Foundations of Object-Oriented Programming Using .NET 2.0 Patterns

Christian Gross

Apress®

Foundations of Object-Oriented Programming Using .NET 2.0 Patterns

Lead Editor: Jonathan Hassell
Technical Reviewer: Brian Myers
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole LeClerc
Copy Editor: Ami Knox
Assistant Production Director: Kari Brooks-Copony
Production Editor: Linda Marousek
Compositor: Susan Glinert Stevens
Proofreader: Elizabeth Berry
Indexer: Valerie Perry
Interior Designer: Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Library of Congress Cataloging-in-Publication Data

Gross, Christian.

Foundations of object-oriented programming using .NET 2.0 patterns / Christian Gross.

p. cm.

ISBN 1-59059-540-8

1. Object-oriented programming (Computer science) 2. Microsoft .NET. 3. Software patterns. I. Title.

QA76.64.G8 2005

005.1'17--dc22

2005025961

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 3

Defining the Foundation

In this chapter, I introduce two main patterns: the Bridge pattern and the Factory pattern. These two patterns provide the basis of all applications. Think of a program as a house: metaphorically speaking, the two patterns provide the foundations and the walls of that house. Without these patterns, no assembly or application would ever be maintainable. The Bridge pattern separates interface from implementation, which is useful from a maintenance and extension perspective. And the Factory pattern instantiates the types that implement the Bridge pattern.

Defining a Base for the Application

When writing code for an application, the basis of the application will be the infrastructure, namely types without an implementation. These types define an overall application and its execution without getting bogged down in the details of implementation. What you are doing when creating the types this way is called *defining an intention*. Typically, intentions are defined using interfaces, and classes that implement interfaces are generally called *components*.

In this section of the chapter, you'll learn how to define intentions and implement test-driven development as a means of creating the base for your applications.

Defining Intentions

An intention is something an application should do, without knowing the details of how to do it. An intention could be the definition of a mechanism to load a configuration. Excluded would be the details of whether that configuration could have been loaded from a database, or an XML file, or some other binary file. When defining an intention, the idea isn't to be vague, but to be as specific as possible without having to define minute details. It's possible to specifically indicate how the configuration is manipulated, and it's even possible to define the information that a configuration references. What is left as a detail for the interface implementation is the creation and wiring of configuration information to the interface instance.

The Bridge pattern[1] provides a way to express an intention and its associated implementation in technical terms. And the Factory pattern illustrates how an implementation is instantiated, but the Factory pattern returns to the caller an instance of the intention, and not

1. Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* (Boston: Addison-Wesley, 1995), p. 151.

the implementation. When implementing the Bridge and Factory patterns, the overall architecture is made up of what are known as *development components*. *Component* is a general term that describes the development of interfaces and assemblies that are distributed as pieces used to build an application. Components have the advantage that multiple parties can work together towards a common goal without having to interact with each other, except for implementing the required interfaces.

Components are useful because a boundary is defined on what can and can't be referenced. Remember from Chapter 1 how uncontrolled references represent a development problem. Components use scope declarations like `internal` and `public` constructively, which requires developers to follow a coding convention.

The general idea of the Bridge pattern is to separate intention from implementation. When a consumer uses the intention, it doesn't know about the implementation. This is done on purpose so that the implementation can be altered or updated without having to update the consumer. Consider the following source code, which is a realization of the Bridge pattern:

```
public interface Intention {
    void Echo( string message);
}

internal class Implementation : Intention {
    public void Echo(string message) {
        Console.WriteLine( "From the console " + message);
    }
}
```

The interface `Intention` defines a method `Echo` that generates an output. The interface doesn't define where the output will be generated because it's an intention of generating output. The class `Implementation` implements the interface `Intention` and provides a meaning to the intention. From the perspective of the class `Implementation`, the intention of generating output means to generate output on the console. Another implementation could define the intention to mean generating output to an e-mail.

For either implementation, the consumer only knows about the intention and that the output is being generated somewhere. The consumer of the interface `Intention` doesn't need to know how the intention is implemented—it simply uses the interface. An example of consuming the interface is as follows:

```
Intention obj = new Implementation();
obj.Echo( "hello anybody there");
```

In this example, the variable `obj` is of type `Intention`, and instantiated is the type `Implementation`. Then when calling the method `obj.Echo`, the actual method called is `Implementation.Echo`.

So far so good, but then a problem arises, and it's the reason why the Factory pattern exists. The objective of the Bridge pattern is to separate the intention from the implementation so that the consumer of the intention has no idea of the implementation type. Yet in the consumer example, the implementation type is instantiated using the `new` statement. Therefore, the consumer does know the type of the implementation, so the example includes an interface just for the sake of using patterns in the source code, which of course is silly.

The solution to the type problem is solved using the Factory pattern. Included as part of the Factory pattern is a change in scope conventions for the intention and implementation types. Look back at the definition of the interface `Intention` and class `Implementation`. The interface scope is public, and the class scope is internal. When these declarations are used in an assembly, it means that all consumers of the assembly can see the interface, but the implementation is only visible within the assembly. To get an instance of the class `Implementation`, a helper type is needed. This type is an implementation of the Factory pattern.

Following is an example Factory implementation:

```
public class Factory {
    public static Intention Instantiate() {
        return new Implementation();
    }
}
```

The class `Factory` scope is public, and it contains a single `static` method, `Instantiate`. The method `Instantiate`, when called, creates a new instance of the class `Implementation`, but returned is an `Intention` interface instance. When a consumer calls the method `Instantiate`, an interface instance is returned, without the consumer knowing the type of the interface instance. The consumer code is rewritten as follows:

```
Intention obj = Factory.Instantiate();
obj.Echo( "hello anybody there");
```

The consumer code calls the method `Factory.Instantiate`, which returns an interface instance of `Intention`. The consumer code uses patterns and is a good example of writing the correct code. Yet, you may have the gnawing feeling that this example passes the buck. To instantiate the type `Implementation`, the method `Factory.Instantiate` will always have to be called. So instead of using the `new` keyword, a method is used; this is trading one type for another type, which is known as *using a helper type*.

The reason why it's better to use a helper type is because it makes it simpler to update the implementation of the interface. If the consumer knows about the implementation types, then when the implementation types are updated, the consumer might need to be updated. If the consumer only knows about the interfaces and uses a helper type to instantiate the implementations, then an update in the implementations will never need an update in the consumer. The helper type reference remains identical.

Implementing Test-Driven Development

As demonstrated in Chapter 2, writing tests is as important as writing components. To test the interface `Intention`, you could write a test like this one:

```
 [TestFixture]
public class IntroTests {
    [Test]public void SimpleBridge() {
        Intention obj = Factory.Instantiate();
        obj.Echo( "hello anybody there");
    }
}
```

Running the test class results in the following output:

```
centaur:~/src/bin cgross$ mono ../../apps/nunit/bin/nunit-console.exe chap03.exe
NUnit version 2.2.0
Copyright (C) 2002-2003 James W. Newkirk, Michael C. Two, Alexei A. Vorontsov,
Charlie Poole.
Copyright (C) 2000-2003 Philip Craig.
All Rights Reserved.

OS Version: Unix 7.7.0.0    Mono Version: 1.1.4322.573

.From the console hello anybody there

Tests run: 1, Failures: 0, Not run: 0, Time: 0.312191 seconds
```

NUnit reports that a single test was run, and there were no failures. The output of the `Implementation` class is dumped to the same output stream as the one where the NUnit output is generated. What is interesting about this test is that it's extremely useless. Sure, you can cross off the writing and execution of a test from your to-do list, but the interface and implementation haven't been tested.

