Article 6

ActionScript Design Patterns for Rich Internet Application Development

by Steven Webster and Alistair McLeod

THE RICH INTERNET APPLICATION (RIA) is a term coined by Macromedia to describe an application with the look, feel, interactivity, and experience of a desktop application that is delivered with the same advantages and benefits of a more traditional HTML-based web application.

The development of Rich Internet Applications is being driven by the fusion of secure, stable, and performant server-side technologies such as J2EE (and ColdFusion MX, which is deployed on J2EE) and .NET, with Macromedia Flash MX providing the interactive presentation tier experience.

iteration::two is a software consultancy based in Edinburgh, Scotland that is applying its expertise in the development of Enterprise Internet Applications using J2EE technologies to the development of RIAs using a fusion of Flash MX and J2EE on the client and server. Intent on delivering an improved user experience to our clients, iteration::two have delivered RIAs to some of the largest media, entertainment, and financial services clients in Europe. Further information on the company can be found at `http://www.iterationtwo.com/`.

An example Rich Internet Application, the fictional "Bank of Edinburgh" online bank, is discussed in *Reality J2EE: Architecting for Macromedia Flash MX* (Macromedia Press, 2003), and we have more recently completed Opal, a desktop RIA providing mobile messaging for global corporations. This development is the result of twelve development months of effort, an achievement made possible only through the re-application of our experience in delivering complex J2EE web applications to the field of RIA development. The Opal application can be seen in Figure 6.1.

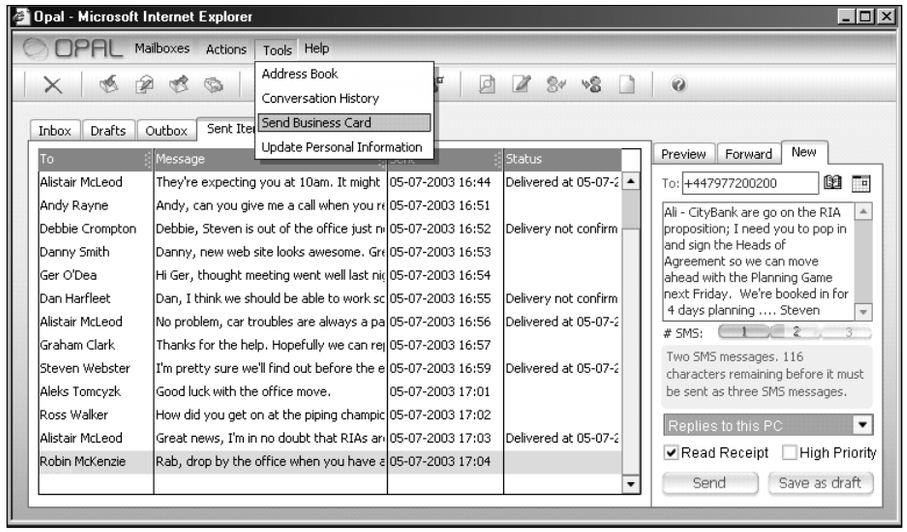

Figure 6.1 The Opal Mobile Messaging RIA from iteration::two.

Traditionally, dynamic applications built in Macromedia Flash would employ a moderate amount of ActionScript, often enabling simple integration with a server-side script. However, RIA development raises the complexity of the server-side development, which correspondingly affects the complexity of the client-side application. To deliver scalable and performant RIAs into production with predictable quality, we must address this increasing complexity.

Though the number of lines of code is a very arbitrary metric of complexity, it is worth realising that the Opal application, from which the iteration::two pattern catalog is born, comprises approximately 65,000 lines of Java code on the server and 25,000 lines of ActionScript code on the client.

In this article, we will introduce software design patterns as a design concept that has been successfully applied in software engineering for a number of years. We will learn a little about the inherent complexity of Rich Internet Application development and learn the design patterns that iteration::two have adopted from the various pattern communities, most notably the J2EE community (Alur, Deepak, Crupi, John, and Malks, Dan. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, 2003). We will learn how ActionScript 2.0, with its improved support for an object-oriented approach to software development, lends itself particularly well to implementing the iteration::two pattern catalog.

Our aim is to cover this subject through the development of a simple RIA browser for the Amazon.com book catalog. Our aim is not to demonstrate an appropriate application of RIA technologies, but rather to demonstrate the engineering of an RIA application using ActionScript 2.0 design patterns through the development of a simple application that is easily understandable. We chose Amazon for our example because Amazon has exposed the back-end logic for their business through Web Services, enabling us to focus our discussion on the client-side development without having to cover the server-side implementation.

We will introduce each pattern in turn, focusing on the development problem that must be addressed before introducing the design pattern as a candidate solution. After we are comfortable with the concept of the design pattern, we will discuss how ActionScript 2.0 can be used to implement the solution.

Technologies for RIA Development

Before delving into ActionScript 2.0, design patterns, and the development of our RIA Amazon browser using the iteration::two pattern catalog, we will first revise the technologies involved in a Rich Internet Application.

The presentation tier of a Rich Internet Application is a Macromedia Flash movie (SWF) that may be run from any device with the Flash 6 Player or greater. These devices obviously include web browsers, but they also include mobile phones, PDAs, interactive television consoles, and so on.

The "business logic" of a Rich Internet Application resides on an application server, using an appropriate server-side technology. The technologies that Macromedia has elected to support for RIA development are J2EE (Enterprise Java), CFMX (Coldfusion MX), and .NET.

So that binary data may be exchanged between the client and server, we have chosen to employ Flash Remoting MX. With Flash Remoting deployed in the application server, business methods on the server may be exposed as "services" that the Flash client can remotely invoke through ActionScript. This enables a Flash client to request that some server-side functionality be invoked using Flash Remoting, and the results are returned to the client through Flash Remoting before being prepared for display on the client.

In our experience, with appropriate architecture on the client and the server, Flash Remoting is well suited to the development of large-scale enterprise Rich Internet Applications.

Flash MX Professional 2004 introduces Web Services as another candidate technology for client-server integration that can be treated as any other service, accessed through a service-oriented architecture exposed on the server. The Service Locator pattern enables choices on the technology for client-server integration to be localized to a single class. Encapsulation of this technology choice not only abstracts the details of the technology used to the developer, but it enables alternative technologies to be swapped in without impacting the rest of the application.

An overview of a Rich Internet Application component architecture is given in Figure 6.2.

