CHAPTER 6

Essential Tools for MVC

In this chapter, I am going to look at three tools that should be part of every MVC programmer's arsenal: a dependency injection (DI) container, a unit test framework, and a mocking tool.

I have picked three specific implementations of these tools for this book, but there are many alternatives for each type of tool. If you cannot get along with the ones I use, do not worry. There are so many out there, that you are certain to find something that suits the way your mind and workflow operate.

As I noted in Chapter 3, Ninject is my preferred DI container. It is simple, elegant, and easy to use. There are more sophisticated alternatives, but I like the way that Ninject works with a minimum of configuration. If you do not like Ninject, I recommend trying Unity, which is an alternative from Microsoft.

For unit testing, I am going to be using the built-in Visual Studio testing tools. I used to use NUnit, which is a popular .NET unit-testing framework, but Microsoft has made a big push to improve the unit-testing support in Visual Studio and now includes it in the free Visual Studio editions. The result is a unit test framework that is tightly integrated into the rest of the IDE and which has actually become pretty good.

The third tool I selected is Moq, which is a mocking tool kit. I use Moq to create implementations of interfaces to use in unit tests. Programmers either love or hate Moq; there is nothing in the middle. Either you will find the syntax elegant and expressive, or you will be cursing every time you try to use it. If you just cannot get along with it, I suggest looking at Rhino Mocks, which is a nice alternative.

I introduce each of these tools and demonstrate their core features, but I do not provide exhaustive coverage of these tools. Each could easily fill a book in its own right. I have given you enough to get started and, critically, to follow the examples in the rest of the book. Table 6-1 provides the summary for this chapter.

Table 6-1. *Chapter Summary*

Problem	Solution	Listing
Decouple classes	Introduce interfaces and declare dependencies on them in the class constructor	1–9, 13–16
Automatically resolve dependencies expressed using interfaces	Use Ninject or another dependency injection container	10
Integrate Ninject into an MVC application	Create an implementation of the `IDependencyResolver` interface that calls the Ninject kernel and register it as a resolver by calling the `System.Web.Mvc.DependencyResolver.SetResolver` method	11, 12
Inject property and constructor values into newly created objects	Use the `WithPropertyValue` and `WithConstructorArgument` methods	17–20
Dynamically select an implementation class for an interface	Use an Ninject conditional binding	21, 22

(*continued*)

Table 6-1. (*continued*)

Problem	Solution	Listing
Control the lifecycle of the objects that Ninject creates	Set an object scope	23–25
Create a unit test	Add a unit test project to the solution and annotate a class file with `TestClass` and `TestMethod` attributes	26, 27, 29, 30
Check for expected outcomes in a unit test	Use the `Assert` class	28
Focus a unit test on a single feature of component	Isolate the test target using mock objects	31–34

■ **Note** This chapter assumes that you want all of the benefits that come from the MVC Framework, including an architecture that supports lots of testing and an emphasis on creating applications that are easily modified and maintained. I love this stuff, but I know that some readers will just want to understand the features that the MVC Framework offers and not get into the development philosophy and methodology. I am not going to try to convert you. It is a personal decision and you know what you need to do to deliver your projects. I suggest that you at least have a quick skim through this chapter to see what is available, but if you are not the unit-testing type, then you can skip to the next chapter and see how to build a more realistic example MVC app.

Preparing the Example Project

I am going to start by creating a simple example project, called `EssentialTools`, which I will use throughout this chapter. I used the `ASP.NET MVC Web Application` template, selected the `Empty` option and checked the box to add the basic MVC project content.

Creating the Model Classes

I added a class file to the `Models` project folder called `Product.cs` and set the content as shown in Listing 6-1. This is the model class from in previous chapters and the only change is that the namespace matches that of the `EssentialTools` project.

Listing 6-1. The Contents of the Product.cs File

```
namespace EssentialTools.Models {

    public class Product {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
}
```

I also want to add a class that will calculate the total price of a collection of Product objects. I added a new class file to the Models folder called LinqValueCalculator.cs and set the contents to match Listing 6-2.

Listing 6-2. The Contents of the LinqValueCalculator.cs File

```
using System.Collections.Generic;
using System.Linq;

namespace EssentialTools.Models {

    public class LinqValueCalculator {

        public decimal ValueProducts(IEnumerable<Product> products) {
            return products.Sum(p => p.Price);
        }
    }
}
```

The LinqValueCalculator class defines a single method called ValueProducts, which uses the LINQ Sum method to add together the value of the Price property of each Product object in an enumerable passed to the method (a nice LINQ feature that I use often).

My final model class is ShoppingCart and it represents a collection of Product objects and uses a LinqValueCalculator to determine the total value. I created a new class file called ShoppingCart.cs and added the statements shown in Listing 6-3.

Listing 6-3. The contents of the ShoppingCart.cs File

```
using System.Collections.Generic;

namespace EssentialTools.Models {
    public class ShoppingCart {
        private LinqValueCalculator calc;

        public ShoppingCart(LinqValueCalculator calcParam) {
            calc = calcParam;
        }

        public IEnumerable<Product> Products { get; set; }

        public decimal CalculateProductTotal() {
            return calc.ValueProducts(Products);
        }
    }
}
```

Adding the Controller

I added a new controller to the Controllers folder called HomeController and set the content to match Listing 6-4. The Index action method creates an array of Product objects and uses a LinqValueCalculator object to produce the total value, which I then pass to the View method. I do not specify a view when I call the View method, so the MVC Framework will select the default view associated with the action method (the Views/Home/Index.cshtml file).

Listing 6-4. The Contents of the HomeController.cs File

```
using System.Web.Mvc;
using System.Linq;
using EssentialTools.Models;

namespace EssentialTools.Controllers {
    public class HomeController : Controller {
        private Product[] products = {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        };

        public ActionResult Index() {

            LinqValueCalculator calc = new LinqValueCalculator();

            ShoppingCart cart = new ShoppingCart(calc) { Products = products };

            decimal totalValue = cart.CalculateProductTotal();

            return View(totalValue);
        }
    }
}
```

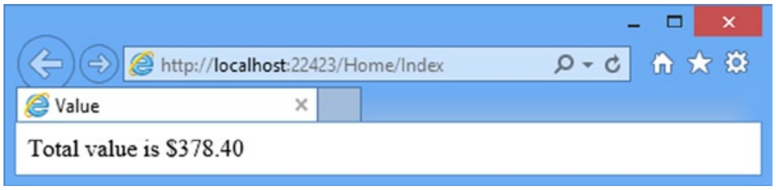
Adding the View

The last addition to the project is the view, called Index. It does not matter which options you check as you create the view as long as you set the contents to match those shown in Listing 6-5.

Listing 6-5. The Contents of the Index.cshtml File

```
@model decimal

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Value</title>
</head>
<body>
    <div>
        Total value is $@Model
    </div>
</body>
</html>
```

This view uses the @Model expression to display the value of the decimal passed from the action method. If you start the project, you will see the total value, as calculated by the LinqValueCalculator class, illustrated by Figure 6-1. This is a simple project, but it sets the scene for the different tools and techniques that I describe in this chapter.



Total value is $378.40

Figure 6-1. *Testing the example app*

Using Ninject

I introduced dependency injection (DI) in Chapter 3. To recap, the idea is to decouple the components in an MVC application, with a combination of interfaces and DI container that creates instances of objects by creating implementations of the interfaces they depend on and injecting them into the constructor.

In the sections that follow, I explain a problem I deliberately created in the example for this chapter and show how to use Ninject–my preferred DI container–to solve it. Do not worry if you find that you cannot get along with Ninject–the basic principles are the same for all DI containers and there are many alternatives from which to choose.

Understanding the Problem

In the example app, I created an example of the basic problem that DI addresses: tightly coupled classes. The ShoppingCart class is tightly coupled to the LinqValueCalculator class and the HomeController class is tightly coupled to both ShoppingCart and LinqValueCalculator.

This means that if I want to replace the LinqValueCalculator class, I have to locate and the change the references in the classes that are tightly coupled to it. This is not a problem with such a simple project, but it becomes a tedious and error-prone process in a real project, especially if I want to switch between different calculator implementations (for testing, for example), rather than just replace one class with another.

Applying an Interface

I can solve part of the problem by using a C# interface to abstract the definition of the calculator functionality from its implementation. To demonstrate this, I have added the IValueCalculator.cs class file to the Models folder and created the interface shown in Listing 6-6.

Listing 6-6. The Contents of the IValueCalculator.cs File

```
using System.Collections.Generic;

namespace EssentialTools.Models {
    public interface IValueCalculator {

        decimal ValueProducts(IEnumerable<Product> products);
    }
}
```

I can then implement this interface in the LinqValueCalculator class, as shown in Listing 6-7.

Listing 6-7. Applying an Interface in the LinqValueCalculator.cs File

```
using System.Collections.Generic;
using System.Linq;

namespace EssentialTools.Models {

    public class LinqValueCalculator : IValueCalculator {

        public decimal ValueProducts(IEnumerable<Product> products) {
            return products.Sum(p => p.Price);
        }
    }
}
```

The interface allows me to break the tight coupling between the `ShoppingCart` and `LinqValueCalculator` class, as shown in Listing 6-8.

