ActionScript 3.0 Design Patterns

William B. Sanders and
Chandima Cumaranatunge

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

ActionScript 3.0 Design Patterns
by William B. Sanders and Chandima Cumaranatunge

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Steve Weiss
Developmental Editor: Robyn G. Thomas
Production Editor: Philip Dangler
Copyeditor: Sohaila Abdulali
Indexer: John Bickelhaupt
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrators: Robert Romano and Jessamyn Read

Printing History:

July 20007: First Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-10: 0-596-52846-9
ISBN-13: 978-0-59652846-1
[M]

Adobe Developer Library, a copublishing partnership between O'Reilly Media Inc., and Adobe Systems, Inc., is the authoritative resource for developers using Adobe technologies. These comprehensive resources offer learning solutions to help developers create cutting-edge interactive web applications that can reach virtually anyone on any platform.

With top-quality books and innovative online resources covering the latest tools for rich-Internet application development, the *Adobe Developer Library* delivers expert training straight from the source. Topics include ActionScript, Adobe Flex®, Adobe Flash®, and Adobe Acrobat®.

Get the latest news about books, online resources, and more at *http://adobedeveloperlibrary.com*.

Table of Contents

Part III. Structural Patterns

CHAPTER 2

Factory Method Pattern

As experimentation becomes more complex, the need for the cooperation in it of technical elements from outside becomes greater and the modern laboratory tends increasingly to resemble the factory and to employ in its service increasing numbers of purely routine workers.

—John Desmond Bernal

The medieval university looked backwards; it professed to be a storehouse of old knowledge. The modern university looks forward, and is a factory of new knowledge.

—Thomas Huxley

One cannot walk through an assembly factory and not feel that one is in Hell.

—W. H. Auden

What Is the Factory Method Pattern?

One of the most common statements in object-oriented programming (OOP) uses the `new` keyword to instantiate objects from concrete classes. ActionScript applications that have multiple classes can have an abundance of code that looks like the following:

```
public class Client
{
    public function doSomething( )
    {
        var object:Object = new Product( );
        object.manipulate( );
    }
}
```

The `Client` class creates a new instance of the `Product` class and assigns it to the variable `object`. There's nothing wrong with this code, but it does create a *coupling*

or *dependency* between the `Client` and `Product` classes. Simply put, the `Client` class depends on the `Product` class to function properly. Any changes to the `Product` class in terms of class name changes or change in the number of *parameters* passed to it will require changes in the `Client` class as well. This situation is exacerbated if multiple clients use the `Product` class, and requires changing code in multiple locations.

The solution to this common problem is to loosen the tight coupling between the client and the concrete classes it uses. This is where the factory method pattern offers a robust solution. It introduces an intermediary between the client and the concrete class. The intermediary is called a *creator* class. It allows the client to access objects without specifying the exact class of object that will be created. This is accomplished by delegating object creation to a separate method in the creator called a *factory*. The primary purpose of the factory method is to instantiate objects and return them.

Model of the Factory Method Pattern

Figure 2-1 shows the high-level model of the factory method pattern. Multiple *clients* can use the factory method in a *creator* to access and use multiple *products*. The intermediary nature of the creator is clear in this model, as it forces the creation of multiple types of objects (different products) through a common point.

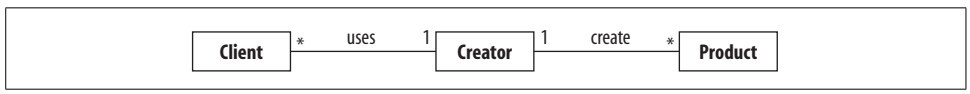


Figure 2-1. Logic model of factory method pattern

This high-level model doesn't show how clients can use a factory method to access objects without specifying their concrete classes. Let's write some code for the creator class to create and return product objects.

```
public class Creator
{
    public static function simpleFactory(product:String)
    {
        if (product == "p1")
        {
            return new product1( );
        } else if (product == "p2") {
            return new product2( );
        }
    }
}
```

The parameterized method called `simpleFactory( )` instantiates product classes and returns them. The client would call this method and pass the product identifier, which in this case is the `String` value `"p1"` or `"p2"`. This loosens the coupling between the client and the product classes. However, it's a very commonly used code segment, and doesn't offer the reusability and flexibility offered by the factory method pattern. This code segment is commonly known as a *simple factory*.

To add a new product, we'll have to modify the `simpleFactory()` method and add another IF clause. This goes against the open-closed principle in OOP where code such as classes and methods should be *open for extension,* but *closed for modification*. One of the primary advantages of the factory method pattern is indeed the extensibility it affords to accommodate change. Let's look at the class diagram of the classic factory method pattern.

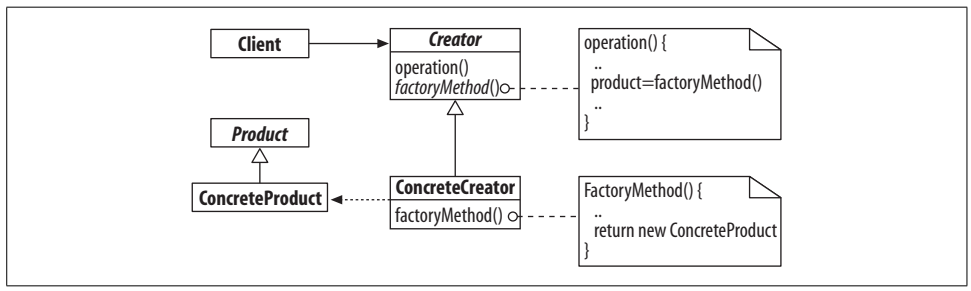


Figure 2-2. Class diagram of the factory method pattern

The creator and the product are both defined as *interfaces* and not concrete classes (interfaces are indicated by italicized class names in class diagrams). Interfaces define the type and method signatures for classes. Classes that implement an interface have to implement the methods declared in the interface. A pure interface does not provide any implementation for declared methods. However, there is a special kind of interface called an *abstract interface*. Abstract interfaces can provide default implementations for methods. They're also called *abstract classes*, and cannot be instantiated, but can be extended by other classes.

The `Creator` class in Figure 2-2 is an abstract class. It declares a *factory method* (called `factoryMethod()` in this case). This is where most of the action takes place. The factory method is defined as an *abstract method* without any implementation. This puts the onus on subclasses (classes that extend the abstract class such as `ConcreteCreator`) to provide the implementation details for the factory method. The `Creator` class also defines and implements a public method (called `operation()` in this case) that calls the factory method.

```
operation()
{
    ..
    product = FactoryMethod()
    ..
}
```

The `operation()` method calls `FactoryMethod()` to create product objects.

```
factoryMethod() {
    ..
    return new ConcreteProduct
}
```

The primary responsibility of the factory method is to instantiate and return product objects. The interesting issue is that even though *factoryMethod*( ) is declared in the abstract class Creator, it's implemented in ConcreteCreator. Therefore, it is the ConcreteCreator class that knows about the product classes, essentially hiding the product classes from the client.

The reason for using interfaces and abstract classes will be clear when we look at a real application as it allows the addition of new products and corresponding creators by extending, as opposed to changing, existing code. This is a big deal in OOP because it allows a safe way to add new functionality without breaking anything. The ConcreteCreator class extends the Creator abstract class. The Product interface can be either a pure interface or abstract class depending on whether there is default functionality that needs to be implemented for all products.

Clients access the ConcreteProduct classes through the Creator interface. To force clients to access products through the factory methods, the product classes are generally hidden from outside access. We can see how this is implemented in ActionScript by developing a minimalist example.

Abstract Classes in ActionScript 3.0

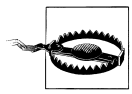
Before developing an example factory method pattern, we need to tackle the issue of *abstract* classes, or more specifically, lack of support for them in ActionScript 3.0. The factory method pattern defines creators as abstract classes, and there's no way around this. In fact, much of the usefulness of the pattern can be attributed to this abstraction.

Abstract classes cannot be instantiated. They have to be extended by subclasses. They can contain abstract methods or unimplemented method declarations that subclasses need to implement. The methods implemented in an abstract class will in most cases be default behaviors, and much of the class will be unimplemented. Before a class derived from an abstract class can be instantiated, it must implement all unimplemented methods. The advantage of deriving from an abstract class is that the subclass does not have to implement a method if the default behavior implemented in the abstract class is what it needs.

Defining the creator class as abstract enables us to concentrate on a few concepts that change, but leave others at their default functionality. This reduces complexity—one of the key benefits of OOP. For example, from our model, we can use the default implementation for the operation( ) method, but override the *FactoryMethod*( ).