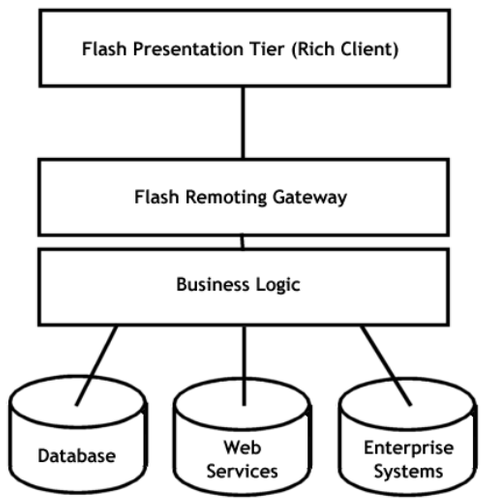

Figure 6.2 Rich Internet Application architecture using Flash, Flash Remoting, and Server-Side Application.

Now that we have discussed Rich Internet Applications and the technologies for RIA development, let's move on to consider what design patterns are and how they can help us in our development of RIAs.

Design Patterns and Pattern Catalog

The seminal book on design patterns is the aptly-titled "Design Patterns," by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, known more commonly throughout the software engineering community as the "Gang of Four" or GoF.

In their own words, "designing object-oriented software is hard, and designing reusable object orient-software is even harder. You must find pertinent objects…and establish key relationships between them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements."

In object-oriented analysis and design, a software application is described as a system of collaborating objects, each object playing a well-defined role within the collaboration. A design pattern can be considered as a solution to a recurring design problem. A design pattern is most often a description of a collaboration of objects, rather than a strategy for designing an object itself.

The Gang of Four pattern catalog contains 12 design patterns that outline common solutions to three types of problems:

- **Creational Patterns**—These address the creation of classes, such as the Singleton pattern that ensures creation of an object occurs once and once only.
- **Structural Patterns**—These address the composition of classes or objects, such as the Façade pattern, which presents a higher-level interface to a series of subsystems.
- **Behavioral Patterns**—These address the ways in which classes or objects interact and distribute responsibility, such as the Command pattern, which encapsulates a request as an object, enabling support for parameterizing requests, logging requests, and supporting undoable operations.

At the Java One conference in 2001, Deepak Alur, John Crupi, and Dan Malks of Sun Microsystems presented their own "Core J2EE Patterns." The Core J2EE Pattern catalog represents solutions to the recurring problems faced by the Sun Java Center consultants during the development of enterprise web applications. J2EE developers have successfully developed systems with

attention to the Core J2EE pattern catalog. When a collection of patterns is repeatedly used to solve a higher-level problem, a software framework emerges. In the J2EE community, many frameworks now exist as implementations of these best-practice design patterns.

At iteration::two, we have applied our experience in developing complex enterprise web applications to the development of Rich Internet Applications. The Core J2EE pattern catalog from Sun Professional Services has proven an excellent set of patterns that can be adapted for Flash MX Rich Internet Applications. We have successfully developed our own ActionScript framework, resulting from the collaboration of these design patterns.

We will now move ahead and further describe the core components of the framework used by iteration::two.

ActionScript 2.0 and Design Patterns

ActionScript 1.0, which is the ActionScript language that has existed up to and including the release of Flash MX, has a pseudo-object-oriented development model, based on the notion of objects and object prototypes.

With the emergence of ActionScript 2.0, which implements much of the ECMA4.0 standard, object-oriented language features have been introduced that enable the ActionScript developer to think of his or her system development in terms of classes, instances of objects, and the collaborations or passing of messages between these objects.

Though it has been possible to implement design patterns to a fashion using ActionScript 1.0, with ActionScript 2.0, we can now more naturally express an architectural framework as a series of collaborating design patterns.

We will now examine some of the key patterns in a framework that support the rapid development of RIAs and examine how these patterns can be naturally and effectively implemented in an object-oriented fashion using ActionScript 2.0.

We will use UML (Unified Modeling Language) notation to describe our patterns, borrowing on two important UML diagrams—the class diagram and the sequence diagram.

The UML class diagram will enable us to document patterns as classes and document those classes in terms of their relationship with other classes. The class diagram will give us a static, stationary view of the system.

Meanwhile, the sequence diagram will enable us to describe how the classes in a particular design pattern interact together and how they interact with the classes involved in other patterns. The sequence diagram will give us a dynamic, living, breathing view of the system.

Let's now jump into the actual discussion of the patterns for our Rich Internet Amazon example.

Rich Internet Amazon Application

Rather than present the patterns in an academic fashion, we will instead follow a simple example of a RIA that enables us to demonstrate the patterns in context.

We will build an RIA browser for the Amazon.com book catalog that enables us to search for relevant titles by keyword, to display the book jackets on our desktop, and to interact with the books before purchasing them from the Amazon site itself. Amazon exposes their product catalog as a set of Web Services, which enables us to demonstrate the pattern catalog without having to perform a significant amount of back-end development.

Flash MX Professional 2004 makes it possible for a Flash client to hook directly to Web Services, using data binding to bind UI controls to the results passed back from a Web Service call. However, this data binding is only supported for web services running on the same server as the one from which the Flash application was served. To access remote Web Services, such as the Amazon Web Services, a "proxy" Web Service must be supplied, which provides a local interface to the remote service. Rather than provide a Web Service proxy in this manner, we have taken the simpler approach of hiding the Web Service implementation behind a Java application running on a Tomcat application server. If you visit `http://aspatterns.iterationtwo.com/`, you can download the full source code for the application, including the server-side implementation. Alternatively, you can learn how to point your client at another server hosting the back-end.

Similarly, we have provided very little functionality on the user interface, limiting the example to the behavior necessary to explore each of the design patterns in context. This enables us to keep the focus of this article on the framework of collaborating ActionScript patterns that enable the engineering of Rich Internet Applications using ActionScript 2.0.

A screenshot of the iteration::two Amazon RIA browser can be seen in Figure 6.3.

