Chapter 11. Design and testability

This chapter covers

- Benefiting from testability design goals
- Weighing pros and cons of designing for testability
- Tackling hard-to-test design

Changing the design of your code so that it's more easily testable is a controversial issue for some developers. This chapter will cover the basic concepts and techniques for designing for testability. We'll also look at the pros and cons of doing so and when it's appropriate.

First, though, let's consider why you would need to design for testability in the first place.

11.1. Why should I care about testability in my design?

The question is a legitimate one. When designing software, you learn to think about what the software should accomplish and what the results will be for the end user of the system. But tests against your software are yet another type of user. That user has strict demands for your software, but they all stem from one mechanical request: testability. That request can influence the design of your software in various ways, mostly for the better.

In a testable design, each logical piece of code (loops, `if` s, switches, and so on) should be easy and quick to write a unit test against, one that demonstrates these properties:

- Runs fast

- Is isolated, meaning it can run independently or as part of a group of tests, and can run before or after any other test
- Requires no external configuration
- Provides a consistent pass/fail result

These are the FICC properties: fast, isolated, configuration-free, and consistent. If it's hard to write such a test, or if it takes a long time to write it, the system isn't testable.

If you think of tests as a user of your system, designing for testability becomes a way of thinking. If you were doing test-driven development, you'd have no choice but to write a testable system, because in TDD the tests come first and largely determine the API c of the system, forcing it to be something that the tests can work with.

Now that you know what a testable design is, let's look at what it entails, go over the pros and cons of such design decisions, discuss alternatives to the testable design approach, and look at an example of hard-to-test design.

11.2. Design goals for testability

There are several design points that make code much more testable. Robert C. Martin has a nice list of design goals for object-oriented systems that largely form the basis for the designs shown in this chapter. See his article, "Principles of OOD," at http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

Most of the advice I include here is about allowing your code to have seams—places where you can inject other code or replace behavior without changing the original class. (Seams are often talked about in connection with the *Open–Closed Principle*, which is mentioned in Martin's "Principles of OOD.") For example, in a method that calls a web service, the web service API can hide behind a web service interface, allowing you to replace the real web service with a stub that will return whatever values you want or with a mock object. Chapters 3–5 discuss fakes, mocks, and stubs in detail.

Table 11.1 lists basic design guidelines and their benefits. The following sections will discuss them in more detail.

Table 11.1. Test design guidelines and benefits (view table figure)

Design guideline	Benefit(s)
Make methods virtual by default.	This allows you to override the methods in a derived class for testing. Overriding allows for changing behavior or breaking a call to an external dependency.
Use interface-based designs.	This allows you to use polymorphism to replace dependencies in the system with your own stubs or mocks.
Make classes nonsealed by default.	You can't override anything virtual if the class is sealed (final in Java).
Avoid instantiating concrete classes inside methods with logic. Get instances of classes from helper methods, factories, inversion of control containers such as Unity, or other places, but don't directly create them.	This allows you to serve up your own fa instances of classes to methods that require them, instead of being tied down to working with an internal production instance of a class.
Avoid direct calls to static methods. Prefer calls to instance methods that later call statics.	This allows you to break calls to static methods by overriding instance methods. (You won't be able to override static methods.)
Avoid constructors and static constructors that do logic.	Overriding constructors is difficult to implement. Keeping constructors simple will simplify the job of inheriting from a class in your tests.
Separate singleton logic from singleton holders.	If you have a singleton, have a way to replace its instance so you can inject a stub singleton or reset it.

11.2.1. Make methods virtual by default

Java makes methods virtual by default, but .NET developers aren't so lucky. In .NET, to be able to replace a method's behavior, you need to explicitly set it as virtual so you can override it in a default class. If you do this, you can use the Extract and Override method that I discussed in chapter 3.

An alternative to this method is to have the class invoke a custom delegate. You can replace this delegate from the outside by setting a property or sending in a parameter to a constructor or method. This isn't a typical approach, but some system designers find this approach suitable. The following listing shows an example of a class with a delegate that can be replaced by a test.

