# 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 design 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**

| 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. |

**Table 11.1 Test design guidelines and benefits *(continued)***

| Design guideline | Benefit(s) |
|---|---|
| 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 fake 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 logic because you're so used to doing it. The reason for doing so is that later your tests might 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 comprehensive list and some starting points in the article, "List of .NET Dependency Injection Containers (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                          <-- Newly testable logic
    {
        public void Foo()
        {
            //lots of logic here
        }
    }

public class MySingletonHolder                           <-- Singleton container
    {
        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 single example where breaking it also breaks testability. But the fact that your testable designs seem to be somehow correlating with the SOLID principles does *not* necessarily mean your 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 posts ruminating about the state of design in the Rails community, such as http://jamesgolick.com/2012/5/22/objectify-a-better-way-to-build-rails-applications.html. 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 might get testable designs as a by-product, but testability shouldn't be a goal in your design. It's 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!

# *appendix*
# *Tools and frameworks*

This book wouldn't be complete without an overview of some tools and basic techniques you can use while writing unit tests. From database testing to UI testing and web testing, this appendix lists tools you should consider. Some are used for integration testing, and some allow unit testing. I'll also mention some that I think are good for beginners.

The tools and techniques listed here are arranged in the following categories:

- Isolation frameworks
- Test frameworks
  - Test runners
  - Test APIs
- Test helpers
- DI and IoC containers
- Database testing
- Web testing
- UI testing
- Thread-related testing
- Acceptance testing

**TIP** An updated version of the list of tools and techniques can be found on the book's website: http://ArtOfUnitTesting.com.

Let's begin.

## A.1 Isolation frameworks

Mock or isolation frameworks are the bread and butter of advanced unit testing scenarios. There are many to choose from, and that's a great thing:

- Moq
- Rhino Mocks

- Typemock Isolator
- JustMock
- Moles/Microsoft Fakes
- NSubstitute
- FakeItEasy
- Foq

The previous edition of this book contained the following tools, which I've removed due to being out of date or relevance:

- NMock
- NUnit.Mocks

Here's a short description of each framework.

### A.1.1 Moq

Moq is an open source isolation framework and has an API that tries to be both simple to learn and easy to use. The API was one of the first to follow the arrange-act-assert style (as opposed to the record-and-replay model in older frameworks) and relies heavily on .NET 3.5 and 4 features, such as lambdas and extension methods. You need to feel comfortable with using lambdas, and the same goes for the rest of the frameworks in this list.

It's quite simple to learn. My only beef with it is that the word *mock* is scattered all over the API, making this confusing. I would have liked, at least, to see a differentiation between creating stubs versus mocks, or using the word *fake* instead of both, to get rid of the confusion.

You can learn about Moq at http://code.google.com/p/moq/ and install it as a NuGet Package.

### A.1.2 Rhino Mocks

Rhino Mocks is a widely used, open source framework for mocks and stubs. Although in the previous edition of this book I recommended using it, I no longer do. Its development has all but stopped, and there are better, more lightweight, simpler, and better-designed frameworks out there. If you have a choice, don't use it. Ayende, the creator, mentioned in a Tweet that he isn't really working on it anymore.


You can get Rhino Mocks at http://ayende.com/projects/rhino-mocks.aspx.

### A.1.3 Typemock Isolator

Typemock Isolator is a commercial *unconstrained* (can fake anything, see chapter 6) isolation framework that tries to remove the terms *mocks* and *stubs* from its vocabulary in favor of a more simple and terse API.

Isolator differs from most of the other frameworks by allowing you to isolate components from their dependencies regardless of how the system is designed (although it supports all the features the other frameworks have). This makes it ideal for people who are getting into unit testing and want an incremental approach to learning about design and testability. Because it doesn't force you to design for testability, you can learn to write tests correctly and then move on to learning better design, without having to mix the two. It's also the most costly of the unconstrained bunch, which it makes up for in usability and features for legacy code.

Typemock Isolator has two flavors: a constrained basic edition that's free and has all the limitations of using a constrained framework (no statics, only virtuals, and so on) and an unconstrained, paid option that has few limits on what it can fake.

**NOTE** Full disclosure: I worked at Typemock between 2008 and 2010.

You can get Typemock Isolator at http://www.typemock.com.

### A.1.4 JustMock

JustMock, from Telerik, is a relatively new isolation framework that's a very obvious competitor to Typemock Isolator. The two frameworks' APIs are so similar in design that it should be relatively easy to move between them for the basic stuff. Like Typemock, JustMock has two flavors: a constrained free edition and an unconstrained, paid option that has few limits on what it can fake.

There's a little roughness with the APIs, and it currently, as far as I've been able to try, doesn't support recursive fakes—the ability to make a fake that returns a fake object that returns a fake object, without needing to specify things explicitly. Grab it at www.telerik.com/products/mocking.aspx.