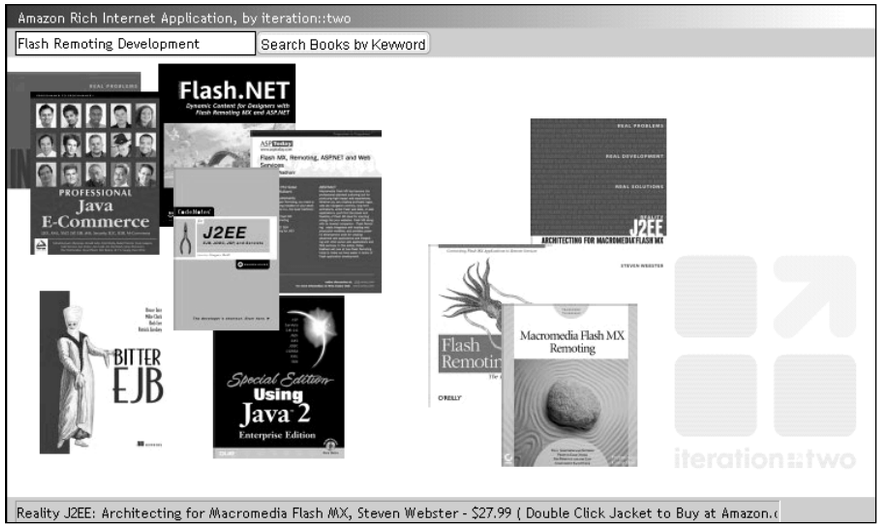

Figure 6.3 Amazon RIA browser showing results of a keyword search.

The iteration::two ActionScript 2.0 Pattern Catalog

Let's now take a look at the iteration::two ActionScript pattern catalog. This catalog represents the ActionScript implementation of some core design patterns that provide a framework for the development of Rich Internet Applications.

If we consider the "back-end" or server-side of an enterprise RIA to be the "bottom," and the presentation tier to be the "top," then we will take a "bottom-up" approach to studying these design patterns. In turn, we will look at the following patterns:

- Service Locator
- Value Object
- Business Delegate
- Front Controller
- Command Pattern
- View Helper

Let's summarise how we'll tackle each of these design patterns. For each pattern, we will first identify the recurring design problem that the pattern addresses. We'll then consider how the pattern may be implemented using ActionScript 2.0, and further consider how the pattern collaborates with other design patterns to provide a structure in our framework. Using our Amazon RIA application, we will explore some concrete examples of how the pattern can be used to solve a real-world problem.

Note

ActionScript 2.0 encourages us to elaborate the best practice of keeping all ActionScript code in external text files by packaging code into namespaces that correspond to a directory structure. All code for our Amazon RIA will exist under the com.iterationtwo.amazonria namespace in the corresponding com/iterationtwo/amazonria directories. Different patterns in the framework will reside in different packages underneath this framework, as will be seen in the class definitions.

Let's start with the first of our patterns—the Service Locator.

Service Locator

When building a Rich Internet Application, we naturally adopt a Service-Oriented Architecture (SOA). A SOA is an application architecture that consists of a set of loosely coupled business services that may be accessed by a range of clients. Assembled together, these services may be aggregated to solve a particular business problem.

As previously discussed, we have made a decision in our Amazon RIA to use Flash Remoting as our technology for client-server integration. The Service Locator class will encapsulate all the details of Flash Remoting. Should we decide at a later stage to use a different technology for passing data between client and server, the Service Locator class will be the only class that we have to change to reflect this decision.

Flash Remoting enables a Flash presentation tier to act as a client to a service-oriented architecture. For a ColdFusion developer, services may include ColdFusion pages or ColdFusion Components. A .NET developer may want to expose ASPX pages, DLL files, or .NET Web Services, while a J2EE developer might choose to expose their plain old Java Classes, Servlets, JSP, Enterprise JavaBeans, or JMX Mbeans as services. And for all these technologies, SOAP Web Services are an emerging standard that enable application functionality to be exposed to third-party clients as services.

In fact, the only thing that all these disparate services have in common is that they are remote and that, to use them, we must first locate them and connect to them.

Though Flash Remoting provides a uniform means of connecting to these disparate services, how we do that depends upon knowing their location, the type of service they are, and how a connection to that type of service is established.

For instance, to connect to a Java class that has been exposed to Flash Remoting as a service, we must know the URL of the application server on which the Flash gateway is running, we must know that the service is indeed a Java class, and we must know the fully qualified path name (such as `com.iterationtwo.amazonria.flash.AmazonService`) of that Java class.

Problem

Looking up and creating connections to a series of different services that may be implemented with different technologies is not the concern of the Flash client application developer. When developing the client-side logic for a Rich Internet Amazon browser, the developer really doesn't care whether the books written by a particular author are located using a SOAP Web Service, a Java application, or a .NET DLL. More importantly, he or she doesn't care about the different ways in which they should locate the service, depending on how the service is implemented.

The developer just wants the service handed to him or her on a plate so that he or she can call the "findBooksWrittenByAuthor" method that does exactly what it says on the tin.

This is a commonly recurring problem. This is a job for the Service Locator pattern.

Solution

The Service Locator provides a uniform means of locating server-side services for use by a client-side client. We can extend the utility of the Service Locator by ensuring that attempts are made to locate only real and existing services. The Service Locator can be created with a lookup table of known services—when an application demands the use of a new service, this service is registered with the Service Locator so that it may be available for use by the client.

As discussed previously, the name by which a service is located depends on its implementation. A plain Java object on the server is looked up by its fully

qualified class name, whereas an Enterprise JavaBean service is located by the fully qualified class name of its home interface. A Web Service may be located by the location of its WSDL file, whereas a ColdFusion component may be located by a combination of the directory from the root web directory in which it is located along with the component's filename. The Service Locator can provide a useful abstraction here—when services are registered with the Service Locator, they are given a "canonical name," such as "AmazonService." Whether "AmazonService" is a Web Service, a Java class, or a ColdFusion Component is of interest only when the service is first registered with the Service Locator. Clients of the service need only locate the service using its simpler name, "AmazonService."

ActionScript 2.0 Implementation

The Service Locator hides from its clients (usually the Business Delegate pattern, which will be discussed later) the details of locating and connecting to a service. In our Amazon RIA, this is making a connection through the Flash Remoting server to the J2EE application responsible for interrogating the Amazon catalog through Web Services.

The creation of connections is always one of the most expensive operations in distributed systems and is a well-established problem in database systems, for instance. Creating a new connection each time a service is to be called can be expensive, both in terms of memory (connections essentially eat up resources) and in speed (opening a connection to a server is usually the critical operation). A Connection Pool is a strategy used in database designs to alleviate this problem. By opening a number of connections on startup and then reusing these connections among all service requests, we can eliminate both the expense of creating connections and the bottleneck of sharing a single connection.