Listing 11.1. A class that invokes a delegate that can be replaced by a test

```
public class MyOverridableClass
{
    public Func<int,int> calculateMethod=delegate(int i)
                                            {
                                                return i*2;
                                            };
    public void DoSomeAction(int input)
    {
        int result = calculateMethod(input);
        if (result==-1)
        {
             throw new Exception("input was invalid");
        }
        //do some other work
    }
}

    [Test]
    [ExpectedException(typeof(Exception))]
    public void DoSomething_GivenInvalidInput_ThrowsException()
    {
        MyOverridableClass c = new MyOverridableClass();
        int SOME_NUMBER=1;

        //stub the calculation method to return "invalid"
        c.calculateMethod = delegate(int i) { return -1; };

        c.DoSomeAction(SOME_NUMBER);
    }
```


Using virtual methods is handy, but interface-based designs are also a good choice, as the next section explains.

11.2.2. Use interface-based designs

Identifying "roles" in the application and abstracting them under interfaces is an important part of the design process. An abstract class shouldn't call concrete classes, and concrete classes shouldn't call concrete classes either, unless they're data objects (objects holding data, with no behavior). This allows you to have multiple seams in the application where you could intervene and provide your own implementation.

For examples of interface-based replacements, see chapters 3–5.

11.2.3. Make classes nonsealed by default

Some people have a hard time making classes nonsealed by default because they like to have full control over who inherits from what in the application. The problem is that if you can't inherit from a class, you can't override any virtual methods in it.

Sometimes you can't follow this rule because of security concerns, but following it should be the default, not the exception.

11.2.4. Avoid instantiating concrete classes inside methods with logic

It can be tricky to avoid instantiating concrete classes inside methods that contain because you're so used to doing it. The reason for doing so is that later your tests need to control what instance is used in the class under test. If there's no seam that returns that instance, the task would be much more difficult unless you employ unconstrained isolation frameworks, such as Typemock Isolator. If your method relies on a logger, for example, don't instantiate the logger inside the method. Get it from a simple factory method, and make that factory method virtual so that you can override it later and control what logger your method works against. Or use DI via a constructor instead of a virtual method. These and more injection methods are discussed in chapter 3.

11.2.5. Avoid direct calls to static methods

Try to abstract any direct dependencies that would be hard to replace at runtime. In most cases, replacing a static method's behavior is difficult or cumbersome in a static language like VB.NET or C#. Abstracting a static method away using the Extract and Override refactoring (shown in section 3.4 of chapter 3) is one way to deal with these situations.

A more extreme approach is to avoid using any static methods whatsoever. That way, every piece of logic is part of an instance of a class that makes that piece of logic more easily replaceable. Lack of replaceability is one of the reasons why some people who do unit testing or TDD dislike singletons; they act as a public shared resource that is static, and it's hard to override them.

Avoiding static methods altogether may be too difficult, but trying to minimize the number of singletons or static methods in your application will make things easier for you while testing.

11.2.6. Avoid constructors and static constructors that do logic

Things like configuration-based classes are often made static classes or singletons because so many parts of the application use them. That makes them hard to replace during a test. One way to solve this problem is to use some form of inversion of control (IoC) containers (such as Microsoft Unity, Autofac, Ninject, StructureMap, Spring.NET, or Castle Windsor—all open source frameworks for .NET).

These containers can do many things, but they all provide a common smart factory, of sorts, that allows you to get instances of objects without knowing whether the instance is a singleton or what the underlying implementation of that instance is. You ask for an interface (usually in the constructor), and an object that matches that type will be provided for you automatically, as your class is being created.

When you use an IoC container (also known as a DI container), you abstract away the lifetime management of an object type and make it easier to create an object model that's largely based on interfaces, because all the dependencies in a class are automatically filled up for you.

Discussing containers is outside the scope of this book, but you can find a comprehe list and some starting points in the article, "List of .NET Dependency Injection Conta (IOC)" on Scott Hanselman's blog: http://www.hanselman.com/blog/ListOfNETDependencyInjectionContainersIOC.aspx.