Unfortunately, ActionScript 3.0 does not support abstract classes. The alternative is to implement abstract classes as concrete classes in ActionScript without the instantiation and method implementation checks. These checks are conducted at compile time in languages that support abstract classes. We can add code to concrete classes in ActionScript 3.0 to do these checks at runtime and throw violation errors. In

addition, commenting the classes and methods that should behave as abstract is also important. It must be emphasized that this puts the burden on the programmer as opposed to the compiler to check if all methods that should behave as abstract are implemented.



We will define *abstract* classes as *concrete* classes with the knowledge that they will not be instantiated, but will be extended by subclasses. *Abstract methods* will be defined simply as a function declaration that will throw an `IllegalOperationError` error if called. Both abstract classes and methods will be clearly identified using comments.

Minimalist Example

We will develop a minimalist example of a factory method pattern in ActionScript 3.0 using Flash CS3. The Project window shown in Figure 2-3 mirrors the file structure for the example.

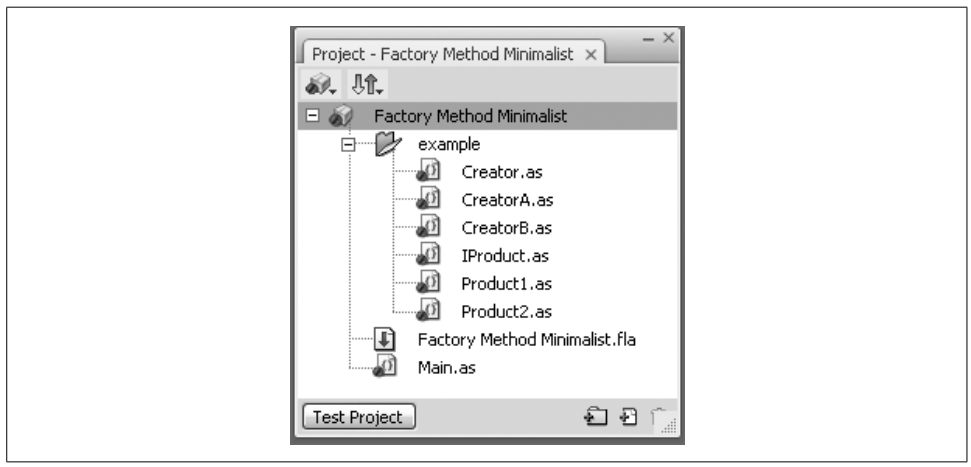


Figure 2-3. Project window for the minimalist example

The *Factory Method Minimalist.fla* is a Flash document whose document class is defined as `Main`. The *document class* is a new feature in Flash CS3 that can be set from the Properties tab in the Properties panel of a Flash document (*.fla* file). This represents the class whose constructor will be automatically run by the Flash Player when the Flash movie (*.swf* file) is loaded. The `Main` class is defined in the ActionScript file called *Main.as*. The project also contains a *package* called `example`. Packages allow you to bundle classes together to facilitate code sharing. They also allow control over the visibility of classes and method names outside the package by using identifiers. This minimizes naming conflicts that can occur when developing applications. We will use packages to hide the product classes from direct access by clients in this example.

The example package contains the Product1 and Product2 classes that implement the IProduct interface. It also contains the *abstract* class Creator that is extended by two subclasses CreatorA and CreatorB. Each class is defined in its own file, as is the convention with ActionScript 3.0. First, let's take a look at the product classes.

Product Classes

Example 2-1 through Example 2-3 show the product interface and concrete product classes. All three classes belong to the example package as indicated by the *package* declaration. The IProduct interface declares a method called manipulate( ) that is implemented by both product classes. True to a minimalist example, the product classes don't do much of anything. However, make note of the class attribute.

Example 2-1. IProduct.as

```
package example
{
    public interface IProduct
    {
        function manipulate( ):void;
    }
}
```

Example 2-2. Product1.as

```
package example
{
    internal class Product1 implements IProduct
    {
        public function manipulate( ):void
        {
            trace("Doing stuff with Product1");
        }
    }
}
```

Example 2-3. Product2.as

```
package example
{
    internal class Product2 implements IProduct
    {
        public function manipulate( ):void
        {
            trace("Doing stuff with Product2");
        }
    }
}
```

The product classes are defined as *internal* (the default class attribute in ActionScript 3.0). This means that they're not publicly visible. They can only be called from within the `example` package. Now, let's examine the creator classes.

Creator Classes

Example 2-4 through Example 2-6 show the creator classes. As with the product classes, all creators belong to the package `example`. The creator classes are defined as *public*, which indicates that they are publicly accessible from outside the package. The creator class, as indicated by the comments, should behave as an abstract class. It should be subclassed and not be instantiated. It also defines the factory method that should behave as an abstract method. Note that the two concrete creator classes `CreatorA` and `CreatorB` extend `Creator`. They also override and implement the `factoryMethod( )` method, each returning a corresponding product object. In addition, the `factoryMethod( )` declared in the `Creator` class has to return `null` to prevent a compile-time error. It will also throw an `IllegalOperationError` if called directly. This is a runtime check to make sure that this method is overridden.

Example 2-4. Creator.as

```
package example
{
    import flash.errors.IllegalOperationError;

    // ABSTRACT Class (should be subclassed and not instantiated)
    public class Creator
    {
        public function doStuff( ):void
        {
            var product:IProduct = this.factoryMethod( );
            product.manipulate( );
        }

        // ABSTRACT Method (must be overridden in a subclass)
        protected function factoryMethod( ):IProduct
        {
            throw new IllegalOperationError("Abstract method:
                                        must be overridden in a subclass");
            return null;
        }
    }
}
```

Example 2-5. CreatorA.as

```
package example
{
    public class CreatorA extends Creator
    {
        override protected function factoryMethod( ):IProduct
        {
```

Example 2-5. CreatorA.as (continued)

```
            trace("Creating product 1");
            return new Product1( ); // returns concrete product
        }
    }
}
```

Example 2-6. CreatorB.as

```
package example
{
    public class CreatorB extends Creator
    {
        override protected function factoryMethod( ):IProduct
        {
            trace("Creating product 2");
            return new Product2( ); // returns concrete product
        }
    }
}
```

Note that the `doStuff( )` method in the `Creator` class is declared as *public* because we need to allow clients to get to it from outside the package. In contrast, the `factoryMethod( )` is declared as *protected*. The *protected* attribute makes the method visible only within the same class or derived classes. Finally, we can take a look at the document class called `Main`, which is the client described in the class diagram shown in Figure 2-2 that uses the creator to access products.

Clients

The document class `Main` does not belong to a named package. Therefore, it must import the packages that contain the classes it uses. The `example` package needs to be imported, as it contains the creator classes. The document class `Main` is the client described in the high-level model of the factory method pattern shown in Figure 2-1. In Example 2-7, the document class calls a static method called `run( )` in the static `Test` class to run some tests.

Example 2-7. Main.as

```
package
{
    import flash.display.Sprite;
    import example.*;

    /**
    *    Main Class
    *    @ purpose: Document class for movie
    */
    public class Main extends Sprite
    {
```

Example 2-7. Main.as (continued)

```
        public function Main( )
        {
            // instantiate concrete creators
            var cA:Creator = new CreatorA( );
            var cB:Creator = new CreatorB( );

            // creators operate on different products
            // even though they are doing the same operation
            cA.doStuff( );
            cB.doStuff( );
        }
    }
}
```

We get the following output after running the project.

```
Creating product 1
Doing stuff with Product1
Creating product 2
Doing stuff with Product2
```

Let's step through the code to see how we end up with the output. The client (document class `Main`) does not know anything about the product classes. It only knows about the creator classes and what they do. Therefore, the client instantiates `CreatorA` and `CreatorB`, and asks them both to `doStuff( )`. The `Creator` class behaving as an abstract class knows how to `doStuff( )`, but it has allowed subclasses to determine the product that it does stuff to. The `doStuff( )` method calls `factoryMethod( )` to return a product object. It then calls the `manipulate( )` method in the product object.

The primary task of the subclasses `CreatorA` and `CreatorB` is to create and return product objects. They know about the product classes to operate on, and they override the factory method to instantiate the appropriate products and return them to the `doStuff( )` method. Knowledge about object creation has been encapsulated within the concrete creator classes.

The factory method pattern has essentially created a firewall between clients and the concrete product classes they use. Is the firewall bulletproof? Have we fully accomplished what we set out to do? Are the product classes only accessible through the creators? We will have to check this.