In our Service Locator, we have implemented a very basic Connection Pool by creating a single connection that is reused by all instances of the Service Locator. We implement this using a private static class attribute.

Should we want to implement a more complete Connection Pool strategy, it can be achieved in the Service Locator, to the ignorance of all other parts of the service. This is a perfect example of good object granularity.

Within our Amazon RIA application, the ServiceLocator class is used only in the AmazonDelegate (which is explained later in this article). When creating a ServiceLocator, we provide a reference to a responseHandler. We discuss the need for response handlers in Flash Remoting in more detail in Reality J2EE—Architecting for Flash MX. In summary, however, the response handler

is the class that contains the methods that will be called when a service call has completed and the results are available, such as a list of books in response to a keyword search.

Rich client applications, such as those developed with Flash MX presentation tiers, advertise "responsiveness" as one of the key benefits. To ensure that the application is responsive, when data is fetched from the server, the application should not "hang" until the server has passed its result back to the client. Instead, the request is made to the server, and the client application is immediately available. The server then passes data back to the client later, when it is available. This is known as an "asynchronous response," which is achieved using a response handler to receive the response at this future point in time.

The response handler strategy can be used whenever the client-server communication is asynchronous, including Flash Remoting and Web Service integration.

Note

ActionScript 2.0 introduces object-oriented features not available in ActionScript 1.0, such as private functions and private class attributes. We use these features in our Amazon RIA example class to encapsulate its implementation.

As can be seen in the ServiceLocator constructor that follows, not only do we store the response handler, but we also create a directory of service names—simple names to map to server-side services—as well as a cache for services that have previously been instantiated.

We also specify the default URL for the NetServices gateway URL. In a production-strength system, this URL would not be hard-wired into the class.

```
public function ServiceLocator( responseHandler )
{
      this.responseHandler = responseHandler;
      serviceDirectory = new Array();
      serviceCache = new Array();

      NetServices.setDefaultGatewayUrl( "http://127.0.0.1:8080/amazonria/gateway" );

initialiseServices();
}
```

The ServiceLocator constructor calls `initialiseServices()` to add the mappings of the canonical service names to their fully qualified name. In our implementation, the private function `addService()` does the actual work of storing the mapped values together.

```
private function initialiseServices()
{
        addService( "AmazonService", "com.iterationtwo.amazonria.flash.
        ➥AmazonService" );
}
```

The only other public function of the ServiceLocator is `getService()`, which enables classes using the ServiceLocator to retrieve a service for a given canonical service name:

```
public function getService( serviceName:String )
{
        if ( serviceExists( serviceName ) )
              return getServiceInstance( serviceName );
        else
              trace( "ServiceLocator: No Such Service - " + serviceName );
}
```

The `serviceExists()` method returns a Boolean to indicate whether the canonical service name has been defined to the ServiceLocator. The `getServiceInstance()` method returns the service for the canonical name and handles the creation and caching of the services. Subsequent requests for the same service will result in the service being retrieved from the cache rather than the construction of a new object and the performance hit that this incurs.

If no service exists for the provided service name, we throw an error to the Output panel. In a production-strength system, we would introduce the Exception processing available with ActionScript 2.0.

From this simple example, we can already see how the ServiceLocator encapsulates the retrieval of services using an easy-to-remember canonical name that isn't tied to the implementation of the service, enabling clients of business services to locate and use them in a simple, scalable, and maintainable fashion.

Let's move on to discuss the next design pattern—the Value Object.

Value Object

In an enterprise application, several tiers exist in the architecture. Architects will often talk of a 3-tier or even n-tier design to describe the layers of architecture that are decoupled from each other. In an RIA, we will typically have a 3-tier design, comprising our presentation tier, business tier, and integration or service tier.

The presentation tier is a Flash application implementing the user interface. The business tier will typically cross the chasm between ActionScript and the server-side technology, with Flash Remoting providing the bridge. At the

back-end, the integration tier will hook the business logic into databases, message queues, LDAP servers, legacy systems, and so on, and it will expose these as services to the business logic.

Problem

An application developer needs not concern himself or herself with the complexities of the integration tier. When asking to find all books written by Webster, the business logic developer cares little for whether these are returned as an SQL ResultSet, a collection of container-managed entity beans, or XML from an object persistence engine. Rather, the business logic developer wants "books" that have "titles," "cover images," and a "price."

Similarly, our presentation tier developer wants to build a user interface without having to worry whether changes to how author data is represented in the database requires changes to how author data is presented on the user interface.

Rather, within each tier of the application, the various developers are interested only in the "currency" of books, authors, publishers—or objects, to be more precise. They are not concerned about how these objects are represented internally at various places in the application architecture, nor do they want to be exposed to the results of a change in the underlying object representation. They care about the value of the data, not the way it is represented.

This is a commonly recurring problem. This is a job for the Value Object pattern.

Solution

The Value Object is simply a container for the data that represents an entity in the system, such as a book, and the attributes of that entity, such as the ISBN number, the author, and the publisher.

The Value Object should be able to carry this data around between application tiers. Clients should be able to populate Value Objects to pass as arguments to method calls further down the architecture. Services should be able to return results as Value Objects or as collections of Value Objects (typically as arrays).

Consequently, the Value Object can contain the attributes representing its data, methods to populate or "set" the data, and methods to fetch or "get" the data.

It is also worth considering that in a complex system, a Value Object may contain other Value Objects—a Book Value Object may contain an array of Review Value Objects, for instance, to represent the different reviews of a particular book.

ActionScript 2.0 Implementation

ActionScript 2.0's new object-oriented features enable us to implement a "proper" object model in our client side application. For example, the following is one possible implementation of a KeywordRequest Value Object that can be used as the "currency" in a request to an Amazon Service.

```
class com.iterationtwo.amazonria.vo.KeywordRequest
{
      public function KeywordRequest()
      {
      }

      public function getKeyword():String
      {
               return keyword;
      }

      public function setKeyword( keyword:String )
      {
               this.keyword = keyword;
      }

      private var keyword:String;
}
```

This class definition enables client code to construct a KeywordRequest Value Object and use it as follows.

```
var keywordRequest:KeywordRequest = new KeywordRequest();
keywordRequest.setKeyword( "harry potter goblet of fire" );
var keyword:String = keywordRequest.getKeyword();
```

Because we have defined the `keyword` attribute within the KeywordRequest class as being `private`, the compiler prohibits attempts to do the following:

```
var keyword:String = keywordRequest.keyword;
```

The above notation to access class attributes (often called the "dot-notation") is the manner in which many ActionScript developers normally write their client code. However, it is considered bad practice to have class members as public scope.