Implementing a proper test isn't that simple for the `Implementation` type because the type generates an output that isn't captured by the test. Essentially, the problem is how to generate feedback to the testing application that indicates that the `Implementation` type did everything correctly. In test-driven development terminology, the solution is to create a mock object.

A *mock object* behaves like the object it's mimicking, but it doesn't actually perform the actions. For example, in a database scenario, such an object would verify a particular SQL statement and not execute the SQL statement. The mock object for the `Implementation` class is the type `Console`. The `Console` type was chosen as a mock object because it's a lower-level class than `Implementation`. In general, mock objects are created for the lower-level called classes. Mock objects enable a subsystem to be tested in isolation.

Creating a mock object for the `System.Console` class is a bit tricky because it means replacing a method call from the standard .NET libraries. The solution is to use namespaces and aliases. The mock object needs to reside in its own namespace as shown in this example:

```
namespace Chap03MockObjects {
    public class Console {
        public Console() { }
        public static void WriteLine( string message) {
            System.Console.WriteLine( "-" + message + "-");
        }
    }
}
```

The class `Console` lies within the namespace `Chap03MockObjects`, which ensures no conflict will happen with `Console` in the namespace `System`. For the time being, the output of the mock object will be generated on the console. This means the `System.Console` class is called again.

■ **Caution** Use the full namespace identifier `System.Console`, otherwise the `Console.WriteLine` will be redirected to `Chap03MockObjects.Console`.

The output is generated on the console to illustrate how it's possible to make a mock object appear like the real object, which then redirects to the real object.

To use the mock class instead of the class `System.Console` within the class `Implementation` requires an alias. The alias will cause the type `Console` to be identified as another type, which may or may not be related to the original type. Following is an alias declaration for this type:

```
using System;
using Console = Chap03MockObjects.Console;
```

The `System` namespace is still included, but the `Console` type is redirected to be the type `Chap03MockObjects.Console`. This means whenever the `Console` type is called, the mock object is called. It's advisable to encapsulate `Console` in an `if` statement that is only included in a debug build or a test build. To accomplish this, the example can be modified like so:

```
using System;
#if TEST_BUILD
using Console = Chap03MockObjects.Console;
#endif
```

Mock objects are an important part of writing applications, as they make it possible to test individual types or subsystems in isolation. Mock objects don't need to be written for every type, just for those at the lowest level. For example, let's say that type A references type B, which references type C. Whatever type C references will need a mock object. When testing type B, there doesn't need to be a mock object for type C. It's assumed that type C will be tested before type B, and therefore type C is considered programmatically correct.

A scenario does exist in which there would be a mock object for type C. If two different teams are developing types B and C, then potentially the team that implements type B might need a mock object of type C. The reason is the team of type B will only get paid if their implementation works. The team won't want to rely on type C as that means testing a type they didn't implement. The team that implements type C could be lazy and say privately that they will let the team implementing type B test their work, saving them from testing type C. Hence, if the team working on type B implements their own mock objects, they aren't dependent on the code of the team working on type C and can get paid once their work is complete.

Mock objects are created for the following scenarios:

- *You need references to types that aren't under your control*: This is part of the internal quality assurance process ensuring that an application isn't working because it isn't working correctly, not because the dependencies are working incorrectly.
- *You need isolation of types to test a set of functionality to simplify testing of the types*: Some subsystems are so complex that the only way to write tests that can verify the correctness of such subsystems is to write mock objects.

- *You would like to simplify the test environment*: For example, testing the type `Implementation` without a mock object is very difficult as there is no direct feedback. The only way to get feedback is to implement more complicated scripts.
- *You want to use recursive type references*: Consider the scenario where a subsystem requires a delegate callback. The testing environment would have to provide mock objects to successfully test the subsystem. Another scenario is when `Assembly1.A` calls `Assembly2.B`, which calls `Assembly1.C`. It isn't possible to test `Assembly1.A` without testing at the same time `Assembly2.B` and vice versa. The only solution is to test `Assembly2.B` and create a mock object for `Assembly1.C`.

Going back to the `System.Console` example, the mock object `Chap03MockObjects.Console` doesn't need to contain all of the functionality of `System.Console` to provide a meaningful test environment. The purpose of the `Chap03MockObjects.Console` is to implement only as much functionality as necessary to make the tests do something meaningful. The class `Chap03MockObjects.Console` is incomplete in that the output is still output to the console, and not verified. To perform a verification, a callback needs to be added. Consider the following source code as a rewritten mock object that has built-in callback facilities:

```
namespace Chap03MockObjects {
    public delegate void FeedbackString( string message);

    class NoCallbackDefinedException : ApplicationFatalException {
        public NoCallbackDefinedException() : base( "No callback is defined") { }
    }

    public class Callback {
        private static FeedbackString _feedback;

        public static FeedbackString CBFeedbackString {
            get {
                if( _feedback == null) {
                    throw new NoCallbackDefinedException();
                }
                return _feedback;
            }
            set {
                _feedback = value;
            }
        }
    }

    public class Console {
        public Console() { }
        public static void WriteLine( string message) {
            Callback.CBFeedbackString( message);
        }
    }
}
```

The general idea in the rewritten classes is to define a delegate that will be implemented by the test infrastructure, which the mock object can call into. The delegate `FeedbackString` pipes the content written to the method `Console.WriteLine` back to whomever implements the delegate. The delegate is stored and referenced in the class property `Callback.CBFeedbackString`. If the property `CBFeedbackString` is referenced without having a valid delegate assigned to the variable `_feedback`, the exception `NoCallbackException` is thrown. This is done on purpose because the property should never be referenced if there is no valid delegate value.

The test needs to be rewritten and a callback needs to be provided so that a complete feedback loop is created, as shown here:

```
 [TestFixture]
public class IntroTests {
    private string _strHelloAnybodyThere =  "hello anybody there";

    [Test]public void SimpleBridge() {
        Intention obj = Factory.Instantiate();
        Chap03MockObjects.Callback.CBFeedbackString =
            new Chap03MockObjects.FeedbackString( this.CallbackSimpleBridge);
        obj.Echo( _strHelloAnybodyThere);
    }
    void CallbackSimpleBridge( string message) {
        string test = "From the console " + _strHelloAnybodyThere;
        if( message != test) {
            throw new Exception();
        }
    }
}
```

The class `IntroTests` has an additional method, `CallbackSimpleBridge`, that is the delegate the `FeedbackString` method used to provide the feedback. In the method `SimpleBridge`, the interface instance `Intention` is created using the factory method `Factory.Instantiate`. Then the delegate is assigned to the property `CBFeedbackString` to complete the structural parts of the feedback loop. Finally, the method `obj.Echo` can be called.

When the method `obj.Echo` is called, the method `CallbackSimpleBridge` is called in turn; this results in the parameter message being tested against the variable test, which is the combination of some text in the string buffer `_strHelloAnybodyThere`. If the buffers are identical, then the test is successful, but if the data isn't identical, an exception is generated. A special exception type doesn't need to be used because the NUnit framework will catch any generated exception. When NUnit catches the exception, the appropriate error messages are generated.

The feedback loop is completed, and it's possible to effectively test a method that seemed untestable. Overall, the callback mechanism is very easy to understand and implement. The callback mechanism provides a method whereby the higher-level testing infrastructure can interact with the lower-level types. Without the feedback loop, it isn't easy to discover whether or not a test has succeeded.