11.2.7. Separate singleton logic from singleton holders

If you're planning to use a singleton in your design, separate the logic of the singleton class and the logic that makes it a singleton (the part that initializes a static variable, for example) into two separate classes. That way, you can keep the single responsibility principle (SRP) and also have a way to override singleton logic.

For example, the next listing shows a singleton class, and listing 11.3 shows it refactored into a more testable design.

Listing 11.2. An untestable singleton design

```
public class MySingleton
    {
        private static MySingleton _instance;

        public static MySingleton Instance
        {
            get
            {
                if (_instance == null)
                {
                    _instance = new MySingleton();
                }
                return _instance;
            }
        }
    }
```

Listing 11.3. The singleton class refactored into a testable design

```
public class RealSingletonLogic #1
    {
        public void Foo()
        {
            //lots of logic here
        }
    }
public class MySingletonHolder  #2
    {
        private static RealSingletonLogic _instance;
        public static RealSingletonLogic Instance
        {
            get
            {
                if (_instance == null)
                {
                    _instance = new RealSingletonLogic();
                }
                return _instance;
            }
        }
    }
```


Now that we've gone over some possible techniques for achieving testable designs, let's get back to the larger picture. Should you do it at all, and are there negative consequences of doing it?

11.3. Pros and cons of designing for testability

Designing for testability is a loaded subject for many people. Some believe that testability should be one of the default traits of designs, and others believe that designs shouldn't "suffer" just because someone will need to test them.

The thing to realize is that testability isn't an end goal in itself but is merely a byproduct of a specific school of design that uses the more testable object-oriented principles laid out by Robert C. Martin (mentioned at the beginning of section 11.2). In a design that favors class extensibility and abstractions, it's easy to find seams for test-related actions. All the techniques shown in this chapter so far are very much aligned with Martin's principles: classes whose behavior can be changed by inheriting and overriding, or by injecting an interface, are "open for extension, but closed for modification"—the Open-Closed Principle. Those classes usually also exhibit the DI principle and the IoC principle combined, to allow constructor injection. By using the Single-Responsibility Principle you can, for example, separate a singleton from its holding logic into a separate singleton holder class. Only the Liskov substitution principle remains alone in the corner, because I couldn't think of a example where breaking it also breaks testability. But the fact that your testable de seem to be somehow correlating with the SOLID principles does *not* necessarily mean design is good or that you have design skill. Oh no. Your design, most likely, like mine, could be better. Grab a good book about this subject like *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley Professional, 2003) by Eric Evans or *Refactoring to Patterns* (Addison-Wesley Professional, 2004) by Joshua Kerievsky. How about *Clean Code* by Robert Martin? Works too!

I find lots of badly designed, very testable code out there. Proof positive that TDD, without proper design knowledge, is not necessarily a good influence on design.

The question remains, is this the best way to do things? What are the cons of such a testability-driven design method? What happens when you have legacy code? And so on.

11.3.1. Amount of work

In most cases, it takes more work to design for testability than not because doing so usually means writing more code. Even Uncle Bob, in his lengthy and occasionally funny videos on http://cleancoders.com, likes to say (in a Sherlock Holmes voice, holding a pipe) that he starts out with simplistic designs that do the simplest thing, and then he refactors only when he sees the need for it.

You could argue that the extra design work required for testability points out design issues that you hadn't considered and that you might have been expected to incorporate in your design anyway (separation of concerns, Single-Responsibility Principle, and so on).

On the other hand, assuming you're happy with your design as is, it can be problematic to make changes for testability, which isn't part of production. Again, you could argue that test code is as important as production code, because it exposes the API usage characteristics of your domain model and forces you to look at how someone will use your code.

From this point on, discussions of this matter are rarely productive. Let's just say that more code, and work, is required when testability is involved, but that designing for testability makes you think about the user of your API more, which is a good thing.