Hiding the Product Classes

Let's check if the product classes are truly hidden and accessible only through the `creator` classes. We can try to instantiate a product by accessing its creator class and the factory method directly. Add the following statements to the `Test` class in the *Main.as* file (Example 2-7) to try and directly instantiate products:

```
// instantiate concrete products
var p1 = new Product1();
var p2 = CreatorB.factoryMethod();
```

The following compile-time errors shown in Figure 2-4 are produced when we run the project.

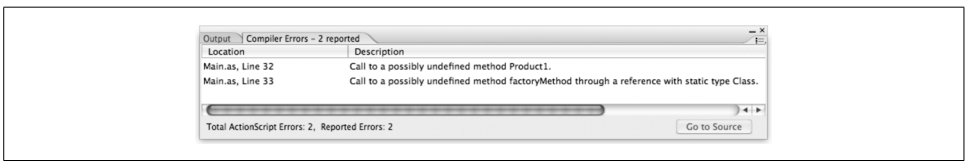


Figure 2-4. Compile-time errors when directly accessing product classes

The errors indicate that the product classes and the factory method aren't visible to the client. The product classes are encapsulated within the package, as they were defined with *attribute internal*. Internal classes can only be accessed from within the package. Similarly, the `factoryMethod()` is not accessible as it a *protected* class. Therefore, external access to the product classes is only possible through the `doStuff()` *public* method.

You may wonder why it's necessary to use this complex design just to prevent clients from directly instantiating concrete classes. The alternative would be much simpler. The client could have instantiated `Product1` and `Product2` and fed it to a `doStuff()` method. Let's address this issue after we develop a simple application using the factory method pattern. It will be easy to see the advantages when there is real context instead of generic products and creators.

Example: Print Shop

Let's develop an example application to dispatch print jobs at a hypothetical print shop (think copy center with printers). Think of the shop as a place where customers bring what they want to print on portable media. The clerk at the counter will initiate a print job on the computer for dispatch and billing, based on the type of print job. We will build the application in Flash and ActionScript 3.0 as a generic example to illustrate the utility of the factory method pattern.

Our print shop is a small time operation with only two printers. We have a workgroup printer and an inkjet that prints in black and white. In order to streamline operations, the manager has a bright idea to create separate print centers in the shop. One print center will handle high-volume jobs that will be sent to the workgroup printer. The other will handle low-volume jobs that will be dispatched to the inkjet. We design the print shop application using the factory method pattern, as we had heard somewhere that it allows for flexibility and expansion. We are hoping to make a profit and add more printers in the future. Figure 2-5 shows the class structure for the print shop example.

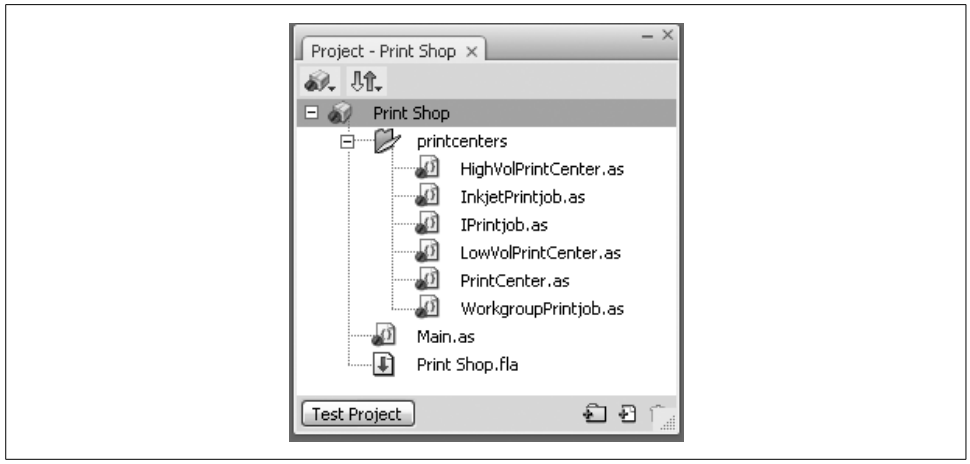


Figure 2-5. Project window for the print shop example

The class structure is very similar to the minimalist example shown in Figure 2-3. The products will be print jobs that will be created and the creators will be the print centers (see class diagram in Figure 2-6).

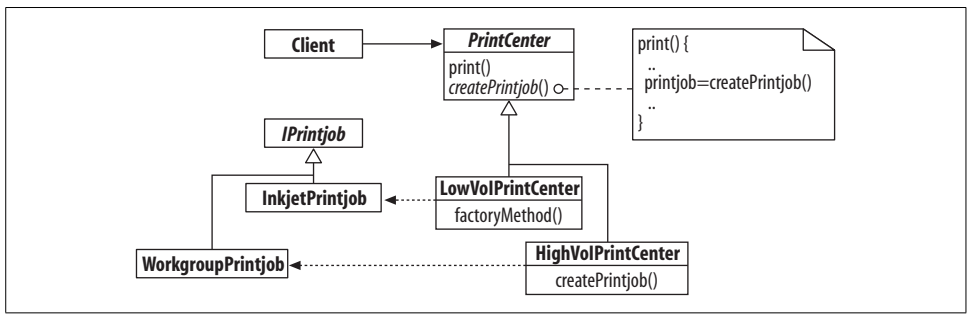


Figure 2-6. Class diagram for the print shop example

Product Classes: Print Jobs

The product classes don't do anything at this point, but in a real application they will initiate billing and dispatch print jobs to a print server. The print job classes belong to the `printcenters` package. Example 2-8 shows the `IPrintjob` interface that declares a parameterized method called `start()` that takes a file name of a document to print.

Example 2-8. IPrintjob.as

```
package printcenters
{
    public interface IPrintjob
    {
        function start(fn:String):void;
    }
}
```

Example 2-9 and Example 2-10 show the two concrete print job classes that implement the `IPrintjob` interface.

Example 2-9. InkjetPrintjob.as

```
package printcenters
{
    internal class InkjetPrintjob implements IPrintjob
    {
        public function start(fn:String):void
        {
            trace("Printing '" + fn + "' to inkjet printer");
        }
    }
}
```

Example 2-10. WorkgroupPrintjob.as

```
package printcenters
{
    internal class WorkgroupPrintjob implements IPrintjob
    {
        public function start(fn:String):void
        {
            trace("Printing '" + fn + "' to workgroup printer");
        }
    }
}
```

Creator Classes: Print Centers

Example 2-11 shows the class `PrintCenter` that should behave as an abstract class. The factory method is called `createPrintjob( )`, and client access to the printer centers is through the `print( )` method.

Example 2-11. PrintCenter.as

```
package printcenters
{
    import flash.errors.IllegalOperationError;

    // ABSTRACT Class (should be subclassed and not instantiated)
    public class PrintCenter
```

Example 2-11. PrintCenter.as (continued)

```
    {
        public function print(fn:String):void
        {
            var printjob:IPrintjob = this.createPrintjob( );
            printjob.start(fn);
        }

        // ABSTRACT Method (must be overridden in a subclass)
        protected function createPrintjob( ):IPrintjob
        {
            throw new IllegalOperationError("Abstract method:
                                        must be overridden in a subclass");
            return null;
        }
    }
}
```

Example 2-12 and Example 2-13 show the `LowVolPrintCenter` and `HighVolPrintCenter` classes that extend the `PrintCenter` class, and override and implement the `createPrintjob( )` factory method.

Example 2-12. LowVolPrintCenter.as

```
package printcenters
{
    public class LowVolPrintCenter extends PrintCenter
    {
        override protected function createPrintjob( ):IPrintjob
        {
            trace("Creating new printjob for the inkjet printer");
            return new InkjetPrintjob( );
        }
    }
}
```

Example 2-13. HighVolPrintCenter.as

```
package printcenters
{
    public class HighVolPrintCenter extends PrintCenter
    {
        override protected function createPrintjob( ):IPrintjob
        {
            trace("Creating new printjob for the workgropup printer");
            return new WorkgroupPrintjob( );
        }
    }
}
```

Clients

Note that the client doesn't know what specific printers the copy shop has in use. The client only knows about the type of print job defined by the creator classes. Based on the volume of the print job brought in by the customer, the application would instantiate the corresponding concrete print center class (`LowVolPrintCenter` or `HighVolPrintCenter`) and call the `print( )` method passing the filename of the document to be printed.