The callback mechanism also needs to be considered very carefully. In the example, a delegate was used, but that isn't the only possible technique. The class `Chap03MockObjects.Console` could have contained some properties that represented the data that was generated. The calling test would reference the properties and verify that everything is OK. If there are data

consistency problems, then an exception is generated. There is no advantage or disadvantage to using either a callback or populated properties. To a large degree, it's the choice of the test writer.

Now that you have an understanding of how to define the base of your application and a general idea of how the Bridge and Factory patterns are involved, let's take a closer look at these patterns, starting with the Bridge pattern.

Implementing the Bridge Pattern

The big picture concept of the Bridge pattern is to separate implementation from intention. But the philosophical question is, Why do you want that? One of the big problems in the software industry is change. It isn't that change is bad, just that change is a fact of life. What is bad is change that is disruptive.

Many architects and developers try to minimize change or make it less disruptive by future-proofing an application or a design. To *future-proof* is to add features to an application that will protect it from early obsolescence. Often in discussions about future-proofing, you'll hear buzzwords like platform neutral, interoperability, language neutral, location transparent, and flexible application environment.

The problem with future-proofing software is that you can only ever achieve limited success. Let me give an example from my personal life. In the late 1990s, I was consulting for a customer and providing them with Visual Basic architectural guidance. The customer's language of choice was Smalltalk, and often I had meetings with higher-level managers who attempted to convince me of the virtues of Smalltalk. I'm not picking on Smalltalk, as Smalltalk is fine language. It was just the way that Smalltalk was sold was wrong. The customer built a huge framework in an attempt to future-proof their applications. The framework was started in the early 1990s, almost done in late in that decade, and cost seven digits. What happened? Java came along and caused the client to throw all of the work out the window and restart with Java.

Future-proofing in literal terms means being certain of the future, which is impossible. We should change the term *future-proofing* to fit an alternative outlook: closing the absolute minimum number of doors for the future. It's simply impossible to future-proof software, as decisions will have to be made and those decisions will eventually need to be locked in.

For example, writing an application in .NET means using .NET. Whatever code you write won't work in Java because Java uses its own runtime. It could be debated that, using J#, it's possible to write code that works in both .NET and Java. However, the obstacles are very high, and the code uses the lowest common denominator libraries and coding techniques. Instead, the objective should be to code using .NET, and write code that is CLS compliant and executes on multiple platforms. Using such a strategy, it's at least possible to be flexible and potentially shift operating systems when necessary. But in the end, the application is and always will be .NET, and if .NET were to be terminated, there might be a problem. Where the future-proofing comes into play is the fact that there is Mono and it's open source.

The main objective with future-proofing is to make decisions that will keep as many doors open as possible using reasonable means. With this in mind, let's look at how you should approach defining interfaces.

Keeping Options Open Using Interfaces

When defining interfaces, the objective should be to solve the task at hand, no more, no less. Many developers attempt to future-proof interfaces by adding features that could potentially be used in the future. The problem with adding such features is that they need to be tested and documented. The reality is features will be added, not properly tested, not properly used, and become dead code that has to be maintained in the future. By keeping interfaces to the minimal requirements, it's possible to solve the problem without getting bogged down in extra work.

For example, from Chapter 2, recall that the type `Mathematics` was defined as well as a single method, `Add`, that calculated everything using the type `int`. A good way to future-proof the type would be to not use the type `int`, but use some general type. An example of using a general type is to employ Generics, as illustrated here:

```
public interface IMathematics< numbertype> {
    numbertype Add( numbertype param1, numbertype param2);
}
```

The interface `IMathematics` is defined using a Generic data type, which allows you to define the specific data type at a later point in time. An implementation of the `IMathematics` interface is as follows:

```
internal class IntMathematicsImpl : IMathematics< int> {
    public int Add(int param1, int param2) {
        checked {
            return param1 + param2;
        }
    }
}
```

The class `IntMathematicsImpl` implements the interface `IMathematics` and defines the data type to be an `int`. Then, like the original implementation of `Mathematics`, the two integers are added in the context of `checked` block.

A factory that instantiates the class `IntMathematicsImpl` is defined as follows:

```
public class FactoryIMathematics {
    public static IMathematics< int> Instantiate() {
        return new IntMathematicsImpl();
    }
}
```

The method `Instantiate` returns a specialized form of the `IMathematics` interface that uses the `int` data type. The code within the method `Instantiate` instantiates a new instance of the class `IntMathematicsImpl`.

Too Much of a Good Thing

The interface that used Generics to keep options open allows different types to be manipulated. A developer might come up with the idea to convert the implementation and factory to use Generics, thus making it possible to manipulate all types without having to explicitly provide

an implementation for the type. The following example illustrates how the factory could be converted into a Generic type:

```
public class FactoryIMathematics< basetype> {
    public static IMathematics< basetype> Instantiate() {
        return new IntMathematicsImpl< basetype>();
    }
}
```

The class `FactoryIMathematics` is converted into a Generic type, and the method `Instantiate` uses the Generic type, as does the class `IntMathematicsImpl`. Following is the Generic type `IntMathematicsImpl`:

```
internal class IntMathematicsImpl< basetype> : IMathematics< basetype> {
    public basetype Add( basetype param1, basetype param2) {
        checked {
            return param1 + param2;
        }
    }
}
```

The converted `IntMathematicsImpl` class is based entirely on using a Generic type. In theory, it's possible to use any data type, and through the "magic" of Generics, it's possible to perform add operations on any type. The problem is that it isn't possible to magically add any data type. This is an example of abusing Generic types. The class `IntMathematicsImpl` won't compile because C# has no idea how to translate the expression `param1 + param2`, as the compiler doesn't understand how to add two unknown types together. This means it isn't possible to use Generics to add two unknown types. A developer might then come up with the idea of abstracting the `Add` method to use another interface that a type must implement à la .NET constraints.

Using constraints, the Generic implementation knows how to add two unknown data types. This solution would work, but it's a delegation, which adds complexity. Remember, the original interface generically defined how to perform mathematics in a general fashion. The added complexity in the example might be obvious, but it happens because one idea leads to another idea, which leads to yet another idea.

It's very easy to overuse Generics, as it's so tempting to define everything in abstract terms. You might want to think that, once those abstract terms have been defined, magically plugging in a data type will result in a working system. The problem is that it's impossible to do that. Adding methods or properties to make the Generic types work is adding complexity. Remember the focus of the Bridge pattern: to separate intention from implementation. An intention can and needs to be described in abstract terms, but an implementation should be specific.

Another danger of attempting to define in abstract terms an addition is that adding integers isn't like adding complex numbers, floating-point numbers, or even different bases. The result is that the different types share the same intention of adding, but their implementations will be entirely different.

What About .NET 1.x?

Without the ability to use Generics, it isn't easily possible to define generic interfaces that can be applied with multiple data types. In essence, it's easy to say that is why Generics were added

to .NET 2.0 in the first place. However, it still doesn't help those developers who are using .NET 1.x.

The simplest solution to the Generics problem is to use the `object` type as follows:

```
public interface IMathematics {
    object Add( object param1, object param2);
}
```

In the declaration of the `IMathematics` interface, the method `Add` uses the type `object` for all of the types. As the type `object` is the basis for all other types, it's possible to define multiple implementations that use different data types. Following is an example implementation that adds two integers:

```
public class IntMathematicsImpl : IMathematics {
    public object Add(object param1, object param2) {
        checked {
            return (int)param1 + (int)param2;
        }
    }
}
```

In the implementation of the `Add` method for the class `IntMathematicsImpl`, the types are converted from `object` to `int`. If the type can't be converted from `object` to `int`, then a typecast exception will be generated. This means another test case must be added to determine whether an exception is thrown. After the typecast, the numbers are added together and returned to the caller as an `object` type. The answer will be correct, as the code is programmatically correct.