11.3.2. Complexity

Designing for testability can sometimes feel a little (or a lot) like it's overcomplicating things. You can find yourself adding interfaces where it doesn't feel natural to use interfaces or exposing class-behavior semantics that you hadn't considered before. In particular, when many things have interfaces and are abstracted away, navigating the code base to find the real implementation of a method can become more difficult and annoying.

You could argue that using a tool such as ReSharper makes this argument obsolete, because navigation with ReSharper is much easier. I agree that it eases most of the navigational pains. The right tool for the right job can help a lot.

11.3.3. Exposing sensitive IP

Many projects have sensitive intellectual property that shouldn't be exposed but that designing for testability would force to be exposed: security or licensing information, for example, or perhaps algorithms under patent. There are workarounds for this—keeping things internal and using the `[InternalsVisibleTo]` attribute—but they essentially defy the whole notion of testability in the design. You're changing the design but still keeping the logic hidden. Big deal.

This is where designing for testability starts to melt down a bit. Sometimes you can't work around security or patent issues. You have to change what you do or compromise on the way you do it.

11.3.4. Sometimes you can't

Sometimes there are political or other reasons for the design to be done a specific way, and you can't change or refactor it (Soul Crushing Enterprise software projects, anyone?). Sometimes you don't have the time to refactor your design, or the design is too fragile to refactor. This is another case where designing for testability breaks down—when the environment prevents you. It's an example of the influence factors discussed in chapter 9.

Now that we've gone through some pros and cons, it's time to consider alternatives to designing for testability.

11.4. Alternatives to designing for testability

It's interesting to look outside the box at other languages to see other ways of working.

In dynamic languages such as Ruby or Smalltalk, the code is inherently testable because you can replace anything and everything dynamically at runtime. In such a language, you can design the way you want without having to worry about testability. You don't need an interface in order to replace something, and you don't need to make something public to override it. You can even change the behavior of core types dynamically, and no one will yell at you or tell you that you can't compile.

In a world where everything is testable, do you still design for testability? The expected answer is, of course, no. In that sort of world, you should be free to choose your own design.

11.4.1. Design arguments and dynamically typed languages

Interestingly enough, since 2010 there has been growing talk in the Ruby community, which I've also been part of, about SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation, and Dependency inversion) design. "Just because you can, doesn't mean you should" say some Rubyists, for example, Avdi Grimm, the author of *Objects on Rails* available at http://objectsonrails.com. You can find many blog ruminating about the state of design in the Rails community, such http://jamesgolick.com/2012/5/22/objectify-a-better-way-to-build-rails-applications.ht Other Rubyists answer back with, "Don't bother us with this overengineering crap." Most notably, David Heinemeier Hansson, a.k.a. DHH, the initial creator of the Ruby on Rails framework, answers in a blog post "Dependency injection is not a virtue" at http://david.heinemeierhansson.com/2012/dependency-injection-is-not-a-virtue.html.

Then fun ensues on Twitter, as you can imagine.

The funny thing about these kinds of discussions is just how much they remind me of the same types of discussions that ensued around 2008–2009 in the .NET community and specifically the recently deceased ALT.NET community. (Most of the ALT.NET folks discovered Ruby or Node.js and moved on from .NET, only to come back a year later and do .NET on the side "for the money." Guilty!) The big difference here is that this is Ruby we're talking about. In the .NET community, there was at least a shred of half-baked evidence that seemed to back the side of the "Let's design SOLID" folks: you couldn't test your designs without having open/closed classes, for example, because the compiler would thump your head if you even tried. So all the design folks said, "See? The compiler is trying to tell you your design sucks," which in retrospect is rather silly, because many testable designs still seem to be overly sucky, albeit testable. Now, here come some Ruby people and say they want to use SOLID principles? Why on earth would they want to do that?

It seems that there are some extra benefits to using SOLID: code is more easily maintained and understood, which in the Ruby world can be a very big problem. Sometimes it's a bigger problem for Ruby than statically typed languages, because in Ruby you can have dynamic code calling all sorts of nasty hidden redirected code underneath, and you can end up in a world of hurt when that happens. Tests help, but only to a degree.