To test our design we run the following code from the client.

```
var pcHighVol:PrintCenter = new HighVolPrintCenter( );
var pcLowhVol:PrintCenter = new LowVolPrintCenter( );

pcHighVol.print("LongThesis.doc");
pcLowhVol.print("ShortVita.doc");
```

As in the minimalist example, we get the following output.

```
Creating new printjob for the workgropup printer
Printing 'LongThesis.doc' to workgroup printer
Creating new printjob for the inkjet printer
Printing 'ShortVita.doc' to inkjet printer
```

Looking at the output, it's clear that the `print( )` method operates on different objects (`WorkgroupPrintjob` and `InkjetPrintjob` objects). This is an elegant solution, as the client simply chooses the proper print center and requests it to print. The print center classes take care of instantiating the correct print job. There is a clear separation between creating an object and using the created object. Object creation is handled by the `factoryMethod( )`, and the created object is used by the `print( )` method. Object creation is completely hidden from the client.

Print Shop Extension

The real utility of the factory method pattern is evident when extending or adding more functionality to an application. Because the print shop is doing good business, the manager decides to add a fancy multi-function printer that has a sorting bin, a built-in stapler, and double-sided (duplex) printing features. How do we add this new printer to our existing application? We need a new print center in the print shop for anyone who needs additional features such as stapling or duplex printing. In terms of code, we need to create a new subclass of the `PrintCenter` class to handle the fancy print jobs. We will call this class `FancyPrintCenter` (Example 2-15). We also need to subclass `Printjob` and develop a new concrete class called `MultifunctionPrintJob` (Example 2-14) to dispatch jobs to the new printer. Let's look at the code changes needed to do this.

Example 2-14. MultifunctionPrintJob.as

```
package printcenters {

    internal class MultifunctionPrintJob implements IPrintjob {

        public function start(fn:String):void {
            trace("Printing '" + fn + "' to multifunction printer");
        }
    }
}
```

Example 2-15. FancyPrintCenter.as

```
package printcenters {

    import printcenters.*;

    // High Volume Print Center (subclass of PrintCenter)
    public class FancyPrintCenter extends PrintCenter {

        override protected function createPrintjob( ):IPrintjob {
            trace("Creating new printjob for the multifunction printer");
            return new MultifunctionPrintJob( );
        }
    }
}
```

We added two new classes, but didn't have to modify existing code at all. We added a new product class by implementing the `Printjob` interface and a new creator class by extending the `PrintCenter` abstract class. This is the big advantage of the factory method pattern when compared to the simple factory discussed previously. By declaring both the product and creator classes as interfaces, we were able to extend the code to add new functionality without changing existing code. To access the new multifunction printer, clients need to instantiate a `FancyPrintCenter` class, and call its `print( )` method.

Parameterized Factory Methods

The examples we've seen have used non-parameterized factory methods. *Non-parameterized* factory methods don't take any parameters in their function declarations. *Parameterized* factory methods take a parameter that specifies a kind of product that will be created. For example, in the print shop application, you can pass an extra *parameter* to the factory method to indicate a particular kind of print job (like multiple kinds of high volume print jobs). Parameterized factory methods allow further encapsulation and illustrate the ultimate utility of the factory method pattern. We will further extend the print shop example to incorporate a parameterized factory method.

Extended Example: Color Printing

The print shop has been going gangbusters! The manager decided to add the single most requested feature at the shop—color printing. We end up purchasing two new color printers: a color inkjet printer for low volume jobs, and a color laser printer for high volume printing. Unfortunately, we didn't think about color printing when first designing the application. How do we add this feature to our application? It's a good thing that we know about parameterized factory methods.

We still have the two print centers for high and low volume printing. However, we now have to specify a kind of print job. Customers can request a color or black and white print job that can be high or low volume. To indicate the kind of print job, we can pass a parameter to the `print( )` method.

New Product Classes

We add two new product classes for the new printers, called `ColorInkjetPrintjob` (Example 2-16) and `ColorLaserPrintjob` (Example 2-17). Again, we're not changing existing code, but extending the application by implementing to the `IPrintjob` interface show in Example 2-8.

Example 2-16. ColorInkjetPrintjob.as

```
package printcenters
{
    internal class ColorInkjetPrintjob implements IPrintjob
    {
    public function start(fn:String):void
        {
            trace("Printing '" + fn + "' to color laser printer");
        }
    }
}
```

Example 2-17. ColorLaserPrintjob.as

```
package printcenters
{
    internal class ColorLaserPrintjob implements IPrintjob
    {
        public function start(fn:String):void
        {
            trace("Printing '" + fn + "' to color laser printer");
        }
    }
}
```

New Creator Classes: Integrating a Parameterized Factory Method

As with the product classes, we can add a parameterized factory method to our application without modifying existing code. We won't change the original `PrintCenter` abstract class and its derived classes, `LowVolPrintCenter` and `HighVolPrintCenter`. By letting these be, we don't break the functionality of the old interface, and clients using it will continue to function. We will define a new abstract interface called `NewPrintCenter` with a parameterized factory method (Example 2-18).

Example 2-18. NewPrintCenter.as

```
package printcenters
{
    import flash.errors.IllegalOperationError;

    // ABSTRACT Class (should be subclassed and not instantiated)
    public class NewPrintCenter
    {

        public function print(fn:String, cKind:uint):void
        {
            var  printjob:IPrintjob = this.createPrintjob(cKind);
            printjob.start(fn);
        }

        // ABSTRACT Method (must be overridden in a subclass)
        protected function createPrintjob(cKind:uint):IPrintjob
        {
            throw new IllegalOperationError("Abstract method:
                                        must be overridden in a subclass");
            return null;
        }
    }
}
```

The factory method `createPrintjob( )` takes a parameter of type `uint` that represents a kind of print job (color or black and white). In addition, the `print( )` method also takes this as an additional parameter.

We next create two concrete classes (Example 2-19 and Example 2-20) that extend the `NewPrintCenter` abstract class for low and high volume printing.

Example 2-19. NewLowVolPrintCenter.as

```
package printcenters
{
    public class NewLowVolPrintCenter extends NewPrintCenter
    {
        public static const BW        :uint =  0;
        public static const COLOR    :uint =  1;

        override protected function createPrintjob(cKind:uint):IPrintjob
```

Example 2-19. NewLowVolPrintCenter.as

```
        {
            if (cKind == BW)
            {
                trace("Creating new printjob for the b/w inkjet printer");
                return new InkjetPrintjob( );
            } else if (cKind == COLOR) {
                trace("Creating new printjob for the color inkjet printer");
                return new ColorInkjetPrintjob( );
            } else {
                throw new Error("Invalid low volume print job");
                return null;
            }
        }
    }
}
```

Example 2-20. NewHighVolPrintCenter.as

```
package printcenters
{
    public class NewHighVolPrintCenter extends NewPrintCenter
    {
        public static const BW        :uint =  0;
        public static const COLOR     :uint =  1;

        override protected function createPrintjob(cKind:uint):IPrintjob
        {
            if (cKind == BW)
            {
                trace("Creating new printjob for the b/w workgroup printer");
                return new WorkgroupPrintjob( );
            } else if (cKind == COLOR) {
                trace("Creating new printjob for the color laser printer");
                return new ColorLaserPrintjob( );
            } else {
                throw new Error("Invalid high volume print job");
                return null;
            }
        }
    }
}
```

We have created a new hierarchy of related classes for the new print centers, with parameterized factory methods. The abstract class is `NewPrintCenter`, and its derived subclasses are `NewLowVolPrintCenter` and `NewHighVolPrintCenter`. Note the public static constants to identify the different kinds of print jobs. These constants are publicly accessible and should be used as the parameters passed to the `print( )` method.

Clients

Clients need to instantiate the new print center classes to access the color printers. The `print( )` method takes a parameter that specifies a color or black and white print job. They operate on different print job objects based on the passed parameters.

```
var pcNewHighVol:NewPrintCenter = new NewHighVolPrintCenter( );
var pcNewLowhVol:NewPrintCenter = new NewLowVolPrintCenter( );

pcNewHighVol.print("LongThesis.doc", NewHighVolPrintCenter.BW);
pcNewHighVol.print("SalesReport.pdf", NewHighVolPrintCenter.COLOR);
pcNewLowhVol.print("ShortVita.doc", NewLowVolPrintCenter.BW);
pcNewLowhVol.print("SalesChart.xlc", NewLowVolPrintCenter.COLOR);
```

The old print center classes, `LowVolPrintCenter` and `HighVolPrintCenter`, which implement a non-parameterized factory method, will continue to work. They'll continue to use the `WorkgroupPrintjob` and `InkjetPrintjob` print job classes without disruption. The power of the factory method design pattern to handle changing requirements is very evident in this example.