Looking at the solution, it's probably easier to stick with the `int` data type. The reason is because the boxing conversion from `int` to `object` and then back to `int` is too expensive. The working code is fairly expensive in computational terms when executed many times. This would mean that individual interfaces for each type (`int`, `long`, `float`, etc.) would have to be defined and implemented. However, often native data types aren't used, and in that case it's absolutely acceptable to use the `object` type. In fact, the list classes in .NET 1.x use that strategy.

This wraps up our look at the details of Bridge pattern implementation. Next up, you get a chance to see variations of this pattern.

Bridge Pattern Implementation Variations

The Bridge pattern has many facets, all of which relate to each other. For example, when defining reusable classes, often the better approach is to use not interfaces, but classes. What will be defined in this section are the different variations and the contexts that they apply to. Note that some of the examples may bear resemblance to patterns defined later. Not all examples are similar to patterns, and it's important to get a feeling for how the Bridge pattern can be used.

Implementing Application Logic

One purpose of the Bridge pattern is to simplify application logic. The interfaces are combined with another type to perform a particular operation. The type that does the combining is called a *controller*. The controller represents the abstract application logic. It's important to realize

that the controller only performs actions, as illustrated by the following example of adding a list of numbers:

```
public class Operations {
    private IMathematics _math;

    public IMathematics Math {
        get {
            if( _math == null) {
                throw new PropertyNotDefined( "Operations.Math");
            }
            return _math;
        }
        set {
            _math = value;
        }
    }
    public int AddArray( int[] numbers) {
        int total = 0;

        foreach( int number in numbers) {
            total = this.Math.Add( total, number);
        }
        return total;
    }
}
```

The class `Operations` is used to perform a higher-level operation that utilizes the lower-level interface `IMathematics`. The `Operations` class is considered a controller because it does something with the lower-level interface. Note that the example uses a single interface instance, but there could be multiple interface instances involved.

The `Operations` class doesn't instantiate the interface instance, but exposes a property `Math` that some other type must assign. The reason why the `Operations` class doesn't instantiate an interface instance is for maximum flexibility. The focus of the controller class is to perform some operations using the interfaces provided. It's also expected that the controller doesn't consider specifics, as it's like the interface intention in that the controller implements an abstract application process logic.

The method `AddArray` then implements the process by using the interface instance. The purpose of the `AddArray` method is to add integer values to generate a grand total that is then returned to the caller. The array parameter `numbers` is iterated using a `foreach` loop that sequentially calls the method `obj.Add`, where the first parameter is a running total and the second parameter is a value from the array of numbers.

To use the `Operations` class, I've written the following test:

```
 [Test]public void AddListDotNet1() {
    Operations ops = new Operations();
    ops.Math = Factory.Instantiate();

    int[] values = new int[] {1,2,3,4,5};

    Assert.AreEqual( 15, ops.AddArray( values), "List did not add");
}
```

The class `Operations` is instantiated using the `new` statement and assigned to the variable `ops`. Then the `Operations.Math` property is assigned using the `Factory.Instantiate` method. After that, an array of integer values is created and passed to the `Operations.Add` method. The `Assert.AreEqual` method tests to make sure that the list is properly added.

What is confusing in the test code is that the `Operations` class wasn't instantiated using a factory. A factory is generally not necessary because the `Operations` class is a controller, and there will only ever be one definition and one implementation. Consider it as follows: the controller only uses interfaces and represents an abstracted logic. Typically, there is only one type of application logic, and hence using an interface and factory complicates a scenario that doesn't need complications. In some instances, the controller will be defined using an interface, and these instances will be discussed later. What I want to point out here is that you shouldn't feel guilty using a class declaration for a controller. It's acceptable so long as the controller uses general types.

Having read all that regarding the direct instantiation of a class, note that the class `Operations` is specific in that the `int` type is used to add numbers. Ideally, the controller class shouldn't be type specific. A solution would be to use the redefined `IMathematics` interface that uses the type `object`. Using that interface definition then, the `Operations.AddArray` method would be rewritten as follows:

```
public object AddArray( object[] numbers) {
    object total = this.Math.Reset();
    foreach( object number in numbers) {
        total = this.Math.Add( total, number);
    }
}
```

In the rewritten `AddArray` method, there is no type defined, and every value referenced is of type `object`. The rewritten method is truly abstract and can add any type of objects. But to make the addition work properly, an additional method needs to be included in the `IMathematics` interface. This method, `Reset`, is used to return a default empty value. An empty object is necessary because `AddArray` is adding an unknown type to another unknown type. The variable total can't be assigned a value of zero, because `total` is an object, and .NET requires assigning `total` something. Using the value null is not appropriate because, as specified in Chapter 2, a null object isn't an empty object, and all methods should return an empty object.

The rewritten `AddArray` method works, but as mentioned previously, based on the type `object`, there is a performance cost. The performance cost is due to boxing and unboxing of the types from a native type to an object instance. Using .NET Generics, it's possible to solve this problem, but then it's necessary to use .NET 2.0. Of course, not all operations are math operations, and using the `object` type is most likely acceptable.

Using .NET Generics, the Operations class is implemented as follows:

```
public class Operations< type> {
    private IMathematics< type> _math;

    public IMathematics<type> Math {
        get {
            if( _math == null) {
                throw new PropertyNotDefined( "Operations.Math");
            }
            return _math;
        }
        set {
            _math = value;
        }
    }
    public type AddArray( type[] numbers) {
        type total = this.Math.Reset();

        foreach( type number in numbers) {
            total = this.Math.Add( total, number);
        }
        return total;
    }
}
```

In the .NET Generics version of the Operations class, the type has become a parameter that is the base type for the interface IMathematics. The method AddArray has no types and only uses the type employed to define the Operations class. Notice with .NET Generics the Operations class is a pure abstract application logic class, and is type safe.

The following test uses the .NET Generics version of the Operations class:

```
 [Test]public void AddListDotNet2() {
    MathBridgeDotNet2.Operations< int> ➥
 ops = new MathBridgeDotNet2.Operations< int>();
    ops.Math = MathBridgeDotNet2.Factory.Instantiate();

    int[] values = new int[] {1,2,3,4,5};

    Assert.AreEqual( 15, ops.AddArray( values), "List did not add");
}
```

In this test version, the same methods and instantiations are used. The only difference is that the type int is used to convert a Generic type to a specific type.

Inspecting the code closer, a potential additional optimization would be to instantiate the types using a factory that is parameterized using a type. An example of such an optimization would be as follows:

```
ops.Math = MathBridgeDoNet2.Factory.Instantiate<int>();
```

In this example, the `Factory.Instantiate` method has an additional type identifier that would be used to instantiate the correct implementation type. The solution is elegant and rather clever. Consider the following implementation:

```
public class Factory {
    public static IMathematics Instantiate<type>() {
        if( typeof( type) is int) {
            return new ImplIMathematics();
        }
    }
}
```

In the implementation of `Instantiate`, an `if` block tests which type the Generic .NET parameter is. Then, depending on the type, the appropriate implementation is instantiated. The advantage of this approach is that the `Factory` class can be extended to support new types without needing the client to be recompiled or changed.