### A.1.5 Microsoft Fakes (Moles)

Microsoft Fakes is a project that started out in Microsoft research as the answer to the question "How can we fake the filesystem and other things like SharePoint without needing to purchase a company like Typemock?" What emerged was a framework called Moles. Moles later grew into Microsoft Fakes and is included in some versions of Visual Studio.

MS Fakes is another unconstrained isolation framework, with no API for verification that something was called. In essence, it provides utilities to create stubs. If you'd like to assert that some object was called, you *could* do it, but the test code would look like a mess.

Like the previous unconstrained frameworks, MS Fakes allows you to create two types of fake objects: you either generate unconstrained classes that inherit and

override from code that is already testable, or you use shims. *Shims* are unconstrained, and *stubs*, the generated classes, are constrained. Confused? Yes, me too. One of the reasons I don't recommend that anyone but brave souls with nothing to lose use MS Fakes is because of the horrible usability factor. They're just confusing to use. Plus, the maintainability of the tests that use either shims or stubs is in question. The generated stubs need to be regenerated every time you change your code under test, followed by changing the tests, and code that uses shims is very long and hard to read and thus hard to maintain. MS Fakes might be free and included with Visual Studio, but it will cost you a lot of money down the line in developer hours, fixing and trying to understand your tests.

Another important point: using MS Fakes forces you to use MSTest as your test framework. If you want to use another one, you're out of luck.

If you need an unconstrained framework for writing tests that will need to last more than a week or two, choose JustMock or Typemock Isolator.

Learn more about MS Fakes at http://msdn.microsoft.com/en-us/library/hh549175.aspx.

### A.1.6 NSubstitute

NSubstitute is an open source constrained isolation framework. Its API is very simple to learn and remember, and it has very good documentation. Also good: errors are very detailed. Along with FakeItEasy, it's my first choice of a constrained framework for a new project.

Learn more about NSubstitute at http://nsubstitute.github.com/ and install it as a NuGet package.

### A.1.7 FakeItEasy

FakeItEasy has not only a great name but also a very nice API. It's my current favorite along with NSubstitute for constrained frameworks, but its documentation isn't as good as NSub's. My favorite thing about its API is that everything you'd want to accomplish starts with the character `A`, for example:

```
var foo = A.Fake<IFoo>();
A.CallTo(() => foo.Bar()).MustHaveHappened();
```

Learn more about FakeItEasy at https://github.com/FakeItEasy/FakeItEasy/wiki and install it as a NuGet package.

### A.1.8 Foq

Foq was created as the result of a need by F# programmers to create fakes in a way that's usable and readable in F#. It's a constrained isolation framework, able to create fakes of abstract classes and interfaces. I've personally not used it because I've never worked with F#, but it seems to be the only reasonable solution in that space, for that purpose. Learn more about Foq at https://foq.codeplex.com/ and install it as a NuGet package.

### A.1.9 Isolator++

Isolator++ was built by Typemock as an unconstrained isolation framework for C++. It can fake static methods, private methods, and more in legacy C++ code. It's another commercial product, and it seems to be the only one in this space with those abilities. Learn more about it at www.typemock.com/what-is-isolator-pp.

## A.2 Test frameworks

Test frameworks are composed of two types of functionality:

- Test runners execute the tests you write, give results, and allow you to know what went wrong where.
- Test APIs include the attributes or classes you need to inherit and assertion APIs.

Let's look at each in turn.

Visual Studio test runners:

- MS test runner built into Visual Studio
- TestDriven.NET
- ReSharper
- NUnit
- DevExpress
- Typemock Isolator
- NCrunch
- ContinuousTests (Mighty Moose)

Test and assertion APIs:

- NUnit.Framework
- Microsoft.VisualStudio.TestPlatform.UnitTestFramework
- Microsoft.VisualStudio.TestTools.UnitTesting
- FluentAssertions
- Shouldly
- SharpTestEx
- AutoFixture

### A.2.1 Mighty Moose (a.k.a. ContinuousTests) continuous runner

A previously commercial tool turned free, Mighty Moose is dedicated to giving feedback on tests and coverage continuously, like NCrunch.

- It runs the tests in a background thread.
- Tests are automatically run as you change code and save and compile.
- It has a smart algorithm to tell which tests need to be run based on which code was changed.

Unfortunately, it looks like development has stopped on this tool. Learn more at http://continuoustests.com.

### A.2.2 NCrunch continuous runner

NCrunch continuous runner is a commercial tool dedicated to giving feedback on tests and coverage continuously. While a relative newcomer, NCrunch has made its way into my heart (I purchased a license) because of several nice features:

- It runs the tests in a background thread.
- Tests are automatically run as you change code, without even needing to save it.
- Green/red coverage dots next to both tests and production code let you know if the current production line you're working on is covered by any test and if that test is currently failing.
- It's very configurable, to the point of annoyance. Just remember, when the wizard initially comes up on a simple project, just hit Esc to get the defaults of running all the tests.