Parallel Class Hierarchies

Take a look at the class diagram for the extended print center application (Figure 2-7). You will see two distinct concrete product class hierarchies. The `WorkgroupPrintjob` and `ColorLaserPrintjob` classes represent the high volume print jobs. Likewise, the `InkjetPrintjob` and `ColorInkjetPrintjob` classes represent the low volume print jobs. In addition, knowledge about these class hierarchies is encapsulated within their corresponding creator classes. Note that the product classes cannot be accessed directly. They can only be accessed through the creator classes. The `NewLowVolPrintCenter` and `NewHighVolPrintCenter` know about their corresponding product class hierarchy. We can think of the factory method pattern as sets of *parallel class hierarchies*: the concrete creator classes and their corresponding products. This is a powerful way of encapsulating knowledge and managing dependencies within software applications.

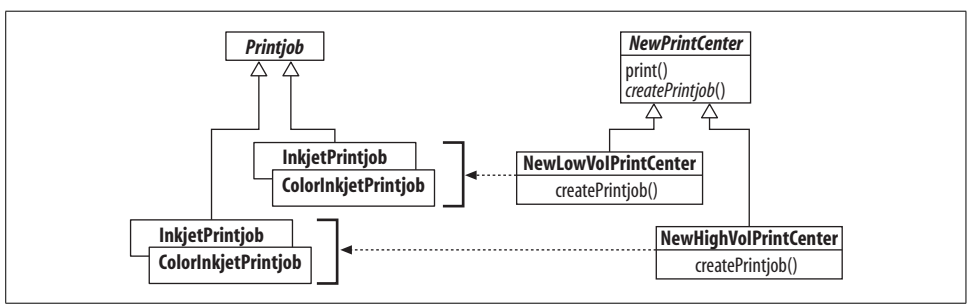


Figure 2-7. Class diagram for the extended print shop example

Key OOP Concepts Used in the Factory Method Pattern

Incorporating changes that were not anticipated in the original software design can sometimes require changes to existing code. Modifying existing code that works well should be avoided if at all possible as it can result in unintended consequences such as the introduction of new bugs. A slight change in a dependant module can result in breaking a program in several places if there's tight coupling between code segments.

Code that handles change well is possible using good OO design. The factory method pattern is an excellent solution to this recurring requirement. The factory method pattern is a solution to one of the most common reasons for tight coupling, which is caused by one class instantiating another class and using the resultant object. Of course, classes need to be instantiated—there's no way to write code that does not instantiate classes. So, what are we talking about?

We're not going to eliminate coupling caused by instantiating concrete classes. However, we can manage the dependency between classes by reducing the coupling. To do this, the factory method pattern lets you separate the *creation* of objects from their *use*.

Here lies the crucial concept. Clients can *use* objects created from another class, but the factory method handles the *creation* of objects by introducing an intermediary called a creator class between the client and the concrete class that is instantiated. The client does not have to specify the class name of the object that it wants to use because the creator class encapsulates that knowledge. This encapsulation allows managing change by extension, as the print shop example showed.

All in all, the factory method pattern allows the creation of loosely coupled designs that stand the test of changing requirements.

Example: Sprite Factory

ActionScript 3.0 introduced the Sprite class, which is a lightweight building block for interactive objects on stage. MovieClips are now derived from the new Sprite class. The factory method pattern can come in handy when developing applications that utilize sprites in Flash. Sprites are frequently added, and their behavior and appearance modified, during the course of Flash application development. Therefore, managing the dependencies between sprites and the rest of the application can be advantageous. We will create a simple example application called Shapes that manages sprite creation by introducing a factory method that enables clients to create sprites without explicitly specifying their class names.

The application shown in Figure 2-8 will simply draw four different shapes based on the `Sprite` class on the Flash stage. The first set of shapes will consist of an *unfilled* rectangle and circle. The second set will be a *filled* rectangle and circle. The example

is not a productive application, but it will serve as a springboard to the *vertical shooter game* that we'll develop later in the chapter.

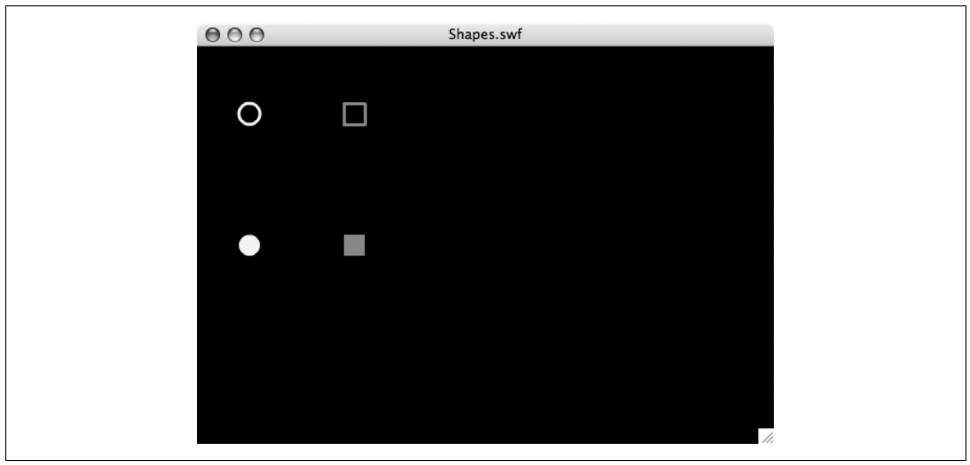


Figure 2-8. Screenshot of Shapes example stage

The Project window in Figure 2-9 shows the file structure of the Shapes example.

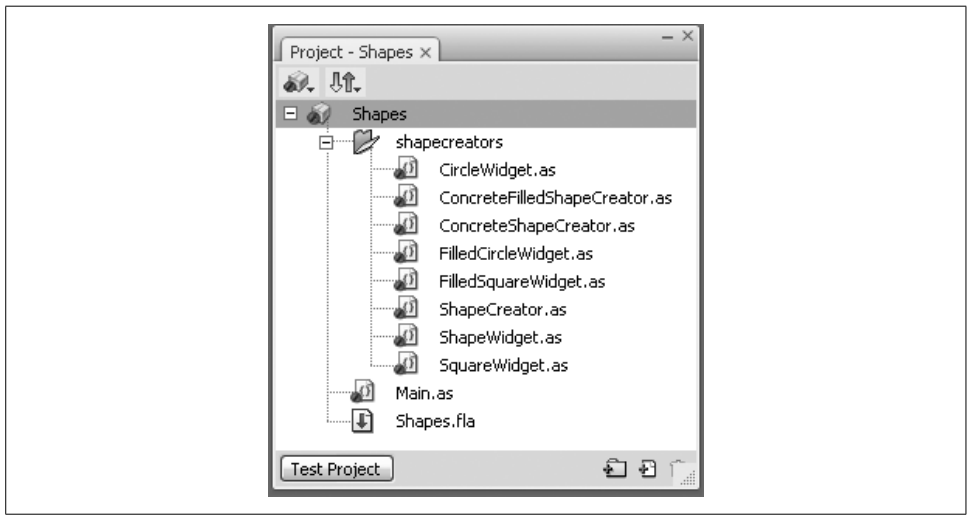


Figure 2-9. Shapes example Project window

Product Classes: Shape Widgets

The shapes on the stage (Figure 2-9) are the products in this example. These concrete shape widgets extend the `ShapeWidget` abstract class (Example 2-21). Unlike in previous examples, we define `ShapeWidget` as an abstract interface because we want

to implement default behavior common to all shape widgets in the definition. In addition, `ShapeWidget` also subclasses `Sprite`. The default behavior is defined in a method called `setLoc( )` to set the X and Y coordinates of the sprite. `ShapeWidget` also defines an abstract method called `drawWidget( )` to draw the sprite.

Example 2-21. ShapeWidget.as

```
package shapecreators
{
    import flash.display.Sprite;

    // ABSTRACT Class (should be subclassed and not instantiated)
    internal class ShapeWidget extends Sprite
    {
        // ABSTRACT Method (should be implemented in subclass)
        internal function drawWidget( ):void {}

        internal function setLoc(xLoc:int, yLoc:int):void {
            this.x = xLoc;
            this.y = yLoc;
        }
    }
}
```

Each Shape, Sprite, and MovieClip object has a property called `graphics` that is an instance of the `Graphics` class. The `Graphics` class includes properties and methods for drawing and manipulating lines and shapes including fills, colors, and patterns. The concrete classes derived from the `ShapeWidget` class are listed in Examples 2-22 through 2-25. They draw the corresponding shapes in the *constructor* using methods accessed through the `graphics` property.