Controller Interfaces

In general, the controller class is defined as a class, but sometimes the controller class needs to be defined as an interface. There are several reasons for doing this, and the most important relate to flexibility and convenience.

Let's start out by focusing on the flexibility issue. Imagine the controller is implementing a taxation system. The general intention of taxation is similar regardless of the country. What's different are the details of calculating the deductions, amortizations, etc. Some developers might be tempted to abstract the details as parameters that can be activated to determine which calculations are performed by the controller. The controller class would then need to be constantly extended to support yet another taxation system. Such an approach is futile and leads to complicated and messy code that should be avoided.

When a controller class needs to switch personality, an interface should be defined for a taxation system such as the following:

```
namespace ITaxation {
    public interface IIncomes {
    }
    public interface IDeductions {
    }
    public interface ITaxation {
        IIncomes [] Incomes { get; set; }
        IDeductions [] Deductions { get; set; }

        Decimal CalculateTax();
    }
}
```

The interface `ITaxation` has two properties, `Incomes` and `Deductions`, and a single method, `CalculateTax`. The properties are assigned the incomes and deductions of the individual. Instead of properties, you might think another solution would be to convert the properties into parameters for the method `CalculateTax`; but that would be incorrect, because the taxation system might

want to perform multiple taxation calculations. Hence, the data is best defined as properties, since the properties are central to the calculation of taxes for the `ITaxation` interface. As a rule of thumb, if the data is needed in more than two methods, then you should create a property.

Let's ignore the implementations of the `IIncomes` and `IDeductions` for the time being and consider them as general implementations that will be instantiated and assigned somewhere. Instead, let's focus on the `ITaxation` interface. Unlike the previous controller definition, there are multiple controller implementations. Each controller implementation would be specific to a country and its tax structure. Following is an example implementation:

```
internal class SwissTaxes : ITaxation {
    public IIncomes[] Incomes {
        get { return null; }
        set { ; }
    }
    public IDeductions[] Deductions {
        get { return null; }
        set { ; }
    }
    public Decimal CalculateTax() {
        return new Decimal();
    }
}
```

The class `SwissTaxes` implements all of the methods. Again, ignore the properties, but consider the method `CalculateTax`. The method `CalculateTax` would be a custom implementation of the Swiss tax code. If the class were `AmericanTaxes`, the method `CalculateTax` would be an implementation of the American tax code.

Implementing a Default Base Class

From the class `SwissTaxes` just presented, let's consider the properties `Incomes` and `Deductions`. In the example class, the properties weren't properly implemented, but this wouldn't be the case in an actual working example. Implementing `SwissTaxes` means implementing the properties using some type of assignment and retrieval. When coding `AmericanTaxes`, the same properties have to be implemented, and the exact same source code, or at least very similar, will be used to code American taxes. The problem with an interface is that there is no implementation. When an interface is implemented multiple times, it could occur that multiple implementations are similar if not identical.

The solution to the redundancy problem is to include a default base class that implements a base logic. The default base class would then be subclassed by the different controller implementations. An example default base class is as follows:

```
public abstract class BaseTaxation : ITaxation {
    private IIncomes[] _incomes;
    private IDeductions[] _deductions;
```

```
    public IIncomes[] Incomes {
        get {
            if( _incomes == null) {
                throw new PropertyNotDefined( "BaseTaxation.Incomes");
            }
            return _incomes;
        }
        set { _incomes = value; }
    }
    public IDeductions[] Deductions {
        get {
            if( _deductions == null) {
                throw new PropertyNotDefined( "BaseTaxation.Deductions");
            }
            return _deductions;
        }
        set { _deductions = value; }
    }
    public abstract Decimal CalculateTax();
}
```

The class `BaseTaxation` has the scope public abstract and implements the interface `ITaxation`. The properties `Incomes` and `Deductions` are implemented in the class and provide the base logic for all taxation implementations. The C# compiler will require the method `CalculateTax` to be implemented and is therefore declared as abstract.

The use of the keyword `abstract` is targeted and makes it simpler to implement a base logic. When using the `abstract` keyword in the context of a class, it means that the class can be subclassed, but can't be instantiated. This is appropriate because the defined base class is a helper class and not a full implementation. It's absolutely vital to remember this; otherwise the class might be inappropriately instantiated. Consider any implemented method in the base class as a method that all derived implementations will use without change. And consider any implemented method as an implementation that is structural and has less to do with application process logic.

When implementing an interface in a default base class, all methods have to be implemented, even though the methods might not be relevant for the default base class. For example, the class `BaseTaxation` requires the implementation of the method `CalculateTax`. As the default base class doesn't actually provide a default implementation for the methods, the appropriate choice for each method is to use the keyword `abstract`. By using this keyword on a method, the method doesn't have to be implemented, but requires any subclass to implement the method.

Default base classes are a very powerful mechanism used by controller classes. But they aren't limited to controller classes and can be used in different contexts where a basic functionality is required. Default base classes will often be exposed using the `internal` keyword, even though the class `BaseTaxation` is public in scope. The scope identifier depends on how many implementations will use the default base class. All implementations of `ITaxation` will subclass `BaseTaxation`, as the properties are useful for all implementations and the public scope is appropriate.

If you are using .NET Generics, default base classes are an excellent opportunity to include .NET Generics. For example, the class `BaseTaxation` could be defined in terms of Generic types. When implementing taxation for a particular country, the type `BaseTaxation` would be subclassed using specific types.

Multiple default base classes could be defined. For example, a `BaseEUTaxation` abstract class would subclass the `BaseTaxation` class. This is because the tax systems within the EU use similar rules even though the rates and calculations are different. Then the German and French tax implementations would derive from the `BaseEUTaxation` abstract class. When defining multiple default base classes, it's important to use the `abstract` keyword to ensure no factory attempts to instantiate an incomplete class.

Abstract classes are useful, but they also create problems when testing the individual classes. The abstract class can't be instantiated, and therefore it's necessary that mock objects derive from the default base classes. Following is an example of a test for the `BaseTaxation` class:

```
internal class MockBaseTaxation : BaseTaxation {
    public override Decimal CalculateTax() {
        throw new MockNotImplemented();
    }
}

internal class MockIncome : IIncomes {
    public void SampleMethod() {
        throw new MockNotImplemented();
    }
}

[TestFixture]public class TaxTests {
    [Test]public void TestAssignIncomeProperty() {
        IIncomes[] inc = new IIncomes[ 1];
        inc[ 0] = new MockIncome();
        ITaxation taxation = new MockBaseTaxation();
        taxation.Incomes = inc;
        Assert.AreEqual( inc, taxation.Incomes, "Not same object");
    }
    [Test]
    [ExpectedException(typeof(PropertyNotDefined))]
    public void TestRetrieveIncomeProperty() {
        ITaxation taxation = new MockBaseTaxation();
        IIncomes[] inc = taxation.Incomes;
    }
}
```

Mock objects were created for the `IIncomes` and `BaseTaxation` types. In real life, the `IDeductions` interface would have had an associated mock object, but it wasn't implemented for the purpose of clarity. A mock object was created for the `IIncomes` interface because the focus of the test is to determine the `BaseTaxation` type. The required methods to be implemented by the types `IIncomes` and `BaseTaxation` will generate an exception. This is done on purpose and reflects the nature of the `BaseTaxation` type in that none of those methods should be used

or called. The test methods that are part of the `TaxTests` class are generic test methods that have been described previously in Chapter 2 and in this chapter, and won't be discussed further.