Learn more at www.ncrunch.net/.

### A.2.3 Typemock Isolator test runner

This test runner is part of a commercial isolation framework called Typemock Isolator.

This extension tries to run the tests and show coverage at the same time, on every compilation. It's very much in beta state and has inconsistent behavior. Perhaps one day it will be more helpful, but these days I tend to turn it off and just use the isolation frameworks APIs.

Learn more at http://Typemock.com.

### A.2.4 CodeRush test runner

This test runner part of a commercial tool called CodeRush is a well-known plug-in for Visual Studio.

Like ReSharper, there are some nice pros for this runner:

- It's nicely integrated into the Visual Studio code editor by showing marks near a test that you can click to run individual tests.
- It supports most of the test APIs out there in .NET.
- If you're already using CodeRush, it's good enough.

Like ReSharper, the visual nature of the test results can hinder the experience for experienced TDD-ers. The tree of running tests, and showing all the results by default, even of passing tests, wastes time when you're in the flow of TDD. But some people like it. Your mileage may vary.

Learn more at www.devexpress.com/Products/Visual_Studio_Add-in/Coding_Assistance/unit_test_runner.xml.

### A.2.5 ReSharper test runner

This test runner is part of a commercial tool called ReSharper, a well-known plug-in for Visual Studio.

There are some nice pros for this runner:

- It's nicely integrated into the Visual Studio code editor by showing marks near a test that you can click to run individual tests.
- It supports most of the test APIs out there in .NET.
- If you're already using ReSharper, it's good enough.

What I consider a con is the overly visual nature of the test results. The tree of running the tests is very nice and colorful. But painting it, and showing all the results by default, even of passing tests, wastes time when you're in the flow of TDD. But some people like it. Your mileage may vary.

Learn more at www.jetbrains.com/resharper/features/unit_testing.html.

### A.2.6 TestDriven.NET runner

This is a commercial test runner (free for personal use). This used to be my favorite test runner until I started using NCrunch. There's a lot to love:

- It's able to run tests for most, if not all, the test API frameworks out there in .NET, including NUnit, MSTest, xUnit.net, as well as some of the BDD API frameworks.
- It's a little package. It's a very small install and a minimalistic interface. The output is simple: it appears in the output window of Visual Studio, and some text is on the bottom sidebars of Visual Studio.
- It's very fast, one of the fastest test runners.
- It has a unique ability: you can right-click *any* piece of code (not just tests) and select Test with > Debugger. You will then step into any code (even production code, even if it doesn't have tests). Underneath TD.NET invoke the method you're currently in using reflection, and it provides default values for the method if parameters are needed. This saves a lot of time in legacy code.

It is recommended to assign a shortcut to the TD.NET `ReRunTests` command in Visual Studio so that the flow of TDD is as smooth as possible.

### A.2.7 NUnit GUI runner

The NUnit GUI runner is free and open source. This runner isn't integrated into Visual Studio, so you have to run it from your desktop. Because of this, almost nobody uses it when they have the other options listed here integrated into Visual Studio. It's crude and unpolished and not recommended.

### A.2.8 MSTest runner

The MSTest runner comes built in with all versions of Visual Studio. In the paid versions it also has a plug-in mechanism that allows you to add support for running tests written in other test APIs such as NUnit or xUnit.net via special adapters that you can install as Visual Studio extensions.

One factor in favor of this runner is that it's integrated into the Visual Studio Team System tool suite and provides good reporting, coverage, and build automation out of the box. If your company uses Team System for automated builds, try MSTest as your test runner in the nightly and CI builds because of the good integration possibilities such as reporting.

Two areas where MSTest lacks are performance and dependencies:

- *Dependencies*—To run your tests using mstest.exe, you need to have Visual Studio installed on your build machine. That might be OK for some, especially if where you compile is the same place you run your tests. But if you want to run your tests in a relatively clean environment, in already compiled form, this can be overkill and problematic if you want your tests to run in an environment that specifically does not have Visual Studio installed.
- *Slow*—The tests in MSTest do a lot under the hood before and after each test, copying files, running external processes, profiling, and more, and this makes MSTest feel like the slowest runner of all the ones I've ever used.

### A.2.9 Pex

Pex (short for program exploration) is an intelligent assistant to the programmer. From a parameterized unit test, it automatically produces a traditional unit test suite with high code coverage. In addition, it suggests to the programmer how to fix the bugs.

With Pex, you can create special tests that have parameters in them and put special attributes on those tests. The Pex engine will generate new tests that you can later run as part of your test suite. It's great for finding corner cases and edge conditions that aren't handled properly in your code. You should use Pex in addition to a regular test framework, such as NUnit or MbUnit.

You can get Pex at http://research.microsoft.com/projects/pex/.

## A.3 Test APIs

The next batch of tools provides higher-level abstractions and wrappers for the base unit testing frameworks.