Listing 6-8. Applying the Interface in the ShoppingCart.cs File

```
using System.Collections.Generic;

namespace EssentialTools.Models {
    public class ShoppingCart {
        private IValueCalculator calc;

        public ShoppingCart(IValueCalculator calcParam) {
            calc = calcParam;
        }

        public IEnumerable<Product> Products { get; set; }

        public decimal CalculateProductTotal() {
            return calc.ValueProducts(Products);
        }
    }
}
```

I have made some progress, but C# requires me to specify the implementation class for an interface during instantiation, which is understandable because it needs to know *which* implementation class I want to use–but it means I still have a problem in the `Home` controller when I create the `LinqValueCalculator` object, as shown in Listing 6-9.

Listing 6-9. Applying the Interface to the HomeController.cs File

```
...
public ActionResult Index() {

    IValueCalculator calc = new LinqValueCalculator();

    ShoppingCart cart = new ShoppingCart(calc) { Products = products };

    decimal totalValue = cart.CalculateProductTotal();

    return View(totalValue);
}
...
```

My goal with Ninject is to reach the point where I specify that I want to instantiate an implementation of the `IValueCalculator` interface, but the details of which implementation is required are not part of the code in the `Home` controller.

This will mean telling Ninject that `LinqValueCalculator` is the implementation of the `IValueCalculator` interface that I want it to use and updating the `HomeController` class so that it obtains its objects via Ninject, rather than by using the `new` keyword.

Adding Ninject to the Visual Studio Project

The simplest way to add Ninject to an MVC project is to use the integrated Visual Studio support for *NuGet*, which makes it easy to install a wide range of packages and keep them up to date. I used NuGet in Chapter 2 to install the Bootstrap library, but there is a huge catalog of packages available, including Ninject.

Select `Tools` ➤ `Library Package Manager` ➤ `Package Manager Console` in Visual Studio to open the NuGet command line and enter the following commands:

```
Install-Package Ninject -version 3.0.1.10
Install-Package Ninject.Web.Common -version 3.0.0.7
Install-Package Ninject.MVC3 -Version 3.0.0.6
```

The first command installs the Ninject core package and the others install extensions to the core that makes Ninject work nicely with ASP.NET applications (as I explain shortly). Do not be put off by the reference to MVC3 in the last package name–it works just fine with MVC 5.

I have used the `version` argument to install specific versions of these packages. These are the latest versions as I write this. You should use the `version` argument to ensure that you get the right results in the examples for this chapter, but you can omit the argument and get the latest (and potentially more recent) releases for real projects.

Getting Started with Ninject

There are three stages to getting the basic Ninject functionality working, and you can see all of them in Listing 6-10, which shows changes I have made to the `Home` controller.

■ **Tip** I am going to go slowly in this section and the sections that follow. Dependency Injection can take a while to understand and I do not want to skip over anything that might help reduce confusion.

Listing 6-10. Adding the Basic Ninject Functionality to the Index Action Method in the HomeController.cs File

```
using System.Web.Mvc;
using EssentialTools.Models;
using Ninject;

namespace EssentialTools.Controllers {

    public class HomeController : Controller {

        private Product[] products = {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        };

        public ActionResult Index() {

            IKernel ninjectKernel = new StandardKernel();
            ninjectKernel.Bind<IValueCalculator>().To<LinqValueCalculator>();

            IValueCalculator calc = ninjectKernel.Get<IValueCalculator>();

            ShoppingCart cart = new ShoppingCart(calc) { Products = products };

            decimal totalValue = cart.CalculateProductTotal();

            return View(totalValue);
        }
    }
}
```

The first stage is to prepare Ninject for use. To do this, I create an instance of a Ninject *kernel*, which is the object that is responsible for resolving dependencies and creating new objects. When I need an object, I will use the kernel instead of the `new` keyword. Here is the statement that creates the kernel from the listing:

```
...
IKernel ninjectKernel = new StandardKernel();
...
```

I need to create an implementation of the `Ninject.IKernel` interface, which I do by creating a new instance of the `StandardKernel` class. Ninject can be extended and customized to use different kinds of kernel, but I only need the built-in `StandardKernel` in this chapter. (In fact, I have been using Ninject for years and I have only ever needed the `StandardKernel`).

The second stage of the process is to configure the Ninject kernel so that it understands which implementation objects I want to use for each interface I am working with. Here is the statement from the listing that does that:

```
...
ninjectKernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
...
```

Ninject uses C# type parameters to create a relationship: I set the interface I want to work with as the type parameter for the `Bind` method and call the `To` method on the result it returns. I set the implementation class I want instantiated as the type parameter for the `To` method. This statement tells Ninject that dependencies on the `IValueCalculator` interface should be resolved by creating an instance of the `LinqValueCalculator` class. The last step is to use Ninject to create an object, which I do through the kernel `Get` method, like this:

```
...
IValueCalculator calc = ninjectKernel.Get<IValueCalculator>();
...
```

The type parameter used for the `Get` method tells Ninject which interface I am interested in and the result from this method is an instance of the implementation type I specified with the `To` method a moment ago.

Setting up MVC Dependency Injection

The result of the three steps I showed you in the previous section is that the knowledge about the implementation class that should be instantiated to fulfill requests for the `IValueCalculator` interface has been set up in Ninject. Of course, I have not improved my application because that knowledge remains defined in the `Home` controller, meaning that the `Home` controller is still tightly coupled to the `LinqValueCalculator` class.

In the following sections, I will show you how to embed Ninject at the heart of the MVC application, which will allow me to simplify the controller, expand the influence Ninject has so that it works across the app, and move the configuration out of the controller.

Creating the Dependency Resolver

The first change I am going to make is to create a *custom dependency resolver*. The MVC Framework uses a dependency resolver to create instances of the classes it needs to service requests. By creating a custom resolver, I can ensure that the MVC Framework uses Ninject whenever it creates an object–including instances of controllers, for example.

To set up the resolver, I created a new folder called `Infrastructure`, which is the folder that I use to put classes that do not fit into the other folders in an MVC application. I added a new class file to the folder called `NinjectDependencyResolver.cs`, the contents of which you can see in Listing 6-11.

Listing 6-11. The Contents of the NinjectDependencyResolver.cs File

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using EssentialTools.Models;
using Ninject;

namespace EssentialTools.Infrastructure {
    public class NinjectDependencyResolver : IDependencyResolver {
        private IKernel kernel;

        public NinjectDependencyResolver(IKernel kernelParam) {
            kernel = kernelParam;
            AddBindings();
        }

        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }

        private void AddBindings() {
            kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
        }
    }
}
```

The NinjectDependencyResolver class implements the IDependencyResolver interface, which is part of the System.Mvc namespace and which the MVC Framework uses to get the objects it needs. The MVC Framework will call the GetService or GetServices methods when it needs an instance of a class to service an incoming request. The job of a dependency resolver is to create that instance, a task that I perform by calling the Ninject TryGet and GetAll methods. The TryGet method works like the Get method I used previously, but it returns null when there is no suitable binding rather than throwing an exception. The GetAll method supports multiple bindings for a single type, which is used when there are several different implementation objects available.

My dependency resolver class is also where I set up my Ninject binding. In the AddBindings method, I use the Bind and To methods to configure up the relationship between the IValueCalculator interface and the LinqValueCalculator class.

Register the Dependency Resolver

It is not enough to simply create an implementation of the IDependencyResolver interface–I also have to tell the MVC Framework that I want to use it. The Ninject packages I added with NuGet created a file called NinjectWebCommon.cs in the App_Start folder that defines methods called automatically when the application starts, in order to integrate into the ASP.NET request lifecycle. (This is to provide the *scopes* feature that I describe later in the chapter.) In the RegisterServices method of the NinjectWebCommon class, I add a statement that creates an instance of my NinjectDependencyResolver class and uses the static SetResolver method defined by the System.Web.Mvc.DependencyResolver class to register the resolver with the MVC Framework, as shown in Listing 6-12. Do not worry if this does not make complete sense. The effect of this statement is to create a bridge between Ninject and the MVC Framework support for DI.

Listing 6-12. Registering the Resolver in the NinjectWebCommon.cs File

```
...
private static void RegisterServices(IKernel kernel) {
    System.Web.Mvc.DependencyResolver.SetResolver(new
        EssentialTools.Infrastructure.NinjectDependencyResolver(kernel));
}
...
```

Refactoring the Home Controller

The final step is to refactor the Home controller so that it takes advantage of the facilities I set up in the previous sections. You can see the changes I made in Listing 6-13.

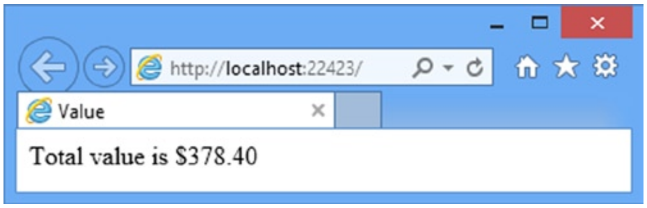
Listing 6-13. Refactoring the Controller in the HomeController.cs File