There is a problem when implementing mock objects for default base classes in that the mock objects need to implement some methods or properties. Going back to the `BaseTaxation` base class, this means the method `CalculateTax` needs to be implemented. The problem is what the mock object implements. The simplest and logical answer is to throw an exception. An exception is the appropriate action because the purpose of the mock object is to provide a placeholder to test other functionality. The only scenario where an exception isn't appropriate is when the mock object method implementation is expected to do something. Then the mock object needs to implement some default behavior.

For example, let's say a test required that the `CalculateTax` method be called. In implementation terms, the `BaseTaxation` mock object would be written as follows:

```
internal class MockBaseTaxationRequiresCall : BaseTaxation {
    private bool _didCall;
    public MockBaseTaxationRequiresCall() {
        _didCall = false;
    }
    public bool DidCall {
        get { return _didCall; }
    }
    public override Decimal CalculateTax() {
        _didCall = true;
        return new Decimal();
    }
}
```

The class `MockBaseTaxationRequiresCall` has a property, `DidCall`, that indicates whether or not the method `CalculateTax` was called. The state of the property is set to `false` when the class is instantiated.

■ **Note** In Chapter 2, I mentioned that initialization code shouldn't be located in the constructor, but in a reset method. For a test, it's acceptable to use the constructor because of the narrow focus of the test.

The method `CalculateTax` implements an expected action, which in the example is nothing, and assigns the `DidCall` property to `true`. The test that uses the class `MockBaseTaxationRequiresCall` is written as follows:

```
 [Test]public void TestDidCallMethod() {
    MockBaseTaxationRequiresCall taxation = new MockBaseTaxationRequiresCall();
    // Call some methods
    Assert.IsTrue( taxation.DidCall);
}
```

It's important to write tests for default base classes, because otherwise it's unknown whether the default functionality works. The tests for default base classes will involve writing

mock objects, and it's highly recommended that the code be kept in a separate namespace, because there might be multiple different implementations of the same mock object type to test different aspects of the base class.

Interface and Class Design Decisions

Interfaces aren't always the best ways to define types. The reason why interfaces are constantly referenced is because in past technologies that was the only way to separate intention from implementation. Interfaces were used for technological reasons and not just design reasons. With .NET, you don't need to use interfaces other than for design reasons. In fact, it isn't even necessary to use interfaces at all, since abstract classes and abstract methods can be used as substitutes. This then raises the question, Is the need for interfaces passé? Interfaces are necessary, but there are specific contexts. The Bridge pattern is designed to separate implementation from intention, and thus interfaces are an ideal context and the most appropriate.

In .NET, it isn't possible to create a class that subclasses multiple classes, as .NET doesn't support multiple inheritance. .NET only supports the implementation of multiple interfaces. This means when a class wants to offer multiple pieces of differing functionality, interfaces must be used. When a class wants to offer a major piece of functionality, then abstract classes can be used. Abstract classes can even be used to control what classes offer as functionality.

The argument is as follows: defining an interface means defining a piece of functionality. When a class implements multiple interfaces, multiple functionalities are combined. In some situations, combining multiple functionalities is entirely inappropriate. But at other times, it's entirely appropriate. For example, the interfaces `IIncomes` and `IDeductions` could be implemented as a single class that implements two interfaces, or as two classes that implement a single interface.

To see how you can implement interfaces, let's extend the taxation example. The Swiss tax system calculates taxes to the nearest Swiss Franc. When filling out a tax form, the amounts are always rounded to the nearest Swiss Franc, and that makes it simpler to add the various amounts. Other tax systems require rounding to the nearest hundredth of a currency unit. An argument could be made that there is a requirement for defining specific methods to perform tax calculations. The question, though, is where this method goes. The simplest solution is to make the method part of the `ITaxation` interface as shown by the following example:

```
public interface ITaxation {
    IIncomes [] Incomes { get; set; }
    IDeductions [] Deductions { get; set; }

    Decimal IncomeTax( Decimal rate, Decimal value);
    Decimal CalculateTax();
}
```

The method `IncomeTax` calculates the income tax for a given rate and value. The calculation method used depends on the implementation. The design at this moment would include two calculation implementations, round off to the nearest currency unit, and round off to the nearest 100th of a currency unit.

Let's consider the ramifications of adding the method `IncomeTax` to the `ITaxation` interface. The main problem is that the method `IncomeTax` has nothing to do with the functionality offered by the `ITaxation` interface, because the calculation of a taxation is separate from determining

the numbers used to calculate the taxation. Simply put, calculating the tax at a mathematical level doesn't depend on the deductions or incomes. Calculating the tax depends on tax rate and taxable income.

If default base classes were defined, then the method `IncomeTax` would have to be referenced. The question is whether a base class defines a base calculation, because after all, the math of calculating taxes is identical across implementations. It could be argued that the math of calculating taxes is a structural issue, and hence needs to be in the default base class. The issue, though, is that not everybody calculates the tax the same way. As odd as it may sound, the mathematics of tax calculation is different in that Americans uses dollars and decimals of dollars, the Swiss use only Francs, and the Japanese have no part values in that there is only the yen and no decimals. Yet Americans and Canadians calculate their taxes using the same mathematical rules. The dilemma is that the functionalities are similar, yet dissimilar. The object-oriented character of a software engineer is screaming to create a default base class and use inheritance.

The solution is to not consider everything as one interface, but as two interfaces:

```
public interface ITaxMath {
    Decimal IncomeTax( Decimal rate, Decimal value);
}
public interface ITaxation {
    IIncomes [] Incomes { get; set; }
    IDeductions [] Deductions { get; set; }
    ITaxMath [] TaxMath { get; set; }

    Decimal CalculateTax();
}
```

With two interfaces, `ITaxMath` and `ITaxation`, the logic is clearer to understand. One interface figures out what the numbers are for calculating the tax, and the other does the actual calculation. Since there are two interfaces, the logical solution is to use two implementations. Yet the better solution is to use a single class as illustrated by the following example:

```
interface class SwissTax : ITaxMath, ITaxation { }
```

The class `SwissTax` implements both interfaces `ITaxMath` and `ITaxation`, and it would seem doing so is counterproductive. In fact, it would probably drive a classical object-oriented designer batty (especially after my long discussion about separating functionality into two interfaces). Then along comes the Swiss taxation example that binds everything into a single class. The difference with the `SwissTax` class is that the implementation of the interfaces `ITaxMath` and `ITaxation` are related. Swiss taxes must be calculated using a Swiss calculator. Therefore, a single class can easily implement both interfaces. Only Swiss taxes will use the Swiss calculator. An argument could be made that the Japanese use the same calculation and hence could use the Swiss calculator, but in fact that is incorrect. The Japanese have no hundredths of a currency unit, but the Swiss do. What this means is that to calculate Japanese taxes, a third tax calculator needs to be implemented. And since the Japanese tax calculator is specific to Japan, a single class as in the `SwissTax` example would be appropriate. Yet when calculating American and Canadian taxes, there will be three classes: `AmericanTaxation`, `CanadianTaxation`, and the taxation calculator `CalculateTaxDecimal`. The main point to remember is that you have the choice and should use what is most appropriate.