Anyway, what was my point? It was that initially, people didn't even try to make the design in Ruby software testable because the code was already testable. Things were just fine, and then they discovered ideas about the *design* of code; this implies that *design* is a separate activity, with different consequences than just simple testability-related code refactoring.

Back to .NET and statically typed languages: consider a .NET-related analogy that shows how using tools can change the way you think about problems and sometimes make big problems a non-issue. In a world where memory is managed for you, do you still design for memory management? Mostly, "no" would be the answer. If you're working in languages where memory isn't managed for you (C++, for example), you need to worry about and design for memory optimization and collection, or the application will suffer. This doesn't stop you from having properly designed code, but memory management isn't the reason for it. Code readability, usability, and other values drive it. You don't use a straw man in your design arguments to design your code, because you might be leaning on the wrong stick to make your case (too many analogies? I know. It's like...oh, never mind).

In the same way, by following testable, object-oriented design principles, you migh testable designs as a by-product, but testability shouldn't be a goal in your desig there to solve a specific problem. If a tool comes along that solves the testability problem for you, there'll be no need to design specifically for testability. There are other merits to such designs, but using them should be a choice and not a fact of life.

The main problem with nontestable designs is their inability to replace dependencies at runtime. That's why you need to create interfaces, make methods virtual, and do many other related things. There are tools that can help replace dependencies in .NET code without needing to refactor it for testability. This is one place where unconstrained isolation frameworks come into play.

Does the fact that unconstrained frameworks exist mean that you don't need to design for testability? In a way, yes. It rids you of the need to think of testability as a design goal. There are great things about the object-oriented patterns Bob Martin presents, and they should be used not because of testability, but because they make sense with respect to design. They can make code easier to maintain, easier to read, and easier to develop, even if testability is no longer an issue.

We'll round out our discussion with an example of a design that's difficult to test.

11.5. Example of a hard-to-test design

It's easy to find interesting projects to dig into. One such project is the open source BlogEngine.NET, whose source code you can find at http://blogengine.codeplex.com/SourceControl/latest. You'll be able to tell when a project was built without a test-driven approach or any testability in mind. In this case, there are statics all over the place: static classes, static methods, static constructors. That's not bad in terms of design. Remember, this isn't a book about design. But this case *is* bad in terms of testability.

Here's a look at a single class from that solution: the `Manager` class under the Ping namespace (located at http://blogengine.codeplex.com/SourceControl/latest#BlogEngine/BlogEngine.Core/Ping/Manager.cs):

```
namespace BlogEngine.Core.Ping
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text.RegularExpressions;

public static class Manager
    {

private static readonly Regex TrackbackLinkRegex = new Regex(
            "trackback:ping=\"([^\"]+)\"", RegexOptions.IgnoreCase |
    RegexOptions.Compiled);

        private static readonly Regex UrlsRegex = new Regex(
            @"<a.*?href=[""']?(?<url>.*?)[""'].*?>(?<name>.*?)</a>",
    RegexOptions.IgnoreCase | RegexOptions.Compiled);

        public static void Send(IPublishable item, Uri itemUrl)
        {
            foreach (var url in GetUrlsFromContent(item.Content))
            {
                var trackbackSent = false;

                if (BlogSettings.Instance.EnableTrackBackSend)
                {
                    // ignoreRemoteDownloadSettings should be set to true
                    // for backwards compatibilty with
                    // Utils.DownloadWebPage.
                    var remoteFile = new RemoteFile(url, true);
                    var pageContent = remoteFile.GetFileAsString();

     var trackbackUrl = GetTrackBackUrlFromPage(pageContent);

                    if (trackbackUrl != null)
                    {
                        var message =
                     new TrackbackMessage(item, trackbackUrl, itemUrl);
                        trackbackSent = Trackback.Send(message);
                    }
                }

                if (!trackbackSent &&
```

```
                    BlogSettings.Instance.EnablePingBackSend)
                {
                    Pingback.Send(itemUrl, url);
                }
            }
        }

        private static Uri GetTrackBackUrlFromPage(string input)
        {
            var url =
               TrackbackLinkRegex.Match(input).Groups[1].ToString().Trim();
            Uri uri;

            return
            Uri.TryCreate(url, UriKind.Absolute, out uri) ? uri : null;
        }

        private static IEnumerable<Uri> GetUrlsFromContent(string content)
        {
            var urlsList = new List<Uri>();
            foreach (var url in
                UrlsRegex.Matches(content).Cast<Match>().Select(myMatch =>
    myMatch.Groups["url"].ToString().Trim()))
            {
                Uri uri;
                if (Uri.TryCreate(url, UriKind.Absolute, out uri))

                {
                    urlsList.Add(uri);
                }
            }

            return urlsList;
        }
    }
}
```