ActionScript 2.0 has introduced implicit getters and setters to help with this issue. Implicit getters and setters can be defined within a class using the following format:

```
class com.iterationtwo.amazonria.vo.KeywordRequest
{
      public function KeywordRequest()
      {
```

```
        }

        public function get keyword():String
        {
                return theKeyword;
        }

        public function set keyword( keyword:String )
        {
                this. theKeyword = keyword;
        }

        private var theKeyword:String;
}
```

Note

Due to the manner in which the implicit getters and setters are compiled, their function names cannot be the same name as an attribute of the class. For this reason, we have had to rename the `keyword` class attribute to `theKeyword`.

Defining a class with implicit getters and setters enables clients of that class to use the dot-notation to access the class attributes through those functions:

```
var keyword:String = keywordRequest.keyword;
keywordRequest.keyword = "new keyword";
```

Although the implicit getters and setters may seem a useful addition to ActionScript 2.0, the enforced renaming of attributes affects the readability of our code when we are using Flash Remoting.

Using Getters and Setters with Flash Remoting

When data is passed through Flash Remoting, the attribute names are used to determine the data to be transferred, so a server-side class has to refer to the attributes in their renamed form. This enforces an inconsistency between the attributes in the server and client-side models, which we do not consider a good practice. Consequently, we have chosen not to use implicit getters and setters in our Value Object implementations. However, so that the client-side developers can still use the notation they are most familiar with, that is, `var keyword:String = keywordRequest.keyword`, we pragmatically mark our Value Object class attributes as public in scope.

In our Amazon RIA, we have created Value Objects to capture the information required for the service request to the Amazon Web services and to capture the information returned from those services.

All Amazon search requests contain some basic information required for all requests and additional information depending on the specific request. The

creation of a base class is the ideal fit for this model. We have created the AmazonBaseRequest class:

```
class com.iterationtwo.amazonria.vo.AmazonBaseRequest
{
      public function AmazonBaseRequest()
      {
               tag = "webservices-20";
               devtag = "<amazon-devtag>";
      }

      private var tag:String;
      private var devtag:String;
      public var locale:String;
}
```

Note

A base class is a class that will be extended by other classes, known as derived classes. Derived classes inherit all functions and attributes of the base class they extend. This class hierarchy is a principal tenet of object-oriented development.

Note

As explained previously, we have decided to keep our class attributes public in scope so that clients of this class can use the "dot-notation" to access the class data.

However, two class attributes, `tag` and `devtag`, have the same value for all Amazon requests. We set their values in the class constructor and defined those attributes as `private` in the class so that classes external to the AmazonBaseRequest class cannot access or change their values.

Note

The Amazon Web Service requires the `tag` and `devtag` attributes. You can receive your own free `devtag` (also called a Token) by registering to use the Amazon Web Service at `http://www.amazon.com/gp/aws/landing.html`.

Individual service requests to the Amazon Web Service require additional information, and we have created individual Value Object classes for these. For example, to perform a keyword search, we have created the KeywordRequest class:

```
class com.iterationtwo.amazonria.vo.KeywordRequest extends AmazonBaseRequest
{
      public function KeywordRequest()
      {
      }

      public var keyword:String;
      public var page:String;
```

```
        public var mode:String;
        public var type:String;

        private static var registered:Boolean = Object.registerClass(
        "com.iterationtwo.amazonria.vo.KeywordRequest", KeywordRequest );
}
```

As can been seen in Figure 6.4, the KeywordRequest class extends the AmazonBaseRequest class, so it picks up all the attributes of the latter through the class hierarchy.

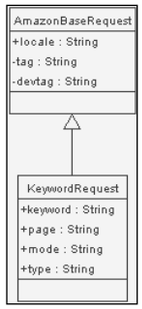

Figure 6.4 UML Class Diagram, showing the KeywordRequest Value Object class as a child of the AmazonBaseRequest parent base class.

The final attribute of the class, `registered`, requires further explanation. `Object.registerClass()` is used to assign a Symbol ID (in this case, `com.iterationtwo.amazonria.vo.KeywordRequest`) to a given function (in this case, our class constructor, `KeywordRequest`). With Flash Remoting, the Symbol ID is available as a `getType()` method in our server-side code. We use this `getType()` method to map the client-side class data onto an instance of the server-side class as specified by the Symbol ID. This provides a neat mapping between client-side and server-side Value Objects.

Note

For the J2EE backend to our Amazon RIA, we use the ASTranslator project (`http://carbonfive.sourceforge.net/astranslator` to perform Value Object translation from client to server. This is discussed in detail in *Reality J2EE: Architecting for Flash MX* (Macromedia Press, 2003), by Steven Webster.

We have used a static class attribute to execute the `Object.registerClass` method so that it is executed at most once only for each Value Object class in our application, not once per instance of class construction.

As well as the request Value Objects, we have also created Value Objects to store the results of the Amazon Web Service calls. For example, searches are returned in a ProductInfo class:

```
class com.iterationtwo.amazonria.vo.ProductInfo
{
      public function ProductInfo()
      {
      }

      public var details:Array;
      public var totalresults:String;
      public var totalpages:String;
      public var listname:String;
}
```

`Object.registerClass()` is not required for Value Objects where the data is only populated from the result of a service request.

ProductInfo contains the primitive attributes `totalresults`, `totalpages`, and `listname`, and an Array of another Value Object, Details. The Details class is also defined as a Value Object.

This "deep" Value Object enables the detailed results of a search to be fetched by clients of the ProductInfo class in various ways:

```
// Get the ProductInfo from the Service Result
var productInfo:ProductInfo = getProductInfo();
var firstDetails:Details = productInfo.details[0];
var productName:String = firstDetails.productName;
```

or

```
var productInfo:ProductInfo = getProductInfo();
var productName:String = productInfo.details[0].productName;
```

The Value Object pattern enables us to contain the data that represents an entity in the system in a single class. Use of this pattern enables client-side and server-side developers to use a common vocabulary when discussing the overall business model and simplifies the communication of information between the different layers of an application.