Example 2-22. SquareWidget.as

```
package shapecreators {

    internal class SquareWidget extends ShapeWidget {

        override internal function drawWidget( ):void
        {
            graphics.lineStyle(3, 0xFF00FF);
            graphics.drawRect(-10, -10, 20, 20);
        }
    }
}
```

Example 2-23. CircleWidget.as

```
package shapecreators {

    internal class CircleWidget extends ShapeWidget {

        override internal function drawWidget( ):void
        {
            graphics.lineStyle(3, 0xFFFF00);
            graphics.drawCircle(0, 0, 10);
        }
    }
}
```

Example 2-24. FilledSquareWidget.as

```
package shapecreators {

    internal class FilledSquareWidget extends ShapeWidget {

        override internal function drawWidget( ):void
        {
            graphics.beginFill(0xFF00FF);
            graphics.drawRect(-10, -10, 20, 20);
            graphics.endFill( );
        }
    }
}
```

Example 2-25. FilledCircleWidget.as

```
package shapecreators {

    internal class FilledCircleWidget extends ShapeWidget {

        override internal function drawWidget( ):void
        {
            graphics.beginFill(0xFFFF00);
            graphics.drawCircle(0, 0, 10);
            graphics.endFill( );
        }
    }
}
```

Creator Classes: Shape Creators

The `ShapeCreator` class shown in Example 2-26 defines the abstract interface for the creator classes. The publicly accessible `draw( )` method is the real workhorse of this class. It calls the `createShape( )` factory method, and operates on the returned `ShapeWidget` object. The parameterized factory method `createShape( )` should behave as an abstract method, and must be overridden by the concrete creator classes.

Example 2-26. ShapeCreator.as

```
package shapecreators
{
    import flash.display.DisplayObjectContainer;
    import flash.errors.IllegalOperationError;

    // ABSTRACT Class (should be subclassed and not instantiated)
    public class ShapeCreator
    {
        public function draw(cType:uint, target:DisplayObjectContainer,
                             xLoc:int, yLoc:int):void {
            var  shape = this.createShape(cType);
            shape.drawWidget( );
            shape.setLoc(xLoc, yLoc); // set the x and y location
            target.addChild(shape); // add the sprite to the display list
        }

        // ABSTRACT Method (must be overridden in a subclass)
        protected function createShape(cType:uint):ShapeWidget
        {
            throw new IllegalOperationError("Abstract method:
                                            must be overridden in a subclass");
            return null;
        }
    }
}
```

The `draw( )` method takes four parameters: the kind of shape to be created, the stage object, and the shape's X and Y location. Even though the shape widget classes draw the shapes, they're not visible until added to the *display list* of the stage via the `addChild( )` method. Clients will pass an instance of the *Stage* object as the display object container to draw shapes on the stage. The concrete creator classes `UnfilledShapeCreator` (Example 2-27) and `FilledShapeCreator` (Example 2-28) extend the `ShapeCreator` (Example 2-26) class and implement the `createShape` factory method.

Example 2-27. UnfilledShapeCreator.as

```
package shapecreators
{
    public class UnfilledShapeCreator extends ShapeCreator
    {
        public static const CIRCLE        :uint =  0;
        public static const SQUARE        :uint =  1;

        override protected function createShape(cType:uint):ShapeWidget
        {
            if (cType == CIRCLE)
            {
                trace("Creating new circle shape");
                return new CircleWidget( );
            } else if (cType == SQUARE) {
```

Example 2-27. UnfilledShapeCreator.as

```
                trace("Creating new square shape");
                return new SquareWidget( );
            } else {
                throw new Error("Invalid kind of shape specified");
                return null;
            }
        }
    }
}
```

Example 2-28. FilledShapeCreator.as

```
package shapecreators
{
    public class FilledShapeCreator extends ShapeCreator
    {
        public static const CIRCLE        :uint =  0;
        public static const SQUARE        :uint =  1;

        override protected function createShape(cType:uint):ShapeWidget
        {
            if (cType == CIRCLE)
            {
                trace("Creating new filled circle shape");
                return new FilledCircleWidget( );
            } else if (cType == SQUARE) {
                trace("Creating new filled square shape");
                return new FilledSquareWidget( );
            } else {
                throw new Error("Invalid kind of shape specified");
                return null;
            }
        }
    }
}
```

Clients

Note that when a client calls the creator classes, it needs to pass an instance of the stage to the `draw( )` method as a parameter. If your client is the document class for a Flash document, it should have access to the stage using the `this.stage` property. In ActionScript 3.0, the stage isn't globally accessible. It can only be accessed by objects that are already attached to the display list.

```
// instantiate concrete shape creators
var unfilledShapeCreator:ShapeCreator = new UnfilledShapeCreator( );
var filledShapeCreator:ShapeCreator = new FilledShapeCreator( );

// draw unfilled shapes
unfilledShapeCreator.draw(UnfilledShapeCreator.CIRCLE, this.stage, 50, 75);
unfilledShapeCreator.draw(UnfilledShapeCreator.SQUARE, this.stage, 150, 75);
```

```
// draw filled shapes
filledShapeCreator.draw(FilledShapeCreator.CIRCLE, this.stage, 50, 200);
filledShapeCreator.draw(FilledShapeCreator.SQUARE, this.stage, 150, 200);
```

As in previous examples, you can extend this application to draw different kinds of sprites with different behaviors without much effort. We will use the sprite factory in our final example, a vertical shooter game.

Example: Vertical Shooter Game

The next application will demonstrate the usefulness of the factory method pattern when designing fast-paced action games, as many sprites on the screen are created at runtime based on user input. When an application doesn't know which objects to create until runtime, the factory method design pattern light bulb should go off in your head. We will develop a portion of a vertical shooter game in the best traditions of the original Space Invaders. The game will be based on the sprite factory example, as all interactive objects that appear on the Flash Stage are derived form the `Sprite` class. This will not be a complete game, but the parts that show the utility of the factory method pattern, such as creating different space ships, including the different projectiles that the ships shoot at each other, will be fully developed.

The game will consist of one hero ship located at the bottom of the stage (see Figure 2-10) that can be moved horizontally using the mouse. Five alien ships are placed in a row at the top of the stage. The alien ships shoot alien cannon balls (unfilled circles) and alien mines (unfilled squares that rotate). Clicking the mouse will make the hero ship shoot hero cannon balls (filled circles). In this example, we will not implement collision detection (aka hit testing) to figure out if projectiles hit the ships.

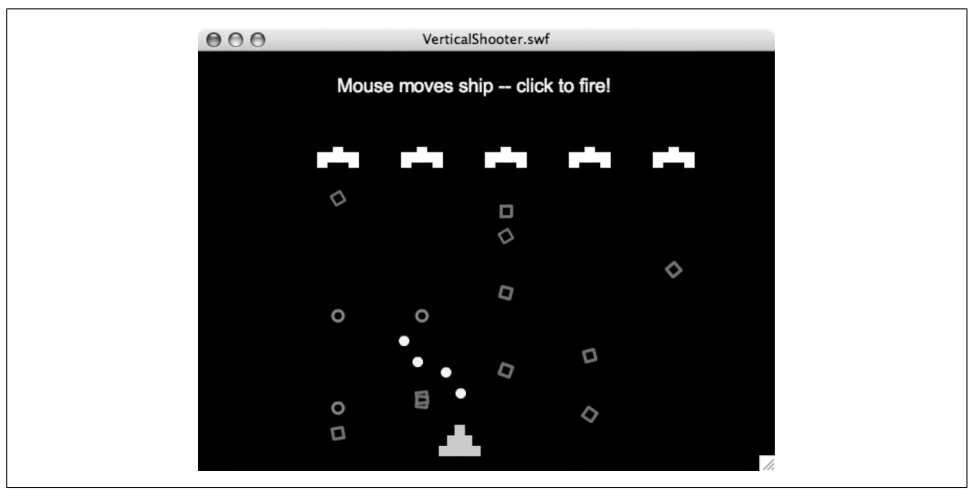


Figure 2-10. Screenshot of Vertical Shooter example showing space ships and projectiles

Figure 2-11 shows the Project window for the example. It consists of two packages that encapsulate space ships and projectiles. The `weapons` package makes use of the factory method pattern, and defines creator classes (weapons platforms) that produce different projectiles. The `ships` package uses a variation on the factory method pattern to create different spaceships.