There is nothing wrong with implementing multiple interfaces using a single class. It's a technique that is used very often and has proven to be a useful solution. However, some important ramifications exist, as implementing multiple interfaces means combining multiple implementations. Following are two of the most important:

- *Resource changes*: Implementing two interfaces using a single class should only be done when both interfaces are going to be used. In the taxation system, a controller requires a calculator, hence both interfaces will be instantiated.
- *Consistency changes*: When both interfaces are instantiated, it's important to consider data consistency. For example, calling some methods on one interface could impact the outcome of methods from another interface. If the original design of the interfaces didn't take this into account, then when testing the interfaces, there could be data consistency issues.

Often when designing applications, in some scenarios it isn't necessary to use interfaces. Sometimes interfaces are overused, and that leads to problems. An example is the XML document object model (DOM). The XML DOM is entirely interface based, and in many respects it leads to slower applications that don't work as well. As a result, many XML libraries are rewritten to not use interfaces. The problem isn't the use of interfaces, but the overuse of interfaces. For example, the interface `ITaxMath` could have been easily implemented as a class as illustrated by the following example:

```
public class TaxMath {
    public virtual Decimal IncomeTax( Decimal rate, Decimal value) {
        return new Decimal();
    }
}
public class TaxMathFactory {
    public static TaxMath Instantiate() {
        return new TaxMath();
    }
}
```

The class `TaxMath` has replaced the `ITaxMath` interface, and the method `IncomeTax` has been declared as virtual so that any other classes derive their own functionality. In the case of calculating Swiss taxes, this means rounding off the values created by the base class `TaxMath`.

Even though a class has replaced the interface, a factory must exist. The factory class `TaxMathFactory` treats the class `TaxMath` as a base class. This is done on purpose because every class that uses the `TaxMath` functionality only needs to consider the type `TaxMath`, and not the types of the derived classes. When implementing the Swiss tax calculation, the code would appear similar to the following:

```
public class SwissTaxMath : TaxMath {
    public override Decimal IncomeTax( Decimal rate, Decimal value) {
        return new Decimal();
    }
}
```

```
public class SwissTaxMathFactory {
    public static TaxMath Instantiate() {
        return new SwissTaxMath();
    }
}
```

The class `SwissTaxMath` derives from `TaxMath`. The factory `SwissTaxMathFactory` instantiates the type `SwissTaxMath` and returns the type `TaxMath`.

When using classes as "interfaces," don't ignore the use of the keyword `abstract`. In the previous examples, it wasn't necessary, but it avoids the problem of having a user instantiate the type directly. For example, to avoid instantiating the `TaxMath` class directly, the following code can be used:

```
public abstract class TaxMath {
    public virtual Decimal IncomeTax( Decimal rate, Decimal value) {
        return new Decimal();
    }
}
internal class StubTaxMath : TaxMath {
}
public class TaxMathFactory {
    public static TaxMath Instantiate() {
        return new StubTaxMath();
    }
}
```

In the example, the class `StubTaxMath` derives from the abstract-scoped class `TaxMath`. The scope of the class `StubTaxMath` is internal, so it can't be inappropriately instantiated. The factory `TaxMathFactory` instantiates the type `StubTaxMath`, but returns the abstract class `TaxMath` type.

Using classes and abstract classes instead of interfaces is useful when the implementations are narrowly scoped and tend to be reusable. For example, nobody would ever think of defining strings as an interface. The `string` type solves a narrowly scoped problem and is often used.

Interfaces define reusable contracts that are implemented in different contexts, and they solve a narrowly defined problem. If an interface tries to solve too many problems, the resulting implementations become unwieldy and problematic. An implementation of an interface is considered a modular solution and not reusable because it solves a single problem. For example, the Swiss or American taxation implementations aren't reusable because they solve a narrowly scoped problem. The Swiss and American taxation implementations are modular because they are used in the overall application to calculate taxations for various countries.

When using classes as interfaces, the classes are considered reusable and not modular. For example, a tax calculator is used in multiple contexts such as for the American, Canadian, German, and British taxation systems. But when defining a reusable class, often it's necessary to specialize the functionality, and as such keywords like `virtual` or `abstract` need to be used.

Now that you've become familiar with the Bridge pattern and it's variations, I want to turn your attention next to instantiating types with the Factory pattern.

Instantiating Types with the Factory Pattern

The Factory pattern[2] is used to instantiate a type. The simplest of factories is one that has a single method and instantiates a single type. Such a factory doesn't solve all problems in all contexts, and therefore different instantiating strategies have to be employed. All of these strategies are creational patterns that operate similarly to the factory. Specifically, in this section we'll explore why you want to use helper types, as well as how to create plug-ins, how to implement objects according to a plan, and when to clone objects.

The Need for a Helper Type

The helper type used to instantiate another type has been illustrated in multiple places thus far in the chapter. Also outlined were some reasons why using helper objects is a good idea. What wasn't covered are the detailed reasons why it's a good idea.

Helper objects make it simpler to keep the details of instantiating a type hidden from the consumer. Let's say that a type needs to be instantiated. The consumer uses the `new` keyword to instantiate a type that implements an interface. Remember that the implementation of an interface is modular. It could be that one implementation operates under one set of conditions, and another uses a different set of operating conditions. The simplest example is that one implementation needs to be used as a singleton, and another can be instantiated for each method call. (A singleton is a single instance of a type.) Consider the following code that illustrates different operating conditions:

```
public interface SimpleInterface {
}
internal class MultipleInstances : SimpleInterface {
}
internal class SingleInstance : SimpleInterface {
}
public class SimpleInterfaceFactory {
    public static SimpleInterface FirstType() {
        return new MultipleInstances();
    }
    private static SingleInstance _instance;
    public static SimpleInterface SecondType() {
        if( _instance == null) {
            _instance = new SingleInstance();
        }
        return _instance;
    }
}
```

The interface `SimpleInterface` is considered a reusable type that is implemented by the classes `MultipleInstances` and `SingleInstance`. The difference with the class `SingleInstance` is that there can only ever exist a single instance. The factory class `SimpleInterfaceFactory` has

2. *Design Patterns: Elements of Reusable Object-Oriented Software,* pp. 87, 107.

two methods that will instantiate both types. The method `FirstType` is a standard factory. What is different is the method `SecondType`, as it only ever instantiates one instance of the class `SingleInstance`. In the method `SecondType`, a test is made against the global variable `_instance`. If the variable `_instance` is null, then a new instance of `SingleInstance` is assigned. After the test, the reference of the variable `_instance` is returned to the caller. Then no matter how often the method `SecondType` is called, the same instance will be returned.

The caller of the method `SecondType` doesn't realize that the interface instance returned is the same instance. This is the main objective behind using a helper type to instantiate a type. The helper type can set the operating conditions of the type being instantiated. It doesn't matter whether the type to be instantiated returns an interface or a default base class, or class. The consumer of a helper type can be assured that whatever object instance is returned is consistent and has the right operating environment.

The factory `SimpleInterfaceFactory` contained two methods, `FirstType` and `SecondType`. Most factories will be structured with multiple instantiation methods. Typically, such a factory will have instantiation methods suitable for a grouping of implementations. If the interface related to transportation methods, then for all cars there would be a factory, and for all ships there would be another factory. The idea behind such multimethod factories is to be able to create all types of implementations that belong to a group based on a single factory.

Implementing a Plug-In Architecture

A more sophisticated factory is the plug-in factory. The plug-in factory instantiates types based on textual definitions extracted from a configuration file. A plug-in factory is flexible and is an effective way to separate intention from implementation. The general idea behind a plug-in factory is to be able to instantiate an interface instance based on some textual representation of an implementation that is part of an assembly. A full plug-in implementation is available in the downloadable source code for this book, which you can get from the Apress website (`http://www.apress.com`). Only covered here are the essential details of a fully functional plug-in factory.