Now that we can locate remote services and describe system objects on the client and the server with a common vocabulary, let's study the Business Delegate pattern to see how we can make our business logic available to clients.

Business Delegate

When building a Rich Internet Application, the Flash client will inevitably request that some data is fetched, some new data is created on the server, or

some existing data is updated on the server. Additionally, the Flash client may ask that some other service is performed, such as sending an email from the server, querying the parcel track number at UPS, and so on.

Problem

When developing an RIA, the application developer does not want to be concerned with the specific implementation details of a particular business service. Furthermore, the application developer does not want to expose himself or herself to the implementation details of a business service any more than he or she wanted to expose himself or herself to the representation of the underlying model (which the Value Object pattern solved).

Solution

The Business Delegate pattern, borrowed from the Core J2EE Pattern catalog, reduces the coupling between the Flash client and business services, hiding the implementation details of the service.

Where there may be volatility in the implementation of the business service API, the Business Delegate is able to shield this volatility somewhat from the client. In the context of our Amazon RIA, we cannot assume that Amazon will not make changes to the API for searching titles. The Business Delegate pattern will ensure that the Flash user interface is isolated from such changes—the presentation tier relies only on the services exposed to the Business Delegate, delegating responsibility for the actually business service integration to the aptly-named Business Delegate.

The Business Delegate is a close friend of the Service Locator pattern—indeed, the Service Locator is likely only to be used by the Business Delegate and no other classes. Acting as a middleman on behalf of the presentation tier, when the Business Delegate is asked to provide a business service, such as returning all book titles by Steven Webster, it will likely call on its colleague the Service Locator to locate the AmazonService so that the Business Delegate may then perform the search.

Furthermore, the Business Delegate will negotiate with its clients using Value Objects. If asked for a list of books written by Steven Webster, the Business Delegate will accept an Author Value Object, representing Steven Webster as the author. The Business Delegate will then ask the Service Locator to find the AmazonService and perform its `searchByAuthor` method using the relevant information extracted from the Author Value Object. As results are returned to the Business Delegate in whichever form the particular service dictates, the

Business Delegate will construct a collection of Book Value Objects, populating each Book with the pertinent details.

Consequently, the presentation tier will "deal in Authors and Books" with the Business Delegate, ambivalent to the implementation details of the search service, the underlying representation of books and authors, and the location of the service itself. Figure 6.5 is a UML sequence diagram demonstrating the collaboration of the Value Object, Business Delegate, and Service Locator patterns to invoke a server-side service.

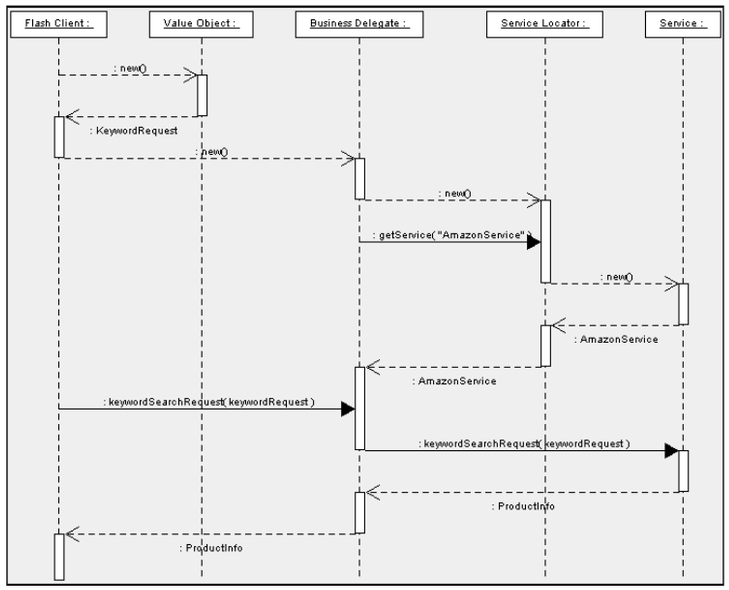

Figure 6.5 Sequence Diagram demonstrating collaboration of Business Delegate, Service Locator, and Value Objects.

A key benefit of the Business Delegate pattern is that it provides a logical place for optimization such as result caching to be performed. If a particular search pattern is likely to be a common one, such as "fetch all books in the Java Programming department," then the Business Delegate may invoke the service the first time only, caching the results. Subsequent requests by the Flash client for Java Programming books will no longer need to perform a remoting call to a remote service and can instead instantaneously return the books that were previously cached.

ActionScript 2.0 Implementation

In ActionScript 2.0, the Business Delegate is implemented in its own class, exposing the business methods as public functions.

The Business Delegate is a close collaborator with the Service Locator pattern that we introduced earlier. When the delegate is created, it immediately creates a new Service Locator and fetches any services it knows it is going to require.

```
public function AmazonDelegate()
{
      var serviceLocator:ServiceLocator = new ServiceLocator( this );
      service = serviceLocator.getService( "AmazonService" );
}
```

Furthermore, when creating the Service Locator, the delegate assumes responsibility for the handling of any results passed back by services by passing itself as the response handler to the Service Locator using the `this` reference to refer to itself.

However, having the AmazonDelegate as the response handler has implications on the use of the Business Delegate class because it restricts the application to a single action following a service call.

If we want our application to be able to call the same service in two different scenarios with different actions being performed on the results of that service call, then we cannot use the AmazonDelegate as the response handler.

We therefore refactor our AmazonDelegate class to enable the client of the class to define which object should be the response handler. Thus, the AmazonDelegate constructor dictates that clients of the class have to provide the response handler:

```
public function AmazonDelegate( responseHandler )
{
      var serviceLocator:ServiceLocator = new ServiceLocator( responseHandler );
      service = serviceLocator.getService( "AmazonService" );
}
```

With this implementation, when service calls have been made, methods on the response handler passed into the AmazonDelegate are executed on completion of service calls.

For example, our delegate could have the following business method used by clients to do a keyword search:

```
public function keywordSearchRequest( keywordRequest:KeywordRequest )
{
        service.keywordSearchRequest( keywordRequest );
}
```

The `keywordSearchRequest()` function of this class is provided with a instance of the KeywordRequest Value Object. The delegate method uses the `service` instance that was created in the AmazonDelegate constructor to call the `keywordSearchRequest()` service, passing it the `keywordRequest` Value Object.

