not know exactly what was changed.

Refactoring 16: Convert Procedural Design to Objects

We previously discussed the two steps in the algorithm: initialization and elimination. There is also a variable, *isPrime*, that is shared between the two stages. So we have the following:

- State (*isPrime*)
- Logic to initialize the state
- Logic to operate on the state

This set of conditions sounds as if we need an object to hold this state, a constructor to initialize the state, and a method to manipulate this state. Meet the next refactoring: "Convert procedural design to objects." This step is a little bit larger, so it probably makes sense to comment out the existing code first so that we have something to fall back on if we fail. Another alternative is to check the file into your source code control system and then make the change. If you fail, you can easily roll back to the previous version. The code after the refactoring looks like this:

```
public static ArrayList Generate(int maxValue)
{
    if(maxValue < 2) return new ArrayList();

    Primes primes = new Primes(maxValue);
    return primes.Sieve();
}

private bool[] isPrime;

private Primes(int maxValue)
{
    isPrime = new bool[maxValue+1];

    for(int i = 0; i < isPrime.Length; i++)
        isPrime[i] = true;
}

private ArrayList Sieve()
{
    ArrayList result = new ArrayList();

    for(int i = 2; i < isPrime.Length; i++)
    {
        if(isPrime[i])
        {
            result.Add(i);
            RemoveMultiples(i, isPrime);
        }
    }

    return result;
}
```

```
private void RemoveMultiples(int prime, bool[] isPrime)
{
    for(int j = 2 * prime; j < isPrime.Length; j += prime)
        isPrime[j] = false;
}
```

We really did not write a lot of new code; we just moved what we had around a bit. After we compiled and ran the tests, they did pass the first time. We then went back and removed the commented-out code. We are definitely getting close to a point of diminishing returns, but let's move on.

Refactoring 17: Keep the Data Close to Where It Is Used

For the first time, the code actually looks like object-oriented code. What a departure from what we had! Now that we have an object, we can see that the *Sieve* method could do a bit more, and the *Generate* method might do a bit less. The guard clause from the *Generate* method can be tucked away into the *Sieve* method to fully encapsulate the algorithm. Here is the code after applying this refactoring:

```
public static ArrayList Generate(int maxValue)
{
    Primes primes = new Primes(maxValue);
    return primes.Sieve();
}

private bool[] isPrime;

private Primes(int maxValue)
{
    isPrime = new bool[maxValue+1];

    for(int i = 0; i < isPrime.Length; i++)
        isPrime[i] = true;
}

private ArrayList Sieve()
{
    if(isPrime.Length < 2) return new ArrayList();

    ArrayList result = new ArrayList();
    for(int i = 2; i < isPrime.Length; i++)
    {
        if(isPrime[i])
        {
            result.Add(i);
```

```
                RemoveMultiples(i, isPrime);
            }
        }

        return result;
    }

    private void RemoveMultiples(int prime, bool[] isPrime)
    {
        for(int j = 2 * prime; j < isPrime.Length; j += prime)
            isPrime[j] = false;
    }
```

After scanning the code, there really isn't much left to do, so we are finished.

Summary

In this chapter, we demonstrated the following points:

- Refactoring allows the design of the code to improve by following a series of simple steps. For example, in this chapter we went from bad procedural code to a cleaner object-oriented implementation—while staying close to the green bar and without a large-scale rewrite. When you write your code, we expect you will refactor as you discover the need for it (not when it's too late and the code is so messed-up that it is more appealing just to throw it away and write it anew). The more "paranoid" you are about all the little problems in the code, the more proactive you will be in correcting them when you notice them rather than waiting until you have a big job on your hands.
- There was no mention of a debugger. Due to the small steps and the ability to verify them with the tests, you will not have to spend as much time debugging the software because changes can easily be rolled back to the previous state.
- The ability to do refactoring is a benefit that you receive from your investment in tests. The tests provide the safety net that enables the routine maintenance of the program. These tests allow you to alter the code without worrying about whether or not you have broken it.

Without the tests, you would not be able to move as quickly or as incrementally through this problem. In fact, you probably would have scrapped the whole thing and rewritten it.

- You should not turn your "pragmatic paranoia" into a "morbid obsession." Your goal, after all, is to write software efficiently and not get stuck tweaking existing code into unattainable "perfection." When do you stop refactoring? There is no simple and fast rule here that we can offer. The general rule of thumb is that you need to refactor whatever code duplication you discover and move toward code that clearly communicates your intentions. And if you have some amount of code duplication that serves the goal of clearly communicating your intentions, it is all right to keep it.
- Last, the order in which we did the refactorings is only an example. There are many other ways this code could be refactored and many other possible implementations. The main point driven home by the series of steps is that the code is the primary feedback mechanism for possible future refactorings.

Index

A

B

C

D

E

F

G

H–I

J–L

M

N

O

P

R

S

T

U

V

W

X-Z