We'll focus on the `send` method of the `Manager` class. This method is supposed to send some sort of ping or trackback (we don't really care what those mean for the purposes of this discussion) if it finds any kind of URLs mentioned in a blog post from a user. There are many requirements already implemented here:

- Only send the ping or trackback if a global configuration object is configured to `true` .

- If a ping isn't sent, try to send a trackback.
- Send a ping or trackback for any of the URLs you can find in the content of the post.

Why do I think this method is really hard to test? There are several reasons:

- The dependencies (such as the configuration) are all static methods, so you can't fake them easily and replace them without an unconstrained framework.
- Even if you were able to fake the dependencies, there's no way to inject them as parameters or properties. They're used directly.
- You could try to use Extract and Override (discussed in chapter 3) to call the dependencies through virtual methods that you can override in a derived class, except that the `Manager` class is static, so it can't contain nonstatic methods and obviously no virtual ones. So you can't even extract and override.
- Even if the class wasn't static, the method you want to test is static, so it can't call virtual methods directly. The method needs to be an instance method to be refactored into extract and override. And it's not.

Here's how I'd go about refactoring this class (assuming I had integration tests):

1. Remove the static from the class.

2. Create a copy of the `Send()` method with the same parameters but not static. I'd prefix it with `Instance` so it's named `InstanceSend()` and will compile without clashing with the original static method.

3. Remove all the code from inside the original static method, and replace it with `Manager().Send(item, itemUrl)` ; so that the static method is now just a forwarding mechanism. This makes sure all existing code that calls this method doesn't break (a.k.a. refactoring!).

4. Now that I have an instance class and an instance method, I can go ahead and use Extract and Override on parts of the `InstanceSend()` method, breaking dependencies such as extracting the call to `BlogSettings.Instance.EnableTrackBackSend` into its own virtual method that I can override later by inheriting in my tests from `Manager` .

5. I'm not finished yet, but now I have an opening. I can keep refactoring and extracting and overriding as I need.

Here's what the class ends up looking like before I can start using Extract and Override:

```
public static class Manager
    {
        ...

        public static void Send(IPublishable item, Uri itemUrl)
        {
            new Manager().Send(item,itemUrl);
        }
        public static void InstanceSend(IPublishable item, Uri itemUrl)
        {
            foreach (var url in GetUrlsFromContent(item.Content))
            {
                var trackbackSent = false;

                if (BlogSettings.Instance.EnableTrackBackSend)
                {
                    // ignoreRemoteDownloadSettings should be set to true
                    // for backwards compatibilty with
                    // Utils.DownloadWebPage.
                    var remoteFile = new RemoteFile(url, true);
                    var pageContent = remoteFile.GetFileAsString();
var trackbackUrl = GetTrackBackUrlFromPage(pageContent);

                    if (trackbackUrl != null)
                    {
                        var message =
                      new TrackbackMessage(item, trackbackUrl, itemUrl);
                        trackbackSent = Trackback.Send(message);
                    }
                }

                if (!trackbackSent &&
     BlogSettings.Instance.EnablePingBackSend)
                {
                    Pingback.Send(itemUrl, url);
                }
            }
        }

        private static Uri GetTrackBackUrlFromPage(string input)
        {
            ...
        }
```

```
        private static IEnumerable<Uri> GetUrlsFromContent(string content)
        {
            ...
}
}
```

Here are some things that I could have done to make this method more testable:

- Default classes to nonstatic. There's rarely a good reason to use a purely static class in C# anyway.
- Make methods instance methods instead of static methods.