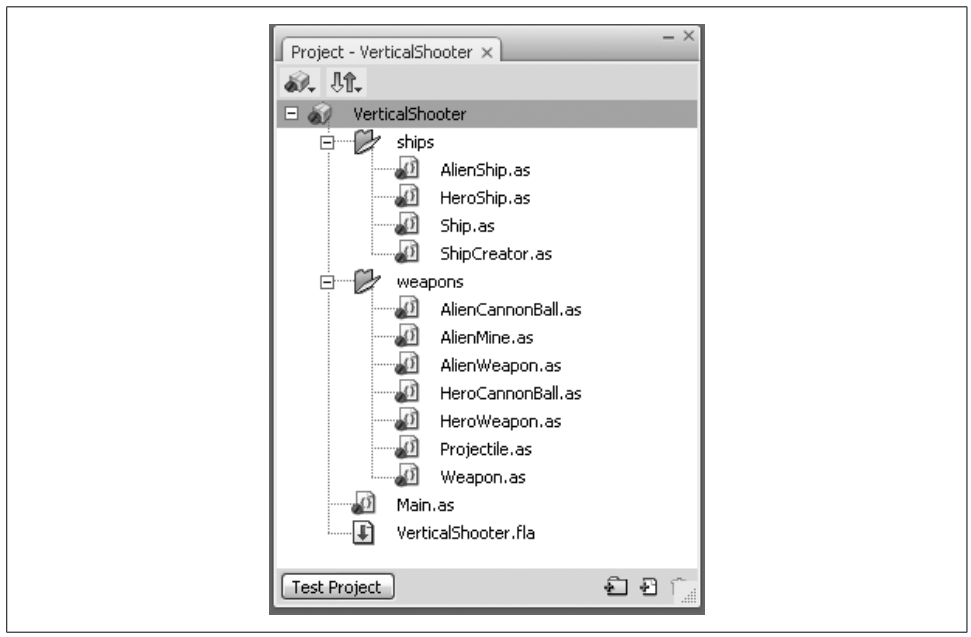


Figure 2-11. Vertical Shooter Project window

Product Classes

The example consists of two different products: projectiles and space ships. Each product is visible as a sprite on the Stage.

Projectiles

The `Projectile` class shown in Example 2-29 defines the abstract interface for the concrete projectile classes. It should behave as an abstract class and is declared as a subclass of `Sprite`. The class defines the `nSpeed` property to hold the speed of the projectile in pixels per second. The `drawProjectile()` method should behave as abstract and has to be implemented in a subclass. The class also defines several methods with default implementations. The `setLoc()` method sets the X and Y coordinates of the sprite. The `arm()` method specifies the default speed of the projectile and sets it to 5 pixels per frame. The `release()` method attaches an `EnterFrame` event handler called `doMoveProjectile()` to move the projectile vertically by `nSpeed` pixels on each `EnterFrame` event. The `doMoveProjectile()` also performs the important function of

checking if the projectile has gone beyond the top or bottom boundaries of the stage. If so, it removes the sprite object as an event listener, and removes it from the display list of the stage. Because there are no references to the projectile object at this point, the Flash garbage collector will recover the memory it occupied.

Example 2-29. Projectile.as

```
package weapons
{
    import flash.display.Sprite;
    import flash.events.*;

    // ABSTRACT Class (should be subclassed and not instantiated)
    internal class Projectile extends Sprite
    {
        internal var nSpeed:Number // holds speed of projectile

        // ABSTRACT Method (must be overridden in a subclass)
        internal function drawProjectile( ):void {}

        internal function arm( ):void
        {
            // set the default speed for the projectile (5 pixels / fame)
            nSpeed = 5;
        }

        internal function release( ):void
        {
            // attach EnterFrame event handler doMoveProjectile( )
            this.addEventListener(Event.ENTER_FRAME, this.doMoveProjectile);
        }

        internal function setLoc(xLoc:int, yLoc:int):void
        {
            this.x = xLoc;
            this.y = yLoc;
        }

        // update the projectile sprite
        internal function doMoveProjectile(event:Event):void
        {
            this.y += nSpeed; // move the projectile
            // remove projectile if it extends off the top or bottom of the stage
            if ((this.y < 0) || (this.y > this.stage.stageHeight))
            {
                // remove the event listener
                this.removeEventListener(Event.ENTER_FRAME,
                                         this.doMoveProjectile);
                this.stage.removeChild(this); // remove sprite from stage
            }
        }
    }
}
```

The default behaviors defined in the `Projectile` class are suitable for most of the derived concrete projectile classes (Examples 2-30 through 2-32). So the derived projectile classes are much simpler. They simply draw the projectile by overriding and implementing the `drawProjectile( )` method. The `arm( )` method is overridden to set a different speed. The advantage of using abstract classes should be noted here, as the code required to add new projectiles is minimal because in most cases, the desired action will be to inherit default behavior.

Example 2-30. HeroCannonBall.as

```
package weapons {

    internal class HeroCannonBall extends Projectile {

        override internal function drawProjectile( ):void
        {
            graphics.beginFill(0xFFFF00);
            graphics.drawCircle(0, 0, 5);
            graphics.endFill( );
        }

        override internal function arm( ):void {
            nSpeed = -10; // set the speed
        }
    }
}
```

Example 2-31. AlienCannonBall.as

```
package weapons {

    internal class AlienCannonBall extends Projectile {

        override internal function drawProjectile( ):void
        {
            graphics.lineStyle(3, 0xFF00FF);
            graphics.drawCircle(0, 0, 5);
        }

        override internal function arm( ):void {
            nSpeed = 8; // set the speed
        }
    }
}
```

Example 2-32. AlienMine.as

```
package weapons {

    import flash.events.*;


    internal class AlienMine extends Projectile {
```

Example 2-32. AlienMine.as (continued)

```
        override internal function drawProjectile( ):void
        {
            graphics.lineStyle(3, 0xFF0000);
            graphics.drawRect(-5, -5, 10, 10);
        }

        override internal function arm( ):void {
            nSpeed = 2; // set the speed
        }

        override internal function doMoveProjectile(event:Event):void {
            super.doMoveProjectile(event);
            this.rotation += 5; // rotate
        }
    }
}
```

Projectiles are drawn using methods available in the `Graphics` class accessible via the `graphics` property. The projectile speed is set to a negative value for the `HeroCannonBall` class as it moves from the bottom to top of the stage (Example 2-30). In contrast, alien projectiles move from top to bottom (See Figure 2-10 to see the locations of the space ships). Also note the overridden `doMoveProjectile( )` method in the `AlienMine` class. It calls the default behavior in the superclass using the *super* property, but adds a statement to make the sprite rotate. So, alien mines spin slowly as they move.

Space ships

The `Ship` class shown in Example 2-33 defines the abstract interface for the concrete space ship classes. The `Ship` class should behave as an abstract class and is declared as a subclass of `Sprite`. It defines a `setLoc( )` method to set the X and Y coordinates of the sprite. It also defines two methods without implementations that should behave as *abstract* methods. The `drawShip( )` method should be overridden and implemented to draw the ship. Similarly, the `initShip( )` method should be overridden to initialize ship behavior such as attach event handlers.

Example 2-33. Ship.as

```
package ships
{
    import flash.display.Sprite;
    import flash.events.*;

    // ABSTRACT Class (should be subclassed and not instantiated)
    internal class Ship extends Sprite
    {
        internal function setLoc(xLoc:int, yLoc:int):void
        {
            this.x = xLoc;
```

Example 2-33. Ship.as (continued)

```
            this.y = yLoc;
        }

        // ABSTRACT Method (must be overridden in a subclass)
        internal function drawShip( ):void {}

        // ABSTRACT Method (must be overridden in a subclass)
        internal function initShip( ):void {}
    }
}
```

The `AlienShip` (Example 2-34) and `HeroShip` (Example 2-35) classes extend the `Ship` (Example 2-33) class.