```
using System.Web.Mvc;
using EssentialTools.Models;

namespace EssentialTools.Controllers {

    public class HomeController : Controller {
        private IValueCalculator calc;
        private Product[] products = {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        };
        public HomeController(IValueCalculator calcParam) {
            calc = calcParam;
        }

        public ActionResult Index() {

            ShoppingCart cart = new ShoppingCart(calc) { Products = products };

            decimal totalValue = cart.CalculateProductTotal();

            return View(totalValue);
        }
    }
}
```

The main change I have made is to add a class constructor that accepts an implementation of the `IValueCalculator` interface, changing the `HomeController` class so that it declares a dependency. Ninject will provide an object that implements the `IValueCalculator` interface when it creates an instance of the controller, using the configuration I set up in the `NinjectDependencyResolver` class in Listing 6-10.

The other change I made is to remove any mention of Ninject or the `LinqValueCalculator` class from the controller. At last, I have broken the tight coupling between the `HomeController` and `LinqValueCalculator` class.

If you run the example, you will see the result shown in Figure 6-2. Of course, I got this same result when I was instantiating the `LinqValueCalculator` class directly in the controller.



Total value is $378.40

Figure 6-2. *The effect of running the example app*

I have created an example of *constructor injection*, which is one form of dependency injection. Here is what happened when you ran the example app and Internet Explorer made the request for the root URL of the app:

1. The MVC Framework received the request and figured out that the request is intended for the `Home` controller. (I will explain how the MVC framework figures this out in Chapter 17).

2. The MVC Framework asked my custom dependency resolver class to create a new instance of the `HomeController` class, specifying the class using the `Type` parameter of the `GetService` method.

3. My dependency resolver asked Ninject to create a new `HomeController` class, passing on the `Type` object to the `TryGet` method.

4. Ninject inspected the `HomeController` constructor and found that it has declared a dependency on the `IValueCalculator` interface, for which it has a binding.

5. Ninject creates an instance of the `LinqValueCalculator` class and uses it to create a new instance of the `HomeController` class.

6. Ninject passes the `HomeController` instance to the custom dependency resolver, which returns it to the MVC Framework. The MVC Framework uses the controller instance to service the request.

I labored this slightly because DI can be a bit mind-bending when you see it used for the first time. One benefit of the approach I have taken here is that *any* controller in the application can declare a dependency and the MVC Framework will use Ninject to resolve it.

The best part is that I only have to modify my dependency resolver class when I want to replace the `LinqValueCalculator` with another implementation, because this is the only place where I have to specify the implementation used to satisfy dependencies on the `IValueCalculator` interface.

Creating Chains of Dependency

When you ask Ninject to create a type, it examines the dependencies that the type has declared. It also looks at those dependencies to see if they rely on other types–or, put another way, if they declare their own dependencies. If there *are* additional dependencies, Ninject automatically resolves them and creates instances of all of the classes that are required, working its way along the chain of dependencies so that it can ultimately create an instance of the type you asked for.

To demonstrate this feature, I have added a file called `Discount.cs` to the `Models` folder and used it to define a new interface and a class that implements it, as shown in Listing 6-14.

Listing 6-14. The Contents of the Discount.cs File

```
namespace EssentialTools.Models {

    public interface IDiscountHelper {
        decimal ApplyDiscount(decimal totalParam);
    }

    public class DefaultDiscountHelper : IDiscountHelper {

        public decimal ApplyDiscount(decimal totalParam) {
            return (totalParam - (10m / 100m * totalParam));
        }
    }
}
```

The `IDiscountHelper` defines the `ApplyDiscount` method, which will apply a discount to a `decimal` value. The `DefaultDiscounterHelper` class implements the `IDiscountHelper` interface and applies a fixed 10 percent discount. I have modified the `LinqValueCalculator` class so that it uses the `IDiscountHelper` interface when it performs calculations, as shown in Listing 6-15.

Listing 6-15. Adding a Dependency in the LinqValueCalculator.cs File

```
using System.Collections.Generic;
using System.Linq;

namespace EssentialTools.Models {
    public class LinqValueCalculator: IValueCalculator {
        private IDiscountHelper discounter;

        public LinqValueCalculator(IDiscountHelper discountParam) {
            discounter = discountParam;
        }

        public decimal ValueProducts(IEnumerable<Product> products) {
            return discounter.ApplyDiscount(products.Sum(p => p.Price));
        }
    }
}
```

The new constructor declares a dependency on the `IDiscountHelper` interface. I assign the implementation object that the constructor receives to a field and use it in the `ValueProducts` method to apply a discount to the cumulative value of the `Product` objects.

I bind the `IDiscountHelper` interface to the implementation class with the Ninject kernel in the `NinjectDependencyResolver` class, just as I did for `IValueCalculator`, as shown in Listing 6-6.

Listing 6-16. Binding Another Interface to Its Implementation in the NinjectDependencyResolver.cs File

```
...
private void AddBindings() {
    kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
    kernel.Bind<IDiscountHelper>().To<DefaultDiscountHelper>();
}
...
```

I have created a dependency chain. My `Home` controller depends on the `IValueCalculator` interface, which I have told Ninject to resolve using the `LinqValueCalculator` class. The `LinqValueCalculator` class depends on the `IDiscountHelper` interface, which I have told Ninject to resolve using the `DefaultDiscountHelper` class.

Ninject resolves the chain of dependencies seamlessly, creating the objects it needs to resolve every dependency and, ultimately in this example, create an instance of the `HomeController` class to service an HTTP request.

Specifying Property and Constructor Parameter Values

I can configure the objects that Ninject creates by providing details of values I want applied to properties when I bind the interface to its implementation. To demonstrate this feature, I have revised the `DefaultDiscountHelper` class so that it defines a `DiscountSize` property, which I use to calculate the discount amount, as shown in Listing 6-17.

Listing 6-17. Adding a Property in the Discount.cs File

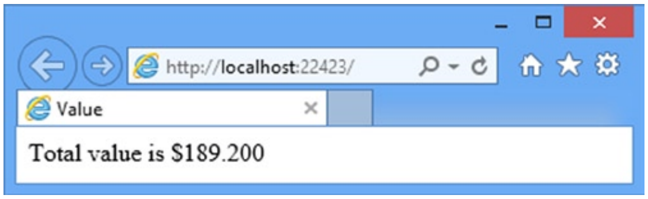
```
namespace EssentialTools.Models {

    public interface IDiscountHelper {
        decimal ApplyDiscount(decimal totalParam);
    }

    public class DefaultDiscountHelper : IDiscountHelper {

        public decimal DiscountSize { get; set; }
        public decimal ApplyDiscount(decimal totalParam) {
            return (totalParam - (DiscountSize / 100m * totalParam));
        }
    }
}
```

When I tell Ninject which class to use for an interface, I can use the `WithPropertyValue` method to set the value for the `DiscountSize` property in the `DefaultDiscountHelper` class. You can see the change I made to the `AddBindings` method in the `NinjectDependencyResolver` class to set this up, as shown in Listing 6-18. Notice that I supply the name of the property to set as a string value.

Listing 6-18. Using the Ninject WithPropertyValue Method in the NinjectDependencyResolver.cs File

```
...
private void AddBindings() {
    kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
    kernel.Bind<IDiscountHelper>()
        .To<DefaultDiscountHelper>().WithPropertyValue("DiscountSize", 50M);
}
...
```

I do not need to change any other binding, nor change the way I use the `Get` method to obtain an instance of the `ShoppingCart` class. The property value is set following construction of the `DefaultDiscountHelper` class, and has the effect of halving the total value of the items. Figure 6-3 shows the result of this change.



Total value is $189.200

Figure 6-3. *The effect of applying a discount through a property when resolving the dependency chain*

If you have more than one property value you need to set, you can chain calls to the `WithPropertyValue` method to cover them all. I can do the same thing with constructor parameters. Listing 6-19 shows the `DefaultDiscounterHelper` class reworked so that the size of the discount is passed as a constructor parameter.

Listing 6-19. Using a Constructor Property in the Discount.cs File

```
namespace EssentialTools.Models {

    public interface IDiscountHelper {
        decimal ApplyDiscount(decimal totalParam);
    }

    public class DefaultDiscountHelper : IDiscountHelper {
        public decimal discountSize;
        public DefaultDiscountHelper(decimal discountParam) {
            discountSize = discountParam;
        }

        public decimal ApplyDiscount(decimal totalParam) {
            return (totalParam - (discountSize / 100m * totalParam));
        }
    }
}
```

To bind this class using Ninject, I specify the value of the constructor parameter using the `WithConstructorArgument` method in the `AddBindings` method, as shown in Listing 6-20.

Listing 6-20. Specifying a Constructor Parameter in the NinjectDependencyResolver.cs File

```
...
private void AddBindings() {
    kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
    kernel.Bind<IDiscountHelper>()
            .To<DefaultDiscountHelper>().WithConstructorArgument("discountParam", 50M);
}
...
```

Once again, I can chain these method calls together to supply multiple values and mix and match with dependencies. Ninject will figure out what I need and create it accordingly.

■ **Tip** Notice that I did not just change the `WithPropertyValue` call to `WithConstructorArgument`. I also changed the name of the member targeted so that it matches the C# convention for parameter names.

Using Conditional Binding

Ninject supports a number of conditional binding methods that allow me to specify which classes the kernel should use to respond to requests for particular interfaces. To demonstrate this feature, I have added a new file to the `Models` folder of the example project called `FlexibleDiscountHelper.cs`, the contents of which you can see in Listing 6-21.

Listing 6-21. The Contents of the FlexibleDiscountHelper.cs File