There's a demo of how I do this refactoring in a video at an online TDD course at http://tddcourse.osherove.com.

11.6. Summary

In this chapter, we looked at the idea of designing for testability: what it involves in terms of design techniques, its pros and cons, and alternatives to doing it. There are no easy answers, but the questions are interesting. The future of unit testing will depend on how people approach such issues and on what tools are available as alternatives.

Testable designs usually only matter in static languages, such as C# or VB.NET, where testability depends on proactive design choices that allow things to be replaced. Designing for testability matters less in more dynamic languages, where things are much more testable by default. In such languages, most things are easily replaceable, regardless of the project design. This rids the community of such languages from the straw-man argument that the lack of testability of code means it's badly designed and lets them focus on what good design should achieve, at a deeper level.

Testable designs have virtual methods, nonsealed classes, interfaces, and a clear separation of concerns. They have fewer static classes and methods, and many more instances of logic classes. In fact, testable designs correlate to SOLID design principles but don't necessarily mean you have a good design. Perhaps it's time that the end goal should not be testability but good design alone.

We looked at a short example that's very untestable and all the steps it would take to refactor it into testability. Think how easily testable it would have been if TDD had been used to write it! It would have been testable from the first line of code, and we wouldn't have had to go through all these loops.

This is enough for now, grasshopper. But the world out there is awesome and filled with materials that I think you'd love to sink your teeth into.

11.7. Additional resources

I find that many of the people who read this book go through the following transformations:

- After they become comfortable with the naming conventions, they begin to adopt others or create their own. This is great. My naming conventions are good if you're a beginner, and I still use them myself, but they're not the only way. You should feel comfortable with your test names.
- They start looking at other forms of writing the tests, such as behavior-driven development (BDD)–style frameworks like MSpec or NSpec. This is great because as long as you keep the three important parts of information (what you're testing, under what conditions, and the expected result), readability is still good. In BDD-style APIs, it's easier to set a single point of entry and assert multiple end results on separate requirements, in a very readable way. This is because most BDD-style APIs allow a hierarchical way of writing them.
- They automate more integration and system tests, because they find unit testing to be too low-level. This is also great, because you do what you need to do to get the confidence you need to change the code. If you end up with no unit tests in your project but still can develop at high speed with confidence and quality, that's awesome, and could I get some of what you're having? (It's possible, but tests get very slow at some point. We still haven't found the magic way to make that happen fully.)

What about books?

One that complements the topics on this book in terms of design is *Growing Object–Oriented Software, Guided by Tests*, by Steve Freeman and Nat Pryce.

A good reference book for patterns and antipatterns in unit testing is *xUnit Test Patterns: Refactoring Test Code*, by Gerard Meszaros.

Working Effectively with Legacy Code by Michael Feathers is a must-read if you're dealing with legacy code issues.

There's also a more comprehensive and continuously (twice a year, really) updated list of interesting books at ArtOfUnitTesting.com.

For some test reviews, check out videos I've made, reading open source projects' tests and dissecting how they could be better, at http://artofunittesting.com/test-reviews/.

I've also uploaded a lot of free videos, test reviews, pair-programming sessions, and test-driven development conference talks to http://ArtOfUnitTesting.com and http://Osherove.com/Videos. I hope these will give you even more information in addition to this book.

You might also be interested in taking my TDD master class (available as online streaming videos) at http://TDDCourse.Osherove.com.

You can always catch me on twitter at @RoyOsherove, or just contact me directly through http://Contact.Osherove.com.

I look forward to hearing from you!