Example 2-34. AlienShip.as

```
package ships
{
    import flash.display.*;
    import flash.events.*;
    import weapons.AlienWeapon;

    public class AlienShip extends Ship
    {
        private var weapon:AlienWeapon;
        // available projectiles
        private const aProjectiles:Array = [AlienWeapon.CANNON, AlienWeapon.MINE];

        override internal function drawShip( ):void
        {
            graphics.beginFill(0xFFFFFF); // white color
            graphics.drawRect(-5, -10, 10, 5);
            graphics.drawRect(-20, -5, 40, 10);
            graphics.drawRect(-20, 5, 10, 5);
            graphics.drawRect(10, 5, 10, 5);
            graphics.endFill( );
        }

        override internal function initShip( ):void
        {
            // instantiate the alien projectile creator
            weapon = new AlienWeapon( );
            // attach the doFire( ) method on this object
            // as an ENTER_FRAME handler of the stage
            this.stage.addEventListener(Event.ENTER_FRAME, this.doFire);
        }

        protected function doFire(event:Event):void
        {
            // fire randomly (4% chance of firing on each enterframe)
            if (Math.ceil(Math.random( ) * 25) == 1)
            {
                // select random projectile
```

Example 2-34. AlienShip.as

```
                var cProjectile:uint =
                  aProjectiles[Math.floor(Math.random( ) * aProjectiles.length)];
                weapon.fire(cProjectile, this.stage, this.x, this.y + 15);
            }
        }
    }
}
```

Example 2-35. HeroShip.as

```
package ships
{
    import flash.display.*;
    import weapons.HeroWeapon;
    import flash.events.*;

    internal class HeroShip extends Ship
    {
        private var weapon:HeroWeapon;

        override internal function drawShip( ):void
        {
            graphics.beginFill(0x00FF00); // green color
            graphics.drawRect(-5, -15, 10, 10);
            graphics.drawRect(-12, -5, 24, 10);
            graphics.drawRect(-20, 5, 40, 10);
            graphics.endFill( );
        }

        override internal function initShip( ):void
        {
            // instantiate the hero projectile creator
            weapon = new HeroWeapon( );
            // attach the doMoveShip() and doFire( ) methods on this object
            // as MOUSE_MOVE and MOUSE_DOWN handlers of the stage
            this.stage.addEventListener(MouseEvent.MOUSE_MOVE, this.doMoveShip);
            this.stage.addEventListener(MouseEvent.MOUSE_DOWN, this.doFire);
        }

        protected function doMoveShip(event:MouseEvent):void
        {
            // set the x coordinate of the sprite to the
            // mouse relative to the stage
            this.x = event.stageX;
            event.updateAfterEvent( ); // process this event first
        }

        protected function doFire(event:MouseEvent):void
        {
            weapon.fire(HeroWeapon.CANNON, this.stage, this.x, this.y - 25);
            event.updateAfterEvent( ); // process this event first
        }
    }
}
```

Note the `initShip( )` methods in both derived classes attach event handlers to intercept events. The event handlers on the `HeroShip` class respond to `MOUSE_MOVE` and `MOUSE_DOWN` events sent to the stage. Intercepting these stage events is necessary as the hero ship should respond to mouse events even when the mouse focus is not on the sprite. In addition, the `initShip( )` method initializes the corresponding creator class for creating projectiles. The `ENTER_FRAME` event handler on the `AlienShip` class is the `doFire( )` method. It fires a random projectile from the available projectile list using the projectile creator class.

Creator Classes

We end up with two sets of creator classes corresponding to the two product types. One creator class encapsulates the creation of projectiles and the other encapsulates space ship creation.

Weapon

The `Weapon` class (Example 2-36) is the abstract interface that encapsulates projectile creation. The weapon is better described as a weapons platform that can fire different kinds of projectiles. The publicly accessible `fire( )` method calls the `createProjectile( )` factory method.

Example 2-36. Weapon.as

```
package weapons
{
    import flash.display.Stage;
    import flash.errors.IllegalOperationError;

    // ABSTRACT Class (should be subclassed and not instantiated)
    public class Weapon
    {

        public function fire(cWeapon:uint, target:Stage, xLoc:int, yLoc:int):void
        {
            var  projectile:Projectile = this.createProjectile(cWeapon);
            trace("Firing " + projectile.toString( ));
            // draw projectile
            projectile.drawProjectile( );
            // set the starting x and y location
            projectile.setLoc(xLoc, yLoc);
            // arm the projectile (override the default speed)
            projectile.arm( );
            // add the projectile to the display list
            target.addChild(projectile);
            // make the projectile move by attaching enterframe event handler
            projectile.release( );
        }
```

Example 2-36. Weapon.as (continued)

```
        // ABSTRACT Method (must be overridden in a subclass)
        protected function createProjectile(cWeapon:uint):Projectile
        {
            throw new IllegalOperationError("Abstract method:
                                            must be overridden in a subclass");
            return null;
        }
    }
}
```

The `AlienWeapon` (Example 2-37) and `HeroWeapon` (Example 2-38) classes extend the `Weapon` class (Example 2-36) and implement the `createProjectile( )` factory method.

Example 2-37. AlienWeapon.as

```
package weapons
{
    public class AlienWeapon extends Weapon
    {
        public static const CANNON    :uint =  0;
        public static const MINE        :uint =  1;

        override protected function createProjectile(cWeapon:uint):Projectile
        {
            if (cWeapon == CANNON)
            {
                trace("Creating new alien cannonball");
                return new AlienCannonBall( );
            } else if (cWeapon == MINE) {
                trace("Creating new alien mine");
                return new AlienMine( );
            } else {
                throw new Error("Invalid kind of projectile specified");
                return null;
            }
        }
    }
}
```

Example 2-38. HeroWeapon.as

```
package weapons
{
    public class HeroWeapon extends Weapon
    {
        public static const CANNON    :uint =  0;

        override protected function createProjectile(cWeapon:uint):Projectile
        {
            if (cWeapon == CANNON)
            {
                trace("Creating new Hero cannonball");
```

Example 2-38. HeroWeapon.as (continued)

```
                return new HeroCannonBall( );
            } else {
                throw new Error("Invalid kind of projectile specified");
                return null;
            }
        }
    }
}
```

ShipCreator

The concrete class `ShipCreator` (Example 2-39) encapsulates ship creation. We don't need to encapsulate knowledge about hero ships and alien ships at this point. After all, we have only one hero ship and one kind of alien ship.

Example 2-39. ShipCreator.as

```
package ships
{
    import flash.display.Stage;

    public class ShipCreator
    {
        public static const HERO        :uint =  0;
        public static const ALIEN    :uint =  1;

        public function addShip(cShipType:uint, target:Stage, xLoc:int, yLoc:int):void
        {
            var ship:Ship = this.createShip(cShipType);
            ship.drawShip( ); // draw ship
            ship.setLoc(xLoc, yLoc); // set the x and y location
            target.addChild(ship); // add the sprite to the stage
            ship.initShip( ); // initialize ship
        }

        private function createShip(cShipType:uint):Ship
        {
            if (cShipType == HERO)
            {
                trace("Creating new hero ship");
                return new HeroShip( );
            } else if (cShipType == ALIEN) {
                trace("Creating new alien ship");
                return new AlienShip( );
            } else {
                throw new Error("Invalid kind of ship specified");
                return null;
            }
        }
    }
}
```

Concrete Creator Classes

Until we encountered the `ShipCreator` class (Example 2-39), the examples defined creator classes as abstract. Concrete creator classes implement the factory method as opposed to leaving the implementation to subclasses. So, adding new products requires changing the factory method in a concrete class. Changing existing code is not as elegant a solution as extending an abstract class to accommodate changes. Concrete creator classes are useful when the design's only motivation is to decouple concrete classes from the clients that use them. When you add the possibility of changing requirements to this equation, in most cases the abstract creator classes are the more prudent choice.

Clients

We have multiple clients accessing the creator classes. Clients can use the `ShipCreator` class (Example 2-39) to place space ships on the stage.

```
// instantiate ship creator
var shipFactory:ShipCreator = new ShipCreator( );

// place hero ship
shipFactory.addShip(ShipCreator.HERO, this.stage,
                this.stage.stageWidth / 2, this.stage.stageHeight - 20);
// place alien ships
for (var i:Number = 0; i < 5; i++)
{
    shipFactory.addShip(ShipCreator.ALIEN, this.stage,
        120 + 80 * i, 100);
}
```

In addition, the hero and alien spaceships access their corresponding weapons classes to create and fire projectiles. The spaceships are the clients for the projectile classes.

Summary

Change is inevitable in software design. Requirements change during the course of application development, and, in some cases, ugly hacks are used to effect changes that the original design didn't anticipate. The antidote to this common issue is robust design that stands up to changes and modifications. The best way to handle changing requirements is to manage the dependencies between code segments. Consider the example of a `Client` class creating a new instance of a `Product` class using the `new` keyword, and saving the resulting object in a variable. This is a very common practice that creates a strong dependency between the two classes. This is also known as strong or *tight coupling*. Changes to either class will most likely propagate to the other class as well. The factory method pattern is an excellent way to manage these types of dependencies, as it introduces a firewall between classes that depend on each other. The pattern does not prevent the classes from depending on each other, but it provides a framework by which this dependency can be managed.

Other resources from O'Reilly

Related titles

Essential ActionScript 3.0
Dynamic HTML: The Definitive Reference
Ajax on Java
Ajax on Rails
Learning JavaScript
Programming Atlas
Head Rush Ajax
Rails Cookbook

oreilly.com

oreilly.com is more than a complete catalog of O'Reilly books. You'll also find links to news, events, articles, weblogs, sample chapters, and code examples.

oreillynet.com is the essential portal for developers interested in open and emerging technologies, including new platforms, programming languages, and operating systems.

Conferences

O'Reilly brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today for free.