```
namespace EssentialTools.Models {
    public class FlexibleDiscountHelper : IDiscountHelper {

        public decimal ApplyDiscount(decimal totalParam) {
            decimal discount = totalParam > 100 ? 70 : 25;
            return (totalParam - (discount / 100m * totalParam));
        }
    }
}
```

The `FlexibleDiscountHelper` class applies different discounts based on the magnitude of the total. Now that I have a choice of classes that implement the `IDiscountHelper` interface, I can modify the `AddBindings` method of the `NinjectDependencyResolver` to tell Ninject when I want to use each of them, as shown in Listing 6-22.

Listing 6-22. Using Conditional Binding in the NinjectDependencyResolver.cs File

```
...
private void AddBindings() {
    kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
    kernel.Bind<IDiscountHelper>()
        .To<DefaultDiscountHelper>().WithConstructorArgument("discountParam", 50M);
    kernel.Bind<IDiscountHelper>().To<FlexibleDiscountHelper>()
        .WhenInjectedInto<LinqValueCalculator>();

}
...
```

The new binding specifies that the Ninject kernel should use the `FlexibleDiscountHelper` class as the implementation of the `IDiscountHelper` interface when it is creating a `LinqValueCalculator` object. Notice that I have left the original binding for `IDiscountHelper` in place. Ninject tries to find the best match and it helps to have a default binding for the same class or interface, so that Ninject has a fallback if the criteria for a conditional binding are not satisfied. Ninject supports a number of different conditional binding methods, the most useful of which I have listed in Table 6-2.

Table 6-2. *Ninject Conditional Binding Methods*

Method	Effect
`When(predicate)`	Binding is used when the predicate—a lambda expression—evaluates to true.
`WhenClassHas<T>()`	Binding is used when the class being injected is annotated with the attribute whose type is specified by `T`.
`WhenInjectedInto<T>()`	Binding is used when the class being injected into is of type `T`.

Setting the Object Scope

The last Ninject feature helps tailor the lifecycle of the objects that Ninject creates to match the needs of your application. By default, Ninject will create a new instance of every object needed to resolve every dependency each time you request an object.

To demonstrate what happens, I have modified the constructor for the `LinqValueCalculator` class so that it writes a message to the Visual Studio Output window when a new instance is created, as shown in Listing 6-23.

Listing 6-23. Adding a Constructor in the LinqValueCalculator.cs File

```
using System.Collections.Generic;
using System.Linq;

namespace EssentialTools.Models {

    public class LinqValueCalculator : IValueCalculator {
        private IDiscountHelper discounter;
        private static int counter = 0;

        public LinqValueCalculator(IDiscountHelper discountParam) {
            discounter = discountParam;
            System.Diagnostics.Debug.WriteLine(
                string.Format("Instance {0} created", ++counter));
        }

        public decimal ValueProducts(IEnumerable<Product> products) {
            return discounter.ApplyDiscount(products.Sum(p => p.Price));
        }
    }
}
```

The System.Diagnostics.Debug class contains a number of methods that can be used to write out debugging messages and I find them useful when following code through to see how it works. I am, sadly, old enough that debuggers were not sophisticated when I started writing code and I still find myself going back to basics when it comes to debugging.

In Listing 6-24, I have modified the Home controller so that it demands two implementations of the IValueCalculator interface from Ninject.

Listing 6-24. Using Multiple Instances of the Calculator Class in the HomeController.cs File

```
...
public HomeController(IValueCalculator calcParam, IValueCalculator calc2) {
    calc = calcParam;
}
...
```

I do not perform any useful task with the object that Ninject provides–what is important it that I asked for two implementations of the interface. If you run the example and look at the Visual Studio Output window, you will see messages that show Ninject created two instances of the LinqValueCalculator class:

```
Instance 1 created
Instance 2 created
```

The LinqValueCalculator can be instantiated repeatedly without any problems–but that is not true for all classes. For some classes, you will want to share a single instance throughout the entire application and for others, you will want to create a new instance for each HTTP request that the ASP.NET platform receives. Ninject lets you control the lifecycle of the objects you create using a feature called a *scope*, which is expressed using a method call when setting up the binding between an interface and its implementation type. In Listing 6-25, you can see how I applied the most useful scope for MVC Framework applications: the *request scope* to the LinqValueCalculator class in the NinjectDependencyResolver.

Listing 6-25. Using the Request Scope in the NinjectDependencyResolver.cs File

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using EssentialTools.Models;
using Ninject;
using Ninject.Web.Common;
```

```
namespace EssentialTools.Infrastructure {
    public class NinjectDependencyResolver : IDependencyResolver {
        private IKernel kernel;

        public NinjectDependencyResolver(IKernel kernelParam) {
            kernel = kernelParam;
            AddBindings();
        }

        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }

        private void AddBindings() {
            kernel.Bind<IValueCalculator>().To<LinqValueCalculator>().InRequestScope();
            kernel.Bind<IDiscountHelper>()
              .To<DefaultDiscountHelper>().WithConstructorArgument("discountParam", 50M);
            kernel.Bind<IDiscountHelper>().To<FlexibleDiscountHelper>()
              .WhenInjectedInto<LinqValueCalculator>();
        }
    }
}
```

The `InRequestScope` extension method, which is in the `Ninject.Web.Common` namespace, tells Ninject that it should only create one instance of the `LinqValueCalculator` class for each HTTP request that ASP.NET receives. Each request will get its own separate object, but multiple dependencies resolved within the same request will be resolved using a single instance of the class. You can see the effect of this change by starting the application and looking at the Visual Studio Output window, which will show that Ninject has created only one instance of the `LinqValueCalculator` class. If you reload the browser window without restarting the application, you will see Ninject create a second object. Ninject provides a range of different object scopes and I have summarized the most useful in Table 6-3.

Table 6-3. *Ninject Scope Methods*

Name	Effect
`InTransientScope()`	This is the same as not specifying a scope and creates a new object for each dependency that is resolved.
`InSingletonScope()` `ToConstant(object)`	Creates a single instance which is shared throughout the application. Ninject will create the instance if you use `InSingletonScope` or you can provide it with the `ToConstant` method.
`InThreadScope()`	Creates a single instance which is used to resolve dependencies for objects requested by a single thread.
`InRequestScope()`	Creates a single instance which is used to resolve dependencies for objects requested by a single HTTP request.

Unit Testing with Visual Studio

In this book, I use the built-in unit test support that comes with Visual Studio, but there are other .NET unit-test packages available. The most popular is probably NUnit, but all of the test packages do much the same thing. The reason I have selected the Visual Studio test tools is that I like the integration with the rest of the IDE.

To demonstrate the Visual Studio unit-test support, I added a new implementation of the IDiscountHelper interface to the example project. Create a new file in the Models folder called MinimumDiscountHelper.cs and ensure that the contents match those shown in Listing 6-26.

Listing 6-26. The Contents of the MinumumDiscountHelper.cs File

```
using System;

namespace EssentialTools.Models {
    public class MinimumDiscountHelper : IDiscountHelper {

        public decimal ApplyDiscount(decimal totalParam) {
            throw new NotImplementedException();
        }
    }
}
```

My objective in this example is to make the MinimumDiscountHelper demonstrate the following behaviors:

- If the total is greater than $100, the discount will be 10 percent.
- If the total is between $10 and $100 inclusive, the discount will be $5.
- No discount will be applied on totals less than $10.
- An ArgumentOutOfRangeException will be thrown for negative totals.

The MinimumDiscountHelper class does not implement any of these behaviors yet. I am going to follow the Test Driven Development (TDD) approach of writing the unit tests and only then implement the code, as described in Chapter 3.

Creating the Unit Test Project

The first step I take is to create the unit test project, which I do by right-clicking the top-level item in the Solution Explorer (which is labeled Solution 'EssentialTools' for my example app) and selecting Add ➤ New Project from the pop-up menu.

■ **Tip** You can choose to create a test project when you create a new MVC project: there is an Add Unit Tests option on the dialog where you choose the initial content for the project.

This will open the Add New Project dialog. Select Test from the Visual C# templates section in the left panel and ensure that Unit Test Project is selected in the middle panel, as shown in Figure 6-4.