### A.3.1 MSTest API—Microsoft's unit testing framework

This comes bundled with any version of Visual Studio .NET Professional or above. It includes basic features that are similar to NUnit.

But several problems make MSTest an inferior product for unit testing compared to NUnit or xUnit.net:

- Extensibility
- Lack of `Assert.Throws`

#### EXTENSIBILITY

One big problem with this framework is that it's not as easily extensible as the other testing frameworks. Although there have been several online discussions in the past

about making MSTest more extensible with custom test attributes, it seems that the Visual Studio team has all but given up on making MSTest a viable alternative to NUnit and others.

Instead, VS 2012 features a plug-in mechanism that allows you to use NUnit or any other test framework as your default test framework, with the MSTest runner running your NUnit tests. There are already adapters available for NUnit and xUnit.net (NUnit test adapter or xUnit.net runner for Visual Studio 2012) if you just want to use the MSTest runner with other frameworks. Unfortunately, the express, free version of Visual Studio doesn't contain this mechanism, forcing you to use the inferior MSTest. (On a side note, why does Microsoft force you to purchase Visual Studio so that you can develop code that makes the MS platform more dominant?)

#### LACK OF ASSERT.THROWS

This is a simple matter. In MSTest you have an `ExpectedException` attribute, but you don't have `Assert.Throws`, which allows testing that a specific line threw an exception. Over six years after inception, and four years after most other frameworks, this framework's developers haven't bothered adding this literally 10 lines of code implementation to it, leaving me wondering how much they really care about unit tests.

### A.3.2 MSTest for Metro Apps (Windows Store)

MSTest for Metro Apps is an API for writing Windows Store apps that looks like MSTest but seems to get the right idea regarding unit tests. For example, it sports its own version of `Asset.ThrowsException()`.

It seems like you're forced to use this framework for writing Windows Store apps with unit tests, but a solution exists if you use linked projects. For information see http://stackoverflow.com/questions/12924579/testing-a-windows-8-store-app-with-nunit.

### A.3.3 NUnit API

NUnit is currently the de facto test API framework for unit test developers in .NET. It's open source and is in almost ubiquitous use among those who do unit testing. I cover NUnit in depth in chapter 2. NUnit is easily extensible and has a large user base and forums. I'd recommend it to anyone starting out with unit testing in .NET. I still use it today.

You can get NUnit at www.Nunit.org.

### A.3.4 xUnit.net

xUnit.net is an open source test API framework, developed in cooperation with one of the original authors of NUnit, Jim Newkirk. It's a minimalist and elegant test framework that tries to get back to basics by having fewer features, not more, than the other frameworks and by supporting different names on its attributes.

What's so radically different about it? It has no setup or teardown methods, for one. You have to use the constructor and a dispose method on the test class. Another big difference is in how easy it is to extend.

Because xUnit.net reads so differently from the other frameworks, it takes a while to get used to if you're coming from a framework like NUnit or MbUnit. If you've never used any test framework before, xUnit.net is easy to grasp and use, and it's robust enough to be used in a real project.

For more information and download see www.codeplex.com/xunit.

### A.3.5 Fluent Assertions helper API

The Fluent Assertions helper API is a new breed of test API. It's a cute library that's designed solely for one purpose: to allow you to assert on anything, regardless of the test API you're using. For example, you can use it to get `Assert.Throws()`-like functionality in MSTest.

More information is available at http://fluentassertions.codeplex.com/.

### A.3.6 Shouldly helper API

The Shouldly helper API is a lot like Fluent Assertions but smaller. It's also designed solely for one purpose: to allow you to assert on anything, regardless of the test API you're using. More information can be found at http://shouldly.github.com.

### A.3.7 SharpTestsEx helper API

Like Fluent Assertions, the SharpTestsEx helper API is designed solely for one purpose: to allow you to assert on anything, regardless of the test API you're using. More information is available at http://sharptestex.codeplex.com.

### A.3.8 AutoFixture helper API

The AutoFixture helper API is not an assertion API. AutoFixture is designed to make it easier to create objects under test that you don't care about. For example, you need some number or some string. Think of it as a smart factory that can inject objects and input values into your test.

I've looked at using it, and the thing I find most appealing about it is the ability to create an instance of the class under test without knowing what its constructor signature looks like, which can make my test more maintainable over time. Still, that's not enough reason for me to use it, because I can simply do that with a small factory method in my tests.

Also, it scares me a bit to let it inject random values into my tests, because it makes me run a different test each time I run it. It also complicates my asserts, because then I have to calculate that my expected output must be based on the random injected parameters, which may lead to repeating production code logic in my tests.

More information can be found at https://github.com/AutoFixture/AutoFixture.

## A.4 IoC containers

IoC containers can be used to improve the architectural qualities of an object-oriented system by reducing the mechanical costs of good design techniques (such as using constructor parameters, managing object lifetimes, and so on).