On completion of the `keywordSearchRequest()` service, either `keywordSearchRequest_Result()` or `keywordSearchRequest_Status()` will be called, depending on the success of the service. This function call will be made on the object passed into the AmazonDelegate constructor.

We have elected not to use strict typing on the response handler so that the framework does not control which classes are allowed to be response handlers. Industrial implementations of a pattern-based framework would likely specify an interface or base class of suitable response handlers to enforce compile-time checking of the object being passed to the Service Locator.

The Business Delegate pattern provides a simple interface through which business logic can be exposed. Use of this pattern enables us to hide the internals of service lookup through the Service Locator and enables the client of the delegate to manage the response from the service as it requires.

Let's now move ahead and take a look at how we incorporate the Front Controller pattern into our architecture.

Front Controller

In a complex Rich Internet Application, the user interface is likely to service a vast number of types of requests. A simple application, sometimes called a "Flashlet" or "Applet," is likely to have a single job such as "fetch weather given zip code," or "fetch stock quote for ticker." An application of this complexity is really a "single use-case" application.

However, with an RIA that may have tens or even hundreds of use-cases, the application architecture needs to support the systematic handling of user gestures, such as menu selections, toolbar button presses, form submissions, drag and drops, and so on.

Problem

When the number of use-cases increases, the number of common system services that must be performed can scale terribly unless there is some centralized control for handling these requests.

Consider an online banking application in which all requests to perform some action on an account first require that the currently logged-in user is

authenticated. As the number of use-cases increases, for example, "Add Account," "View Transactions," and "Transfer Money between Accounts," then the number of occurrences of authentication code splattered around the architecture will increase as well.

Furthermore, as the number of use-cases increases, it becomes necessary to have a centralized means of deciding which use-cases must be executed as a result of the gestures made by the user (button clicks, menu selections) and the current state of the application.

This is a problem that we can choose to solve with the Front Controller pattern.

Solution

The Front Controller is the initial point of contact for handling a request. In a Rich Internet Application, a request can be considered a menu selection, a press of a toolbar button, the submission of a form, the dragging and dropping of an item from one area to another, and so on. A request is initiated by a "user gesture."

The Front Controller manages the handling of each request, invoking centralized services such as authentication or logging on each request. The Front Controller is then responsible for ensuring that the control logic necessary to complete a particular request is carried out.

ActionScript 2.0 Implementation

The Front Controller pattern could be implemented in a number of ways in ActionScript 2.0. We have chosen to present a simplified implementation in this article.

A first-cut implementation of the Front Controller could be as follows:

```
class com.iterationtwo.amazonria.control.AmazonController
{
      public function AmazonController()
      {
      }

      public function performAction( action:String )
      {
               if ( action == "keywordSearch" )
                        doKeywordSearch();
               else if ( action == "authorSearch" )
                        doAuthorSearch();
      }
}
```

Although this code would work as a Front Controller, it is hardly extensible. This implementation forces clients of the Front Controller to construct a new instance of the controller and to call a function each time an action is to be performed. Also, as the application grows, the `performAction()` method would grow, until it reached an unmanageable state.

A more scalable solution is to use an event-driven Front Controller. ActionScript 2.0 introduces the EventDispatcher class to help with this. Using the EventDispatcher, we can decouple the Front Controller from its clients. Rather than expect explicit function calls, the controller listens for defined events and performs actions on the arrival of those events.

We have created a new class, the EventBroadcaster, to handle the event processing on behalf of the Front Controller. The EventBroadcaster class uses the EventDispatcher to provide a mechanism to broadcast events and also to keep a record of the classes that have registered an interest in specific events. Our implementation is as follows:

```
class com.iterationtwo.amazonria.control.EventBroadcaster
{
      public static function getInstance()
      {
               if ( eventBroadcaster == undefined )
                        eventBroadcaster = new EventBroadcaster();

               return eventBroadcaster;
         }

      private function EventBroadcaster()
      {
               EventDispatcher.initialize( this );
      }

      public function broadcastEvent( eventName:String )
      {

               var event:Event = new Event();
               event.type = eventName;

               dispatchEvent( event );
      }

      private static var eventBroadcaster;

      public var dispatchEvent:Function;
      public var addEventListener:Function;
      public var removeEventListener:Function;
}
```

We have implemented the EventBroadcaster using the Singleton pattern. The Singleton pattern ensures that one and only one instance of the EventBroadcaster class can exist in the system.

Note

The Singleton pattern is implemented by having a `static getInstance()` method on the class and by making the constructor `private`. A `private static` variable in the class stores the only instance of the EventBroadcaster, which is created using "lazy instantiation" the first time `getInstance()` is executed.

By initializing itself with the EventDispatcher in its own constructor, the EventBroadcaster is furnished with the implementations of the `dispatchEvent()`, `addEventListener()`, and `removeEventListner()` functions declared at the end of the class. These functions are defined in the EventDispatcher class.

Clients of the EventBroadcaster, such as our Front Controller, can broadcast events of a given type by using the `broadcastEvent()` method. The `broadcastEvent()` method creates an instance of the ActionScript 2.0 Event class and dispatches that event to interested objects using the `dispatchEvent()` method. Objects can register their interest in an event by using the `addEventListener()` function, as follows:

```
EventBroadcaster.getInstance().addEventListener( "eventName", responseHandler );
```

In the example above, when an event "eventName" is broadcast, an appropriate method on the `responseHandler` will be invoked.

Which method is invoked on the `responseHandler` is governed by the following rules defined in the EventDispatcher class.

If the `typeof` the `responseHandler` is Object or MovieClip, the EventDispatcher first checks if the `responseHandler` has a `handleEvent()` function. If it exists, that function is called with the event object. If no `handleEvent()` function exists, the EventDispatcher will call a function on the `responseHandler` with the name of the event. In the example above, a call will be made to `responseHandler.eventName()`, passing the event object.

If the `responseHandler` is a function, that function will be invoked and will be passed the event object.

With the functionality provided in the EventBroadcaster, client classes can register themselves as being interested in specific events, and other client classes can broadcast those events.