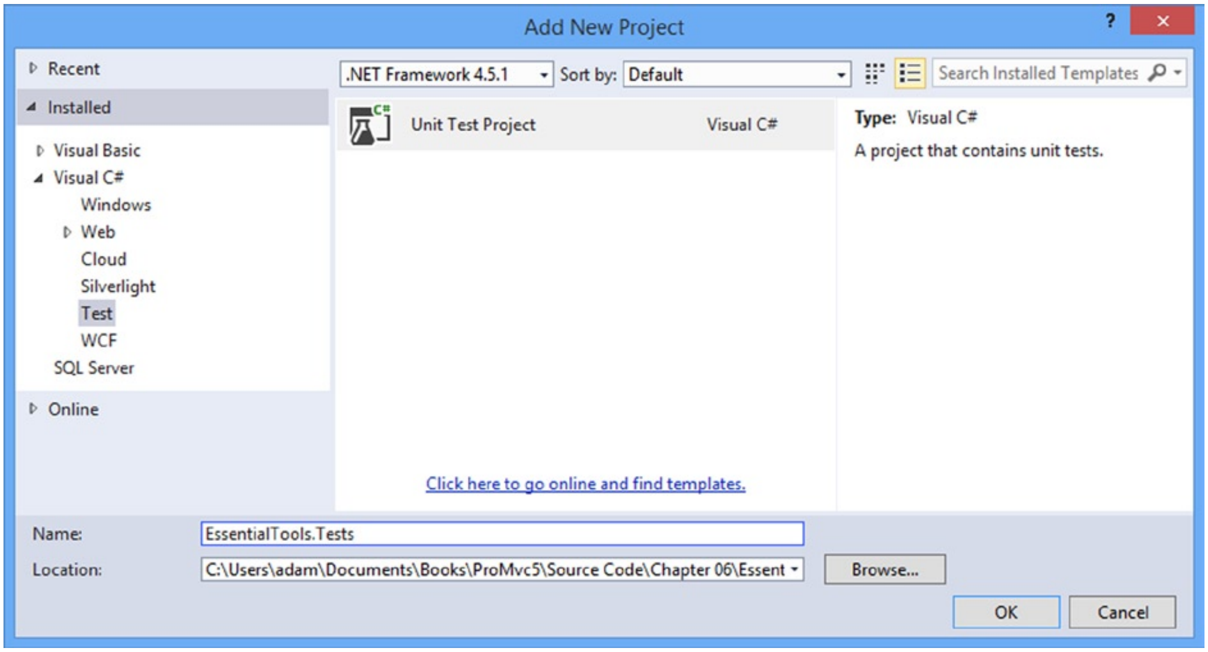

Add New Project
Recent
Installed
Visual Basic
Visual C#
Windows
Web
Cloud
Silverlight
Test
WCF
SQL Server
Online
.NET Framework 4.5.1
Sort by: Default
Search Installed Templates
Unit Test Project
Visual C#
Type: Visual C#
A project that contains unit tests.
Click here to go online and find templates.
Name: EssentialTools.Tests
Location: C:\Users\adam\Documents\Books\ProMvc5\Source Code\Chapter 06\Essent
Browse...
OK
Cancel

Figure 6-4. *Creating the unit test project*

Set the project name to `EssentialTools.Tests` and click the `OK` button to create the new project, which Visual Studio will add to the current solution alongside the MVC application project.

I need to give the test project a reference to the application project so that it can access the classes and perform tests upon them. Right-click the `References` item for the `EssentialTools.Tests` project in the Solution Explorer, and then select `Add Reference` from the pop-up menu. Click `Solution` in the left panel and check the box next to the `EssentialTools` item, as shown in Figure 6-5.

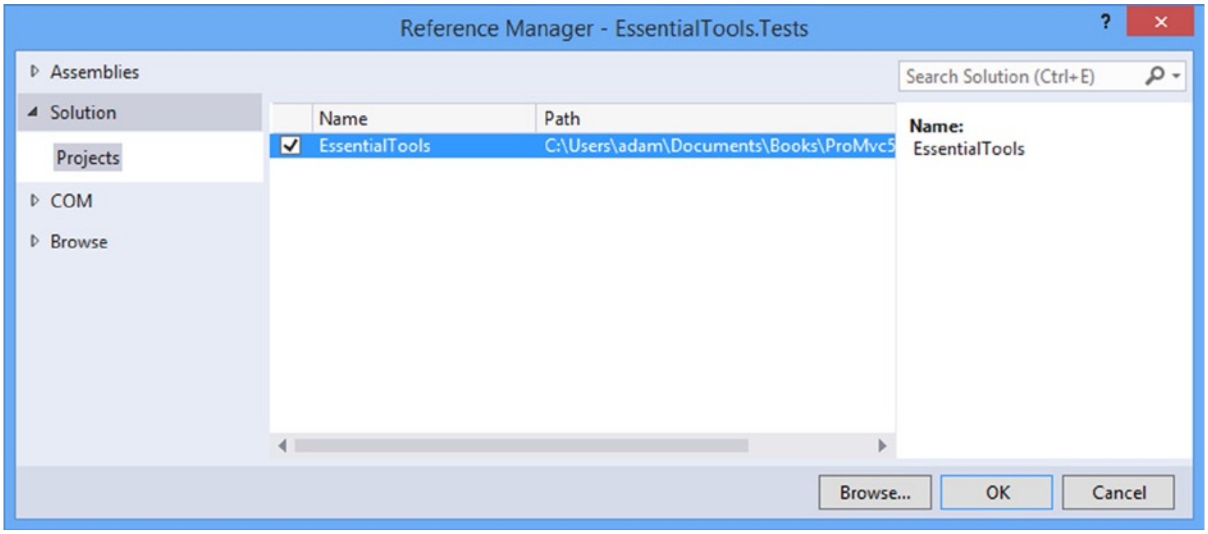


Figure 6-5. *Adding a reference to the MVC project*

Creating the Unit Tests

I will add my unit tests to the `UnitTest1.cs` file in the `EssentialTools.Tests` project. The paid-for Visual Studio editions have some nice features for automatically generating test methods for a class that are not available in the Express edition, but I can still create useful and meaningful tests. To get started, I made the changes shown in Listing 6-27.

Listing 6-27. Adding the Test Methods to the UnitTest1.cs File

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using EssentialTools.Models;

namespace EssentialTools.Tests {

    [TestClass]
    public class UnitTest1 {

        private IDiscountHelper getTestObject() {
            return new MinimumDiscountHelper();
        }

        [TestMethod]
        public void Discount_Above_100() {
            // arrange
            IDiscountHelper target = getTestObject();
            decimal total = 200;

            // act
            var discountedTotal = target.ApplyDiscount(total);