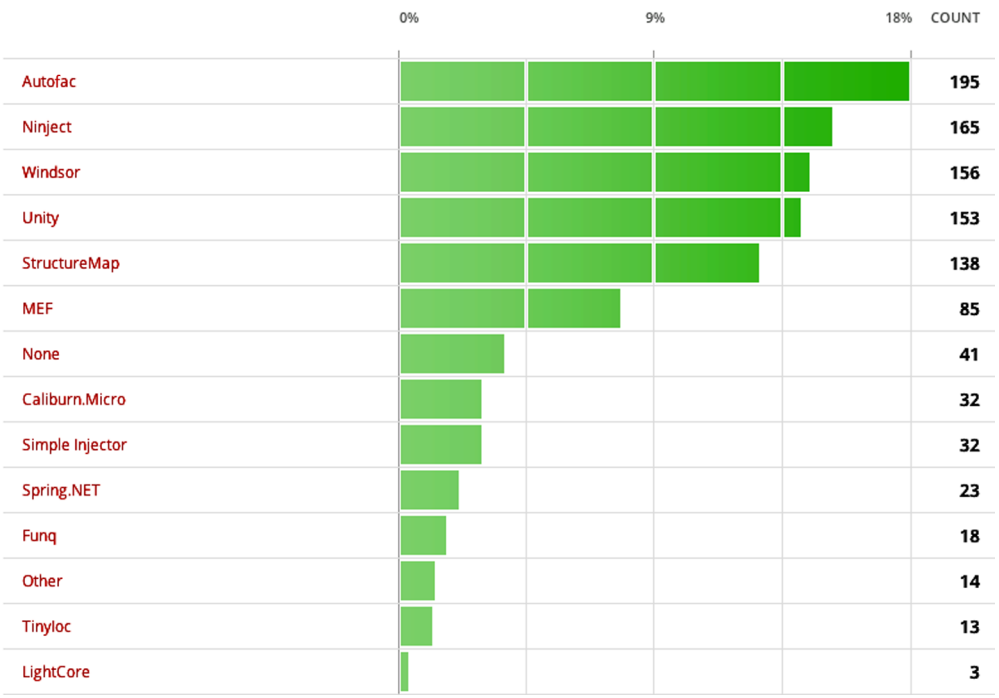
Containers can enable looser coupling between classes and their dependencies, improve the testability of a class structure, and provide generic flexibility mechanisms. Used judiciously, containers can greatly enhance the opportunities for code reuse by minimizing direct coupling between classes and configuration mechanisms (such as by using interfaces).

There are *a lot* of those in the .NET space. And they're varied and interesting to take a look at. Performance-wise, if you care much about that, there's a good comparison of them all at http://www.palmmedia.de/Blog/2011/8/30/ioc-container-benchmark-performance-comparison. Personally, I never felt that IoC containers were my root cause of performance issues, and if that were ever the case, that would be a very good place to be.

Anyway, there are lots of them, but we'll look at the following tools, which are used frequently in the community.

I chose the tools to cover based on a usage poll I took on my blog during March 2013. Here's the top of the results heap for usage:

- Autofac (Auto Factory)
- Ninject
- Castle Windsor
- Microsoft Unity
- StructureMap
- Microsoft Managed Extensibility Framework



Let's look briefly at each of these frameworks.

### A.4.1 Autofac

Autofac was one of the first to offer a fresh approach to IoC in .NET that fits well with the C# 3 and 4 syntax. It takes a rather minimalistic approach in terms of APIs. The API is radically different from those of the other frameworks and requires a bit of getting used to. It also requires .NET 3.5 to work, and you'll need a good knowledge of lambda syntax. Autofac is difficult to explain, so you'll have to go to the site to see how different it is. I recommend it for people who already have experience with other DI frameworks.

You can get it at http://code.google.com/p/autofac/.

### A.4.2 Ninject

Ninject also has simple syntax and good usability. There isn't much else to say about it except that I highly recommend taking a look at it.

You can find out more about Ninject at http://ninject.org/.

### A.4.3 Castle Windsor

Castle is a large, open source project that covers a lot of areas. Windsor is one of those areas, and it provides a mature and powerful implementation of a DI container.

Castle Windsor contains most of the features you'll ever want in a container and more, but it has a relatively high learning curve because of all the features.

You can learn about the Castle Windsor container at http://docs.castleproject.org/Windsor.MainPage.ashx.

### A.4.4 Microsoft Unity

Unity is a latecomer to the DI container field, but it provides a simple and minimal approach that can be easily learned and used by beginners. Advanced users may find it lacking, but it certainly answers my 80–20 rule: it provides 80% of the features you look for most of the time.

Unity is open source by Microsoft, and it has good documentation. I'd recommend it as a starting point for working with containers.

You can get Unity at www.codeplex.com/unity.

### A.4.5 StructureMap

StructureMap is an open source container framework written by Jeremy D. Miller. Its API is very fluent and tries to mimic natural language and generic constructs as much as possible.