From the consumer perspective, the way to instantiate an interface instance using a plug-in factory is as follows:

```
ITaxation obj = (ITaxation)(Factory.GetObject( "Taxation.SwissTaxationImpl"));
```

The consumer would instantiate the interface `ITaxation` using the type identifier of an `ITaxation` implementation. Because the code is written using .NET 1.x, a typecast is required as the `GetObject` method is implemented using the type object.

Using Generics, the method `GetObject` would be rewritten as follows:

```
ITaxation obj = Factory.GetObject<ITaxation>( "Taxation.SwissTaxationImpl");
```

As this demonstrates, the advantage of a plug-in factory is that it's an elegant solution that can be used generically.

The implementation of the plug-in factory is a bit more complicated because multiple actions have to be fulfilled:

- Identify the Generic types that are shared by the consumer and implementations. What binds the consumer and the implementation together is the interface, default base class, or some Generic class. The Generic types should be stored in a separate assembly referenced by both.
- Identify the assembly(ies). The interface implementations are located in separate assemblies, which need to be loaded when an interface instance is instantiated.
- When requested, load an assembly, instantiate the object, and typecast to the Generic type.
- Return the instantiated and typecast object to the consumer.

Following is an example that loads an assembly:

```
Assembly assembly = Assembly.LoadFrom( path);
```

The method `LoadFrom` is a static method that loads an assembly based on the path stored in the `path` variable. When an assembly is loaded, the types implemented within the assembly can be instantiated using the following method call:

```
object obj = assembly.CreateInstance( typeidentifier);
```

The type that is instantiated is the text-based identifier in the variable `typeidentifier`. An example might be `Taxation.SwissTaxationImpl`. If the type could be instantiated, then the object instance will be stored in the variable `obj`. If the object couldn't be instantiated, then the variable `obj` will be `null`.

Another solution to dynamically loading and instantiating an object instance is to use the `Activator` class type as shown here:

```
object obj = Activator.CreateInstance( Type.GetType( typedidentifier));
```

When called from the `Activator` class, the method `CreateInstance` requires a .NET type class instance. The type class instance is identified by the variable `typeidentifier`, which is a string. The string has a special notation: "`{type name},{Assembly path}`". Returned will be an object instance.

When loading assemblies dynamically, it isn't easy to debug the assembly. This is because the debugger doesn't know about the assembly, and hence trying to set breakpoints is difficult. However, since we are using test-driven development, this isn't an issue, as the assemblies are tested in isolation.

The illustrated plug-in architecture works for scenarios where the type is specified via a configuration defined elsewhere (for example, in a file) that won't change throughout the execution of an application. The plug-in architecture can't be used to reload or update an assembly while the application is executing. To do that, you will need to use the Client-Dispatcher-Server pattern, which is discussed in Chapter 4.

Creating Objects According to a Plan

In previous examples, the interfaces `ITaxation` and `ITaxMath` were defined. Within the interfaces defined was the property `TaxMath`. The idea is to associate a generic tax calculator with a taxation system. Each interface would have its own factory, but some code would have to associate the `ITaxMath` instance with the `ITaxation` instance. To solve this problem, you use a pattern called the Builder pattern.

The general idea of the Builder pattern is to be able to create a set of object instances. Remember, previously I mentioned that the helper types used to instantiate the objects are intended to define an operating context. The Builder pattern extends this by instantiating multiple objects in the context of one method. Following is an example of the Builder pattern that creates a complete Swiss taxation system:

```
public class Builder {
    public ITaxation InstantiateSwiss() {
        ITaxation taxation = new SwissTaxationImpl();
        taxation.TaxMath = new SwissTaxMathImpl();
        return taxation;
    }
}
```

The method `InstantiateSwiss` instantiates two types, `SwissTaxationImpl` and `SwissTaxMathImpl`, and wires them together. It's important to have the type `SwissTaxationImpl` not implicitly create the type `SwissTaxMathImpl` because that would corrupt the functionality of the controller. A controller is a generic operations type that only manipulates Generic types. The controller expects the right types to be associated with each other and the controller.

Between the Builder and Factory patterns, there isn't that much difference. The only real difference is that the Builder pattern will instantiate multiple types and wire them together. This raises the question whether the Builder pattern should only use `Factory` classes. The answer is no for most cases. A builder class needs to have intimate knowledge about the types that it's instantiating. Adding another abstraction layer will, for most cases, complicate the application architecture.

Where factory classes must be used is if a plug-in architecture exists. Plug-ins are referenced using general identifiers and not specific type identifiers. Therefore, when wiring together plug-ins, the plug-in factory is used, and the instances are wired together using the provided interfaces.

Cloning Objects

The last major way to instantiate a type is to instantiate the type based on an object instance. This is called *cloning an object*, or *implementing the Prototype pattern*. The big picture idea of this pattern is to be able to create a copy of an object that already exists.

In .NET, the way to clone an object is to implement the `ICloneable` interface as shown by the following example (ignore the lack of implementation details for the property `TaxMath`, which was done for clarity):

```
class SwissTaxationImpl : ITaxation, System.ICloneable {
    ITaxMath _taxMath;
    public ITaxMath TaxMath { get {;} set {;} }
    public Object Clone() {
        SwissTaxationImpl obj = (SwissTaxationImpl)this.MemberwiseClone();
        obj._taxMath = (ITaxMath)((System.ICloneable)_taxMath).Clone();
        return obj;
}
```

The method `Clone` implements both a shallow clone and deep clone. A shallow clone is when only the type's local data members are copied. The shallow clone is realized using the method call `MemberwiseClone`. It isn't necessary to call the method `MemberwiseClone` and a type instantiation using the `new` keyword. The method `MemberwiseClone` was used for safety purposes. To implement a deep clone, the `clone` method of the classes' data members is called.

One of the debates of cloning is whether or not to implement a deep clone. For example, if type A references type B, then a deep clone will copy both type instances. But imagine if type A references type B, and type B references another instance of type B, which references the original type B instance. A cyclic reference scenario has been established, which when cloned using a deep clone could lead to a never-ending cloning scenario. To avoid those tricky situations, adding a `didClone` flag could be useful. However, even with a `didClone` flag, problems will arise. The better solution is to rethink the cloning scenario and to manually clone each data member and rebuild the structure using manual object assignment. There is no silver bullet solution to implementing a deep clone.

Some Final Thoughts

This chapter provided a look into how to develop code that defines an intention and implementation that is instantiated using a factory type. The key aspect to remember is that intentions can be defined using interfaces or classes. Intentions are a mechanism used to describe generic actions using a general type that makes it simpler for a consumer to interact with other types. Implementations are shielded from each other and are specific solutions to an intention.

Intentions are generally reusable, and implementations are generally modular, but usually not vice versa. Keeping this philosophy in mind makes it simpler to define and implement applications.

For a consumer to interact with an implementation based on an intention, you have to create a helper object called a factory. The factory object is the "middleman" that knows how to instantiate a type and create the proper operating environment. The factory object may be realized as a single method, a group of methods, or methods that wire multiple objects together. The essential point is that there is a helper class that is responsible for instantiating the right types, and it establishes a state and context so that consumers can execute their logic.

The information in this chapter serves as a foundation for the rest of the chapters. In a way, it's interesting that the Bridge and Factory patterns, including their variations, are the basis of all modern object-oriented applications. It also means that if you don't properly understand the Bridge and Factory patterns, you won't properly understand the other patterns described in this book.