            // assert
            Assert.AreEqual(total * 0.9M, discountedTotal);
        }
    }
}
```

I have added a single unit test. A class that contains tests is annotated with the `TestClass` attribute and individual tests are methods annotated with the `TestMethod` attribute. Not all methods in a unit test class have to be unit tests. To demonstrate this, I have defined the `getTestObject` method, which I will use to arrange my tests. Because this method does not have a `TestMethod` attribute, Visual Studio will not treat it as a unit test.

■ **Tip** Notice that I had to add a `using` statement to import the `EssentialTools.Models` namespace into the test class. Test classes are just regular C# classes and have no special knowledge about the MVC project. It is the `TestClass` and `TestMethod` attributes which add the testing magic to the project.

You can see that I have followed the arrange/act/assert (A/A/A) pattern in the unit test method that I described in Chapter 3. There are countless conventions about how to name unit tests, but my advice is simply that you use names that make it clear what the test is checking. My unit test method is called `Discount_Above_100`, which is clear and meaningful to me. But all that really matters is that you (and your team) understand whatever naming pattern you settle on, so you adopt a different naming scheme if you do not like mine.

I set up the test method by calling the `getTestObject` method, which creates an instance of the object I am going to test: the `MinimumDiscountHelper` class in this case. I also define the `total` value with which I am going to test. This is the *arrange* section of the unit test.

For the *act* section of the test, I call the `MinimumDiscountHelper.ApplyDiscount` method and assign the result to the `discountedTotal` variable. Finally, for the `assert` section of the test, I use the `Assert.AreEqual` method to check that the value I got back from the `ApplyDiscount` method is 90% of the total that I started with.

The `Assert` class has a range of static methods that you can use in your tests. The class is in the `Microsoft.VisualStudio.TestTools.UnitTesting` namespace along with some additional classes that can be useful for setting up and performing tests. You can learn more about classes in the namespace at `http://msdn.microsoft.com/en-us/library/ms182530.aspx`.

The `Assert` class is the one that I use the most and I have summarized the most important methods in Table 6-4.

Table 6-4. *Static Assert Methods*

Method	Description

`AreEqual<T>(T, T)`
`AreEqual<T>(T, T, string)`
	Asserts that two objects of type `T` have the same value.

`AreNotEqual<T>(T, T)`
`AreNotEqual<T>(T, T, string)`
	Asserts that two objects of type `T` do not have the same value.

`AreSame<T>(T, T)`
`AreSame<T>(T, T, string)`
	Asserts that two variables refer to the same object.

`AreNotSame<T>(T, T)`
`AreNotSame<T>(T, T, string)`
	Asserts that two variables refer to different objects.

`Fail()`
`Fail(string)`
	Fails an assertion: no conditions are checked.

`Inconclusive()`
`Inconclusive(string)`
	Indicates that the result of the unit test cannot be definitively established.

`IsTrue(bool)`
`IsTrue(bool, string)`
	Asserts that a `bool` value is true. Most often used to evaluate an expression that returns a `bool` result.

`IsFalse(bool)`
`IsFalse(bool, string)`
	Asserts that a `bool` value is false.

`IsNull(object)`
`IsNull(object, string)`
	Asserts that a variable is not assigned an object reference.

`IsNotNull(object)`
`IsNotNull(object, string)`
	Asserts that a variable is assigned an object reference.

`IsInstanceOfType(object, Type)`
`IsInstanceOfType(object, Type, string)`
	Asserts that an object is of the specified type or is derived from the specified type.

`IsNotInstanceOfType(object, Type)`
`IsNotInstanceOfType(object, Type, string)`
	Asserts that an object is not of the specified type.

Each of the static methods in the Assert class allows you to check some aspect of your unit test and the methods throw an exception if the check fails. All of the assertions have to succeed for the unit test to pass.

Each of the methods in the table has an overloaded version that takes a string parameter. The string is included as the message element of the exception if the assertion fails. The AreEqual and AreNotEqual methods have a number of overloads that cater to comparing specific types. For example, there is a version that allows strings to be compared without taking case into account.

■ **Tip** One noteworthy member of the Microsoft.VisualStudio.TestTools.UnitTesting namespace is the ExpectedException attribute. This is an assertion that succeeds only if the unit test throws an exception of the type specified by the ExceptionType parameter. This is a neat way of ensuring that exceptions are thrown without needing to mess around with try...catch blocks in your unit test.

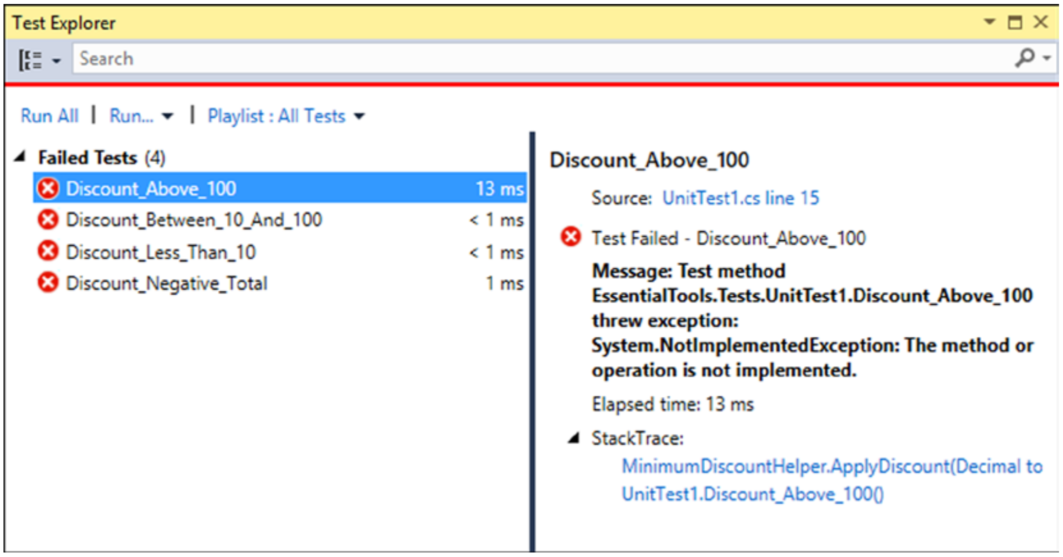
Now that I have shown you how to put together one unit test, I have added further tests to the test project to validate the other behaviors I want for my MinimumDiscountHelper class. You can see the additions in Listing 6-28, but these unit tests are so short and simple (which is generally a characteristic of unit tests) that I am not going to explain them in detail.

Listing 6-28. Defining the Remaining Tests in the UnitTest1.cs File

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using EssentialTools.Models;

namespace EssentialTools.Tests {

    [TestClass]
    public class UnitTest1 {

        private IDiscountHelper getTestObject() {
            return new MinimumDiscountHelper();
        }

        [TestMethod]
        public void Discount_Above_100() {
            // arrange
            IDiscountHelper target = getTestObject();
            decimal total = 200;

            // act
            var discountedTotal = target.ApplyDiscount(total);

            // assert
            Assert.AreEqual(total * 0.9M, discountedTotal);
        }

        [TestMethod]
        public void Discount_Between_10_And_100() {
            //arrange
            IDiscountHelper target = getTestObject();
            // act
            decimal TenDollarDiscount = target.ApplyDiscount(10);
            decimal HundredDollarDiscount = target.ApplyDiscount(100);
            decimal FiftyDollarDiscount = target.ApplyDiscount(50);

            // assert
            Assert.AreEqual(5, TenDollarDiscount, "$10 discount is wrong");
            Assert.AreEqual(95, HundredDollarDiscount, "$100 discount is wrong");
            Assert.AreEqual(45, FiftyDollarDiscount, "$50 discount is wrong");
        }

        [TestMethod]
        public void Discount_Less_Than_10() {
            //arrange
            IDiscountHelper target = getTestObject();

            // act
            decimal discount5 = target.ApplyDiscount(5);
            decimal discount0 = target.ApplyDiscount(0);

            // assert
            Assert.AreEqual(5, discount5);
            Assert.AreEqual(0, discount0);

        }

        [TestMethod]
        [ExpectedException(typeof(ArgumentOutOfRangeException))]
        public void Discount_Negative_Total() {
            //arrange
            IDiscountHelper target = getTestObject();

            // act
            target.ApplyDiscount(-1);
        }
    }
}
```

Running the Unit Tests (and Failing)

Visual Studio provides the `Test Explorer` window for managing and running tests. Select `Windows` ➤ `Test Explorer` from the Visual Studio `Test` menu to see the window and click the `Run All` button near the top-left corner. You will see results similar to the ones shown in Figure 6-6.



Failed Tests (4)

```
Discount_Above_100            13 ms
Discount_Between_10_And_100   < 1 ms
Discount_Less_Than_10         < 1 ms
Discount_Negative_Total       1 ms
```

Discount_Above_100

Source: UnitTest1.cs line 15

Test Failed - Discount_Above_100

Message: Test method EssentialTools.Tests.UnitTest1.Discount_Above_100 threw exception: System.NotImplementedException: The method or operation is not implemented.

Elapsed time: 13 ms

StackTrace:

```
MinimumDiscountHelper.ApplyDiscount(Decimal to
UnitTest1.Discount_Above_100()
```

Figure 6-6. *Running the tests in the project*

You can see the list of tests I defined in the left-hand panel of the `Test Explorer` window. All of the tests have failed, of course, because I have yet to implement the method I am testing. You can click any of the tests in the window to see details of why it has failed. The `Test Explorer` window provides a range of different ways to select and filter unit tests and to choose which tests to run. For my simple example project, however, I will just run all of the tests by clicking `Run All`.

Implementing the Feature

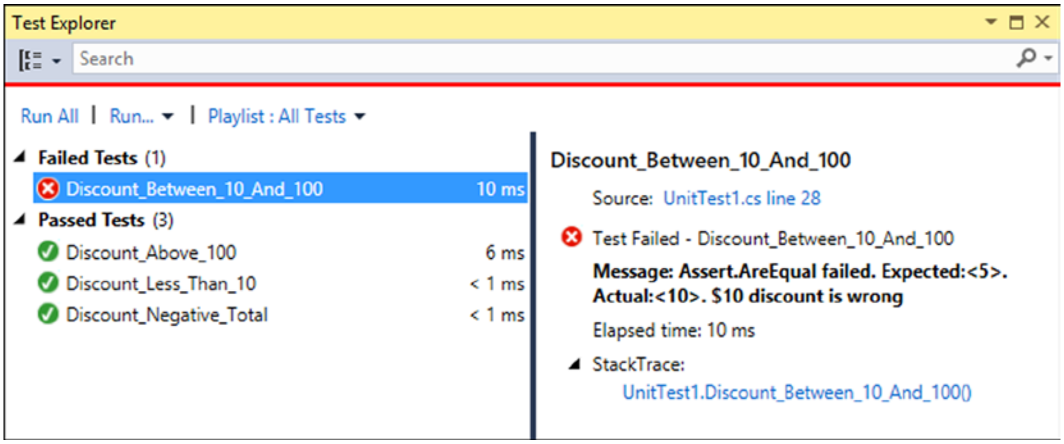
I have reached the point where I can implement the feature, safe in the knowledge that I will be able to check that the code works as expected when I am finished. For all of my preparation, the implementation of the `MinimumDiscountHelper` class is simple, as shown by Listing 6-29.

Listing 6-29. The Contents of the MinimumDiscountHelper.cs File

```
using System;

namespace EssentialTools.Models {
    public class MinimumDiscountHelper : IDiscountHelper {

        public decimal ApplyDiscount(decimal totalParam) {
            if (totalParam < 0) {
                throw new ArgumentOutOfRangeException();
            } else if (totalParam > 100) {
                return totalParam * 0.9M;
            } else if (totalParam > 10 && totalParam <= 100) {
                return totalParam -5;
            } else {
                return totalParam;
            }
        }
    }
}
```

Testing and Fixing the Code

I have left a deliberate error in the code to demonstrate how iterative unit testing with Visual Studio works and you can see the effect of the error if you click the `Run All` button in the `Test Explorer` window. You can see the test results in Figure 6-7.