The current documentation on it is lacking, but it contains powerful features, such as a built-in automocking container (a container that can create stubs automatically when requested to by the test), powerful lifetime management, XML-free configuration, integration with ASP.NET, and more.

You can get StructureMap at http://structuremap.net.

### A.4.6 Microsoft Managed Extensibility Framework

The Managed Extensibility Framework (MEF) isn't actually a container, but it does fall in the same general category of providing services that instantiate classes in your code. It's designed to be much more than a container; it's a full plug-in model for small and large applications. MEF includes a lightweight IoC container framework so you can easily inject dependencies into various places in your code by using special attributes.

MEF does involve a bit of a learning curve, and I wouldn't recommend using it strictly as an IoC container. If you do use it for extensibility features in your application, it can also be used as a DI container.

You can get MEF at http://mef.codeplex.com/.

## A.5 Database testing

How to do database testing is a burning question for those who are starting out. Many questions arise such as, "Should I stub out the database in my tests?" This section provides some guidelines.

First, let's talk about doing integration tests against the database.

### A.5.1 Use integration tests for your data layer

How should you test your data layer? Should you abstract away the database interfaces? Should you use the real database?

I usually write integration-style tests for the data layer (the part of the app structure that talks directly to the database) in my applications because data logic is almost always divided between the application logic and the database itself (triggers, security rules, referential integrity, and so on). Unless you can test the database logic in complete isolation (and I've found no really good framework for this purpose), the only way to make sure it works in tests is to couple testing the data-layer logic to the real database.

Testing the data layer and the database together leaves few surprises for later in the project. But testing against the database has its problems, the main one being that you're testing against state shared by many tests. If you insert a line into the database in one test, the next test can see that line as well.

What you need is a way to roll back the changes you make to the database, and thankfully, there's an easy way to do it in the .NET Framework.

### A.5.2 Use TransactionScope to roll back changes to data

The `TransactionScope` class is smart enough to handle very complicated transactions, as well as nested transactions where your code under test calls commits on its own local transaction.

Here's a simple piece of code that shows how easy it is to add rollback ability to your tests:

```
[TestFixture]
public class TrannsactionScopeTests
```

```
{
    private TransactionScope trans = null;
    [SetUp]
    public void SetUp()
    {
        trans = new TransactionScope(TransactionScopeOption.Required);
    }
    [TearDown]
    public void TearDown()
    {
        trans.Dispose();
    }

    [Test]
    public void TestServicedSameTransaction()
    {
        MySimpleClass c = new MySimpleClass();

        long id = c.InsertCategoryStandard("whatever");
        long id2 = c.InsertCategoryStandard("whatever");
        Console.WriteLine("Got id of " + id);
        Console.WriteLine("Got id of " + id2);
        Assert.AreNotEqual(id, id2);
    }

}
```

You set up a transaction scope in the setup and dispose of it in the teardown.

By not committing it at the test class level, you basically roll back any changes to the database, because dispose initiates a database rollback if commit wasn't called first.

Some feel that another good option is to run the tests against an in-memory database. My feelings on that are mixed. On the one hand, it's closer to reality, in that you also test the database logic. On the other hand, if your application uses a different database engine, with different features, there's a big chance that some things will pass or fail during tests with the in-memory database and will work differently in production. I choose to work with whatever is as close to the real thing as possible. Usually that means using the same database engine.

If the in-memory database engine has the same features and logic embedded in it, it might be a great idea.

## A.6 Web testing

"How do I test my web pages?" is another question that comes up a lot. Here are some tools that can help you in this quest:

- Ivonna
- Team System Web Test
- Watir
- Selenium

Following is a short description of each tool.

### A.6.1 Ivonna

Ivonna is a unit testing-helping framework that abstracts away the need to run ASP.NET-related tests using a real HTTP session and pages. It does some powerful things behind the scenes, such as compiling pages that you want to test and letting you test controls inside them without needing a browser session, and it fakes the full HTTP runtime model.

You write the code in your unit tests just like you're testing other in-memory objects. There's no need for a web server and such nonsense.

Ivonna is being developed in partnership with Typemock and runs as an add-on to the Typemock Isolator framework. You can get Ivonna at http://ivonna.biz.

### A.6.2 Team System web test

Visual Studio Team Test and Team Suite editions include the powerful ability to record and replay web requests for pages and verify various things during these runs. This is strictly integration testing, but it's really powerful. The latest versions also support recording Ajax actions on the page and make things much easier to test in terms of usability.

You can find more info on Team System at http://msdn.microsoft.com/en-us/teamsystem/default.aspx.

### A.6.3 Watir

Watir (pronounced "water") stands for "web application testing in Ruby." It's open source, and it allows scripting of browser actions using the Ruby programming language. Many Rubyists swear by it, but it does require that you learn a whole new language to use. A lot of .NET projects are using it successfully, so it's not a big deal.

You can get Watir at http://watir.com/.

### A.6.4 Selenium WebDriver

Selenium is a suite of tools designed to automate web app testing across many platforms. It's older than all the other frameworks in this list, and it also has an API wrapper for .NET. WebDriver is an extension of it that fits many different kinds of browsers, including mobile ones. It's very powerful.

Selenium is an integration testing framework, and it's in wide use. It's a good place to start. But beware: it has many features and the learning curve is high.

You can get it at http://docs.seleniumhq.org/projects/webdriver/.

### A.6.5 Coypu

Coypu is a .NET abstract on top of Selenium and other web-related testing tools. It's quite new at the time of writing but might have plenty of potential. It might be worth checking it out.

Learn more at https://github.com/featurist/coypu.

### A.6.6 Capybara

Capybara is a Ruby-based tool that automates the browser. It allows you to use the RSpec (BDD-style) API to automate the browser, which many people find just lovely to read.

Selenium is more mature, but Capybara is more inviting and progressing quickly. When I do Ruby stuff, this is what I use.

Learn more at https://github.com/jnicklas/capybara.

### A.6.7 JavaScript testing

There are several tools to look at if you intend to write unit tests or acceptance tests for JavaScript code. Note that many of these will require installing Node.JS on your machine, which is a no-brainer these days. Just head to http://nodejs.org/download/.

Here's a partial list of frameworks to look at:

- *JSCover*—Use it for checking coverage of your JavaScript by tests. http://tntim96.github.com/JSCover/
- *Jasmin*—A very well-known BDD-style framework that I have used. I recommend it. http://pivotal.github.io/jasmine/
- *Sinon.JS*—Create fakes in JS. http://sinonjs.org/
- *CasperJS + PhantomJS*—Use this for headless testing of your browser JavaScript. That's right—no real browser needs to be alive (uses node.js under the covers). http://casperjs.org/
- *Mocha*—Also very well known and used in many projects. http://visionmedia.github.com/mocha/
- *QUnit*—A bit long in the tooth but still a good test framework. http://qunitjs.com/
- *Buster.JS*—A very new framework. http://docs.busterjs.org/en/latest/
- *Vows.js*—An up and coming framework. https://github.com/cloudhead/vows

## A.7 UI testing (desktop)

UI testing is always a difficult task. I'm not a great believer in writing unit tests or integration tests for UIs because the return is low compared to the amount of time you invest in writing them. UIs change too much to be testable in a consistent manner, as far as I'm concerned. That's why I usually try to separate all the logic from the UI into a lower layer that I can test separately with standard unit testing techniques.

There are no tools I can heartily recommend (that won't make you break the keyboard after three months) to look at in this space.

## A.8 Thread-related testing

Threads have always been the bane of unit testing. They're simply untestable. That's why new frameworks are emerging that let you test thread-related logic (deadlocks, race conditions, and so on), such as these:

- Microsoft CHESS
- Osherove.ThreadTester

I'll give a brief rundown of each tool.

### A.8.1 Microsoft CHESS

CHESS was an upcoming tool that's now somewhat open source by Microsoft on Codeplex.com. CHESS attempts to find thread-related problems (deadlocks, hangs, livelocks, and more) in your code by running all relevant permutations of threads on existing code. These tests are written as simple unit tests.

Check it out at http://chesstool.codeplex.com.

### A.8.2 Osherove.ThreadTester

This is a little open source framework I developed a while back. It allows you to run multiple threads during one test to see if anything weird happens to your code (deadlocks, for example). It isn't feature complete, but it's a good attempt at a multithreaded test (rather than a test for multithreaded code).

You can get it from my blog, at http://osherove.com/blog/2007/6/22/multithreaded-unit-tests-with-osherovethreadtester.html.

## A.9 Acceptance testing

Acceptance tests enhance collaboration between customers and developers in software development. They enable customers, testers, and programmers to learn what the software should do, and they automatically compare that to what it actually does. They compare customers' expectations to actual results. It's a great way to collaborate on complicated problems (and get them right) early in development.

Unfortunately, there are few frameworks for automated acceptance testing and just one that works these days! I'm hoping this will change soon. Here are the tools we'll look at:

- FitNesse
- SpecFlow
- Cucumber
- TickSpec

Let's take a closer look.

### A.9.1 FitNesse

FitNesse is a lightweight, open source framework that supposedly makes it easy for software teams to define acceptance tests—web pages containing simple tables of inputs and expected outputs—and to run those tests and see the results.

FitNesse is quite buggy, but it has been in use in many places with varying degrees of success. I personally haven't gotten it working quite perfectly.

You can learn more about FitNesse at www.fitnesse.org.

### A.9.2 SpecFlow

SpecFlow tries to give the .NET world what Cucumber has given the Ruby world: a tool that allows you to write the specification language as simple text files, which you can then collaborate on with your customers and QA departments.