Test Explorer
Search
Run All | Run... | Playlist : All Tests
Failed Tests (1)
Discount_Between_10_And_100 10 ms
Passed Tests (3)
Discount_Above_100 6 ms
Discount_Less_Than_10 < 1 ms
Discount_Negative_Total < 1 ms
Discount_Between_10_And_100
Source: UnitTest1.cs line 28
Elapsed time: 10 ms
StackTrace:
UnitTest1.Discount_Between_10_And_100()

Test Failed - Discount_Between_10_And_100

Message: Assert.AreEqual failed. Expected:<5>. Actual:<10>. $10 discount is wrong

Figure 6-7. *The effect of implementing the feature with a bug*

Visual Studio always tries to promote the most useful information to the top of the `Test Explorer` window. In this situation, this means that it displays failed tests before passed tests.

You can see that I passed three of the unit tests, but I have a problem that the `Discount_Between_10_And_100` test method detected. When I click the failed test, I can see that my test expected a result of `5`, but actually got a value of `10`.

At this point, I return to my code and see that I have not implemented my expected behaviors properly. Specifically, I do not handle the discounts for totals that are `10` or `100` properly. The problem is in this statement from the `MinumumDiscountHelper` class:

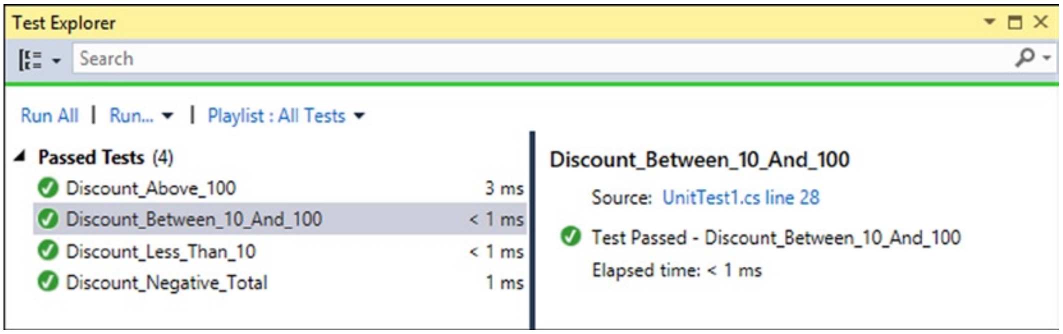
```
...
} else if (totalParam > 10 && totalParam < 100) {
...
```

The specification that I am working to implement sets out the behavior for values which are between $10 and $100 *inclusive*, but my implementation excludes those values and only checks for values which are greater than $10, excluding totals which are exactly $10. The solution is simple and is shown in Listing 6-30. Only a single character needs to be added to change the effect of the `if` statement.

Listing 6-30. Fixing the Feature Code in the MinimumDiscountHelper.cs File

```
using System;

namespace EssentialTools.Models {
    public class MinimumDiscountHelper : IDiscountHelper {

        public decimal ApplyDiscount(decimal totalParam) {
            if (totalParam < 0) {
                throw new ArgumentOutOfRangeException();
            } else if (totalParam > 100) {
                return totalParam * 0.9M;
            } else if (totalParam >= 10 && totalParam <= 100) {
                return totalParam -5;
            } else {
                return totalParam;
            }
        }
    }
}
```

When I click the `Run All` button in the `Test Explorer` window, the results show that I have fixed the problem and that my code passed all of the tests (see Figure 6-8).



```
Passed Tests (4)
  Discount_Above_100            3 ms
  Discount_Between_10_And_100   < 1 ms
  Discount_Less_Than_10         < 1 ms
  Discount_Negative_Total       1 ms

Discount_Between_10_And_100
  Source: UnitTest1.cs line 28
  Test Passed - Discount_Between_10_And_100
  Elapsed time: < 1 ms
```

Figure 6-8. *Passing all of the unit tests*

This is just a quick introduction to unit testing, and I will further demonstrate unit tests in later chapters as well. The unit test support in Visual Studio is pretty good and I recommend you explore the unit testing documentation on MSDN, which you can find at `http://msdn.microsoft.com/en-us/library/dd264975.aspx`.

Using Moq

One of the reasons that I am able to keep my tests so simple in the previous section was because I am testing a single class that depends on no other class to function. Such objects exist in real projects, of course, but you will also need to test objects that cannot function in isolation. In these situations, you need to be able to focus on the class or method you are interested in, so that you are not implicitly testing the dependencies as well.

One useful approach is to use *mock objects*, which simulate the functionality of real objects from your project, but in a specific and controlled way. Mock objects let you narrow the focus of your tests so that you only examine the functionality in which you are interested.

The paid-for versions of Visual Studio include support for creating mock objects through a feature called *fakes*, but I prefer to use a library called Moq, which is simple, easy-to-use and you can use with all Visual Studio editions, including the free ones.

Understanding the Problem

Before I get into the using Moq, I want to demonstrate the problem that I am trying to fix. In this section, I am going to unit test the `LinqValueCalculator` class, which I defined in the `Models` folder of the example project. As a reminder, Listing 6-31 shows the definition of the `LinqValueCalculator` class.

Listing 6-31. The Contents of the LinqValueCalculator.cs File

```
using System.Collections.Generic;
using System.Linq;

namespace EssentialTools.Models {

    public class LinqValueCalculator : IValueCalculator {
        private IDiscountHelper discounter;
        private static int counter = 0;

        public LinqValueCalculator(IDiscountHelper discountParam) {
            discounter = discountParam;
            System.Diagnostics.Debug.WriteLine(
                 string.Format("Instance {0} created", ++counter));
        }

        public decimal ValueProducts(IEnumerable<Product> products) {
            return discounter.ApplyDiscount(products.Sum(p => p.Price));
        }
    }
}
```

To test this class, I added a new unit test class to the test project. You do this by right-clicking the test project in the Solution Explorer and selecting `Add` ➤ `Unit Test` from the pop-up menu. If your `Add` menu does not have a `Unit Test` item, select `New Item` instead and use the `Basic Unit Test` template. You can see the changes I made to the new file, which Visual Studio names `UnitTest2.cs` by default, in Listing 6-32.

Listing 6-32. Adding a Unit Test for the ShoppingCart Class in the UnitTest2.cs File

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using EssentialTools.Models;
using System.Linq;

namespace EssentialTools.Tests {
    [TestClass]
    public class UnitTest2 {

        private Product[] products = {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        };
        [TestMethod]
        public void Sum_Products_Correctly() {

            // arrange
            var discounter = new MinimumDiscountHelper();
            var target = new LinqValueCalculator(discounter);
            var goalTotal = products.Sum(e => e.Price);

            // act
            var result = target.ValueProducts(products);

            // assert
            Assert.AreEqual(goalTotal, result);
        }
    }
}
```

The problem I face is that the `LinqValueCalculator` class depends on an implementation of the `IDiscountHelper` interface to operate. In the example, I used the `MinimumDiscountHelper` class and this presents two different issues.

First, I have made my unit test complex and brittle. In order to create a unit test that works, I need to take into account the discount logic in the `IDiscountHelper` implementation to figure out the expected value from the `ValueProducts` method. The brittleness comes from the fact that my tests will fail if the discount logic in the implementation changes, even though the `LinqValueCalculator` class may well be working properly.

Second, and most troubling, I have extended the scope of my unit test so that it implicitly includes the `MinimumDiscountHelper` class. When my unit test fails, I will not know if the problem is in the `LinqValueCalculator` or `MinimumDiscountHelper` class.

The best unit tests are simple and focused, and my current setup does not allow for either of these characteristics. In the sections that follow, I show you how to add and apply Moq in your MVC project so that you can avoid these problems.

Adding Moq to the Visual Studio Project

Just like with Ninject earlier in the chapter, the easiest way to add Moq to an MVC project is to use the integrated Visual Studio support for *NuGet*. Open the NuGet console and enter the following command:

```
Install-Package Moq -version 4.1.1309.1617 -projectname EssentialTools.Tests
```

The `projectname` argument allows me to tell NuGet that I want the Moq package installed in my unit test project, rather than in the main application.

Adding a Mock Object to a Unit Test

Adding a mock object to a unit test means telling Moq what kind of object you want to work with, configuring its behavior and then applying the object to the test target. You can see how I added a mock object to my unit test for the `LinqValueCalculator` in Listing 6-33.

Listing 6-33. Using a Mock Object in a Unit Test in the UnitTest2.cs File

```
using EssentialTools.Models;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Linq;
using Moq;
```

```
namespace EssentialTools.Tests {
    [TestClass]
    public class UnitTest2 {

        private Product[] products = {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        };

        [TestMethod]
        public void Sum_Products_Correctly() {
            // arrange
            Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>();
            mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>()))
                .Returns<decimal>(total => total);
            var target = new LinqValueCalculator(mock.Object);

            // act
            var result = target.ValueProducts(products);

            // assert
            Assert.AreEqual(products.Sum(e => e.Price), result);
        }
    }
}
```

The syntax for using Moq is a little odd when you first see it, so I will walk through each stage of the process.

■ **Tip** Bear in mind that there are a number of different mocking libraries available, so the chances are good that you can find an alternative to suit you if you do not like the way that Moq works, although Moq is actually an easy library to use. You can expect some of the other popular libraries to have manuals hundreds of pages long.

Creating a Mock Object

The first step is to tell Moq what kind of mock object you want to work with. Moq relies heavily on type parameters, and you can see this in the way that I tell Moq I want to create a mock IDiscountHelper implementation:

```
...
Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>();
...
```

I create a strongly typed Mock<IDiscountHelper> object, which tells the Moq library the type it will be handling. This is the IDiscountHelper interface for my unit test, but it can be any type that you want to isolate to improve the focus of your unit tests.

Selecting a Method

In addition to creating the strongly typed Mock object, I also need to specify the way that it behaves. This is at the heart of the mocking process and it allows you to ensure that you establish a baseline behavior in the mock object, which you can use to test the functionality of your target object in the unit test. Here is the statement from the unit test that sets up the behavior I want:

```
...
mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>())).Returns<decimal>(total => total);
...
```

I use the Setup method to add a method to my mock object. Moq works using LINQ and lambda expressions. When I call the Setup method, Moq passes me the interface that I have asked it to implement, cleverly wrapped up in some LINQ magic that I am not going to get into here. This allows me to select the method I want to configure by using a lambda expression. For my unit test, I want to define the behavior of the ApplyDiscount method, which is the only method in the IDiscountHelper interface, and the method I need to test the LinqValueCalculator class.

I also have to tell Moq what parameter values I am interested in, which I do using the It class, which I have highlighted as follows:

```
...
mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>())).Returns<decimal>(total => total);
...
```

The It class defines a number of methods that are used with generic type parameters. In this case, I have called the IsAny method using decimal as the generic type. This tells Moq to apply the behavior I am defining whenever I call the ApplyDiscount method any decimal value. Table 6-5 shows the methods that the It class provides, all of which are static.

Table 6-5. *The Methods of the It Class*

Method	Description
Is<T>(predicate)	Specifies values of type T for which the predicate will return true. See Listing 6-34 for an example).
IsAny<T>()	Specifies any value of the type T.
IsInRange<T>(min, max, kind)	Matches if the parameter is between the defined values and of type T. The final parameter is a value from the Range enumeration and can be either Inclusive or Exclusive.
IsRegex(expr)	Matches a string parameter if it matches the specified regular expression.

I will show you a more complex example later that uses some other It methods, but for the moment I will stick with the IsAny<decimal> method which allows me to respond to any decimal value.

Defining the Result

The `Returns` method allows me to specify the result that Moq will return when I call my mocked method. I specify the type of the result using a type parameter and specify the result itself using a lambda expression. You can see how I have done this for the example:

```
...
mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>())).Returns<decimal>(total => total);
...
```

By calling `Returns` method with a `decimal` type parameter (i.e., `Returns<decimal>`), I tell Moq that I am going to return a `decimal` value. For the lambda expression, Moq passes me a value of the type I receive in the `ApplyDiscount` method. I create a *pass-through* method in the example, in which I return the value that is passed to the mock `ApplyDiscount` method without performing any operations on it. This is the simplest kind of mock method, but I will show you more sophisticated examples shortly.

Using the Mock Object

The last step is to use the mock object in the unit test, which I do by reading the value of the `Object` property of the `Mock<IDiscountHelper>` object:

```
...
var target = new LinqValueCalculator(mock.Object);
...
```

To summarize the example, the `Object` property returns an implementation of the `IDiscountHelper` interface where the `ApplyDiscount` method returns the value of the `decimal` parameter it is passed.

This makes it easy to perform my unit test because I can sum the prices of my test `Product` objects myself and check that I get the same value back from the `LinqValueCalculator` object:

```
...
Assert.AreEqual(products.Sum(e => e.Price), result);
...
```

The benefit of using Moq in this way is that my unit test only checks the behavior of the `LinqValueCalculator` object and does not depend on any of the real implementations of the `IDiscountHelper` interface in the `Models` folder. This means that when my tests fail, I know that the problem is either in the `LinqValueCalculator` implementation or in the way I set up mock object, and solving a problem from either of these sources is simpler and easier than dealing with a chain of real objects and the interactions between them.

Creating a More Complex Mock Object

I showed you a simple mock object in the last section, but part of the beauty of Moq is the ability to build up complex behaviors to test different situations. In Listing 6-34, I added a new unit test to the `UnitTest2.cs` file that mocks a more complex implementation of the `IDiscountHelper` interface. In fact, I used Moq to model the behavior of the `MinimumDiscountHelper` class.

Listing 6-34. Mocking the Behavior of the MinimumDiscountHelper Class in the UnitTest2.cs File

```
using EssentialTools.Models;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using System.Linq;

namespace EssentialTools.Tests {
    [TestClass]
    public class UnitTest2 {

        private Product[] products = {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        };

        [TestMethod]
        public void Sum_Products_Correctly() {
            // arrange
            Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>();
            mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>()))
                .Returns<decimal>(total => total);
            var target = new LinqValueCalculator(mock.Object);

            // act
            var result = target.ValueProducts(products);

            // assert
            Assert.AreEqual(products.Sum(e => e.Price), result);
        }

        private Product[] createProduct(decimal value) {
            return new[] { new Product { Price = value } };
        }

        [TestMethod]
        [ExpectedException(typeof(System.ArgumentOutOfRangeException))]
        public void Pass_Through_Variable_Discounts() {
            // arrange
            Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>();
            mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>()))
                .Returns<decimal>(total => total);
            mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v == 0)))
                .Throws<System.ArgumentOutOfRangeException>();
            mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v > 100)))
                .Returns<decimal>(total => (total * 0.9M));
            mock.Setup(m => m.ApplyDiscount(It.IsInRange<decimal>(10, 100,
                Range.Inclusive))).Returns<decimal>(total => total - 5);
            var target = new LinqValueCalculator(mock.Object);
            // act
            decimal FiveDollarDiscount = target.ValueProducts(createProduct(5));
            decimal TenDollarDiscount = target.ValueProducts(createProduct(10));
            decimal FiftyDollarDiscount = target.ValueProducts(createProduct(50));
            decimal HundredDollarDiscount = target.ValueProducts(createProduct(100));
            decimal FiveHundredDollarDiscount = target.ValueProducts(createProduct(500));

            // assert
            Assert.AreEqual(5, FiveDollarDiscount, "$5 Fail");
            Assert.AreEqual(5, TenDollarDiscount, "$10 Fail");
            Assert.AreEqual(45, FiftyDollarDiscount, "$50 Fail");
            Assert.AreEqual(95, HundredDollarDiscount, "$100 Fail");
            Assert.AreEqual(450, FiveHundredDollarDiscount, "$500 Fail");
            target.ValueProducts(createProduct(0));
        }
    }
}
```

In unit test terms, replicating the expected behavior of one of the other model classes would be an odd thing to do, but it is a perfect demonstration of some of the different Moq features.

I have defined four different behaviors for the `ApplyDiscount` method based on the value of the parameter that I receive. The simplest is the catch-all, which returns a value for any `decimal` value, like this:

```
...
mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>())).Returns<decimal>(total => total);
...
```

This is the same behavior used in the previous example, and I have included it here because the order in which you call the `Setup` method affects the behavior of the mock object. Moq evaluates the behaviors in reverse order, so that it considers the most recent calls to the `Setup` method first. This means that you have to take care to create your mock behaviors in order from the most general to the most specific. The `It.IsAny<decimal>` condition is the most general condition I define in this example and so I apply it first. If I reversed the order of my `Setup` calls, this behavior would capture all of the calls I make to the `ApplyDiscount` method and generate the wrong mock results.

Mocking For Specific Values (and Throwing an Exception)

For the second call to the `Setup` method, I have used the `It.Is` method:

```
...
mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v == 0)))
    .Throws<System.ArgumentOutOfRangeException>();
...
```

The predicate I have passed to the `Is` method returns `true` if the value passed to the `ApplyDiscount` method is `0`. Rather than return a result, I used the `Throws` method, which causes Moq to throw a new instance of the exception I specify with the type parameter.

I also use the `Is` method to capture values that are greater than 100, like this:

```
...
mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v > 100)))
    .Returns<decimal>(total => (total * 0.9M));
...
```

The It.Is method is the most flexible way of setting up specific behaviors for different parameter values because you can use any predicate that returns true or false. This is the method I use most often when creating complex mock objects.

Mocking For a Range of Values

My final use of the It object is with the IsInRange method, which allows me to capture a range of parameter values:

```
...
mock.Setup(m => m.ApplyDiscount(It.IsInRange<decimal>(10, 100, Range.Inclusive)))
    .Returns<decimal>(total => total - 5);
...
```

I have included this for completeness, but in my own projects I tend to use the Is method and a predicate that does the same thing, like this:

```
...
mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v >= 10 && v <= 100)))
    .Returns<decimal>(total => total - 5);
...
```

The effect is the same, but I find the predicate approach more flexible. Moq has a range of extremely useful features and you can see how to apply them by reading the quick start provided at http://code.google.com/p/moq/wiki/QuickStart.

Summary

In this chapter, I looked at the three tools I find essential for effective MVC development: Ninject, the built-in Visual Studio support for unit testing, and Moq. There are many alternatives, both open source and commercial, for all three tools and you will not lack alternatives if you do not get along with the tools I like and use.

You may find that you do not like TDD or unit testing in general, or that you are happy performing DI and mocking manually. That, of course, is entirely your choice. However, I think there are some substantial benefits in using all three tools in the development cycle. If you are hesitant to adopt them because you have never tried them, I encourage you to suspend disbelief and give them a go, at least for the duration of this book.