It does a pretty good job at that. Learn more at http://www.specflow.org.

### A.9.3 Cucumber

Cucumber is a Ruby-based tool that allows you to write your specifications in a special language called Gherkin (yes, I agree). These are simple text files, and you then have to write special connector code to run actual code that acts on your application code.

It sounds complicated, but it isn't.

So what's it doing here if it's a Ruby tool? It's here because it has inspired a whole suite of tools in the .NET world, of which only one seems to be surviving right now—SpecFlow.

But there is a way to run Cucumber on .NET if you use IronRuby—a language abandoned by Microsoft and thrown to the open source world over the wall, never to be heard from again. (Great job!)

In any case, Cucumber is important enough to be aware of regardless of whether you intend to use it. It will help you understand why some things in .NET try to do the same thing.

Also, it's the basis of the Gherkin language, which other tools will try to implement now and in the future. Learn more at http://cukes.info/.

### A.9.4 TickSpec

TickSpec is for you if you use F#. I haven't used it myself, because I haven't used F#, but it's meant as a similar framework relating to acceptance and BDD-styled frameworks, as mentioned previously. I've also not heard of others using it, but that may just be because I'm not that much in F# circles. Learn more at https://tickspec.codeplex.com/.

## A.10 BDD-style API frameworks

The last few years have also given rise to a bunch of frameworks that imitate another tool from the Ruby world, called *RSpec*. This tool introduced the idea that maybe unit testing isn't a great naming convention, and by changing it to BDD we can make things more readable and perhaps even converse more with our customers about it.

To my mind, the idea of implementing these frameworks simply as different APIs in which you'd write unit or integration tests already negates most of the possibility of conversing more with your customers about them (than before), because they're not likely to really read your code or change it. I feel that the acceptance frameworks from the previous section fit more into that state of mind.

So this leaves us with just coders trying to use these APIs.

Because these APIs draw inspiration from the BDD-style language of Cucumber, in some cases they seem more readable, but to my mind, not the simple cases, which benefit more from simple assert-style tests. Your mileage may vary.

Here are some of the better-known BDD-style frameworks. I'm not creating a subsection of any of them, because I haven't personally used any of them on a real project over a long period of time:

- NSpec is the oldest and seems in pretty good shape. Learn it at http://nspec.org/.
- StoryQ is another oldie but goodie. It produces very readable output and also has a tool that translated Gherkin stories to compliable test code. Learn it at http://storyq.codeplex.com/.
- MSpec, or Machine.Specifications, tries to be as close to the source (RSpec) as possible with many lambda tricks. It grows on you. Learn it at https://github.com/machine/machine.specifications.
- TickSpec is the same idea implemented for F#. Learn it at http://tickspec.codeplex.com/.

# index

## A

## D



## E

## F

## G



## H



## I

## J



## L



## M

## N

## O



## P

## Q



## R



## S

## T

## U



## V



## W

## X

PROGRAMMING/PATTERNS

# The Art of Unit Testing SECOND EDITION

Roy Osherove

You know you *should* be unit testing, so why aren't you doing it? If you're new to unit testing, if you find unit testing tedious, or if you're just not getting enough payoff for the effort you put into it, keep reading.

**The Art of Unit Testing, Second Edition** guides you step by step from writing your first simple unit tests to building complete test sets that are maintainable, readable, and trustworthy. You'll move quickly to more complicated subjects like mocks and stubs, while learning to use isolation (mocking) frameworks like Moq, FakeItEasy, and Typemock Isolator. You'll explore test patterns and organization, refactor code applications, and learn how to test "untestable" code. Along the way, you'll learn about integration testing and techniques for testing with databases.

## What's Inside

- Create readable, maintainable, trustworthy tests
- Fakes, stubs, mock objects, and isolation (mocking) frameworks
- Simple dependency injection techniques
- Refactoring legacy code

The examples in the book use C#, but will benefit anyone using a statically typed language such as Java or C++.

**Roy Osherove** has been coding for over 15 years, and he consults and trains teams worldwide on the gentle art of unit testing and test-driven development. His blog is at ArtOfUnitTesting.com.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/TheArtofUnitTestingSecondEdition

"This book is something special. The chapters build on each other to a startling accumulation of depth. Get ready for a treat."
—From the Foreword by Robert C. Martin, cleancoder.com

"The best way to learn unit testing from what is now a classic in the field."
—Raphael Faria, LG Electronics

"Teaches you the philosophy as well as the nuts and bolts for effective unit testing."
—Pradeep Chellappan, Microsoft

"When my team members ask me how to write unit tests the right way, I simply answer: Get this book!"
—Alessandro Campeis, Vimar SpA

"The single best resource on unit testing."
—Kaleb Pederson, Next IT Corporation


ISBN 13: 978-1-617290-89-3
ISBN 10: 1-617290-89-0
54499
9 781617 290893

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]