We will now see how the Front Controller can use the EventBroadcaster to act on broadcasted events:

```
class com.iterationtwo.amazonria.control.AmazonController
{
      public function AmazonController()
      {
            EventBroadcaster.getInstance().addEventListener( "keywordSearch",
            ➥this );
            EventBroadcaster.getInstance().addEventListener( "authorSearch",
            ➥this );
      }

      public function keywordSearch( event:Event )
      {
            doKeywordSearch();
      }

      public function authorSearch( event:Event )
      {
            doAuthorSearch();
      }
}
```

The AmazonController constructor registers itself as the response handler for the events named `keywordSearch` and `authorSearch` through the `addEventListener()` function.

If another class in the application, perhaps as the result of the user pressing a "Search" push button, broadcasts the `keywordSearch` event, as follows:

```
EventBroadcaster.getInstance().broadcastEvent( "keywordSearch" )
```

then because the AmazonController has registered an interest in `keywordSearch` events, the `keywordSearch()` function of the AmazonController class will be invoked when the event is broadcast. This demonstrates the collaboration between the EventBroadcaster, which ensures that user gesture events are broadcast, and the AmazonController, which ensures that user gesture events are handled.

The collaboration of the Front Controller and the Event Broadcaster classes also enables finer control over an application. A single application could choose to have more than one controller to manage separate parts of an application (for example, if the system was modular in design with pluggable components, each component might have its own controller).

In such an application, each controller can register with the Singleton EventBroadcaster for its own set of events. For example, `controller1` might be interested in events `authorSearch` and `keywordSearch`, whereas `controller2` might be interested in events `authorSearch` and `asinSearch`.

Although the event-driven Front Controller described above is better than the function-based one described initially, it still has the same problem of extensibility. Over time, as more events are added to the system, the class will turn into a "Fat Controller," bloated with additional logic. Let's take a look now at the Command Pattern—a diet pill for a fat Front Controller!

Command Pattern

In many strategies for object-oriented analysis of a system, some form of "Use-Case Analysis" is performed. A use-case can be considered a typical interaction between one or more users and the system to carry out a specific task. In agile development, a use-case is typically a "user story," or in feature-driven development, we could call a use-case a "feature."

In the context of our Amazon RIA, a use-case might include "fetch all books by author," "fetch product specific details given ISBN," or "keyword search for books."

Martin Fowler describes refactoring as "improving the design of existing code." In an RIA, the typically large number of use-cases means that a commonly applied J2EE refactoring becomes necessary in the ActionScript world as well.

In refactoring, code that is getting a little stale, a little moldy, and is generally becoming unhealthy for us is identified by its "code smell." A common code smell in presentation-tier design is that of the "Fat Controller."

Problem

As more and more code is added to the controller to handle the different events pertaining to different use-cases, the controller grows in complexity. When any individual class in the system grows in complexity, it becomes difficult to test, difficult to maintain, and a target for failure.

Solution

Refactoring of this bad practice is achieved by introducing the Command pattern. The Command pattern enables us to "shed weight from the controller," reducing the controller's task to centralized request handling. The Front Controller handles incoming requests but acts as a delegation point with specific Command classes containing control code to which the controller delegates.

In our previous description of the Front Controller pattern, we allowed the controller itself to handle the control logic for each particular use-case by implementing an appropriate event-handling method for each event broadcast as a result of a user gesture.

By introducing the Command pattern, the Front Controller can perform that essential skill of management—delegating someone else to do all the hard work! The Front Controller can then specialize in determining what needs to be done and by whom in response to a user gesture, and then it can leave a utility class—the Command class—to actually perform the control logic pertaining to a particular use-case.

A use-case will require the invocation of business services, the handling of results of these service method calls, and the updating of the view with these results. Encapsulation of the control code that uses the Business Delegate to invoke services, that handles the Business Delegate results, and that both interrogates and updates the view is a problem solved by the Command pattern.

Another benefit emerges with this strategy. In our initial discussion of the Business Delegate pattern, the delegate invoked server-side methods and then handled the results of those methods. A side effect of this was that each call to a server-side method could only be used in a single context. If we wanted to call a `fetchBooksByAuthor()` method and display the results in both a tabular view and an interactive jacket cover view, we would need two methods on our Business Delegate, each using the results from the `fetchBooksByAuthor()` method in different contexts.

When we introduce the Command pattern, however, each command pertains to the appropriate context. We would likely have a `ViewBookListCommand` and a `ViewBookCoversCommand`. Each Command class can now invoke the same `fetchBooksByAuthor()` method on the Business Delegate and provide the handler methods for the service results. The Business Delegate is then simplified dramatically, delegating the result handling to the Command classes, rather than having to deal with results itself.

As shown in Figure 6.6, this collaboration of the Front Controller, Command pattern, and Business Delegate pattern becomes a very powerful collaboration indeed when we start to build large-scale RIAs.

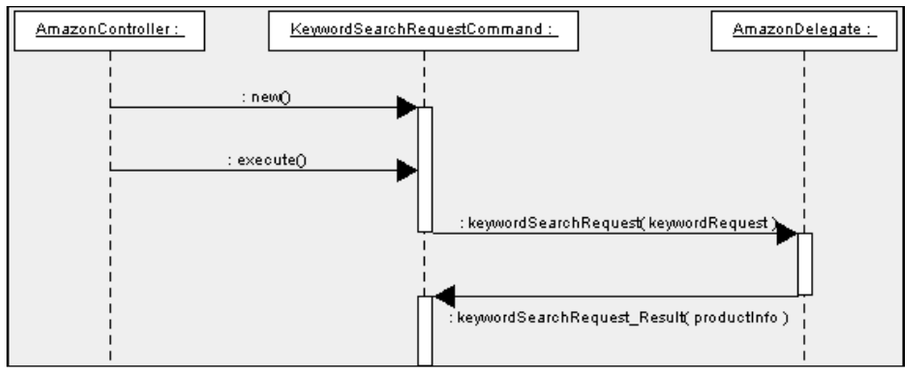

Figure 6.6 Command Pattern handling results passed back from Business Delegate.

ActionScript 2.0 Implementation

In ActionScript 2.0, we implement each command in its own class. The command's responsibility is to prepare a request for a server-side service, make the request to the Business Delegate, and handle the response. The command should prepare the response for the presentation tier, ensuring that the view is updated with the results of the server-side method call.

In our Amazon RIA, all concrete command classes must implement the Command interface. This enables us to use the Command interface in our Front Controller, which needs not concern itself with the detailed implementation of each concrete command.

Interfaces are another new feature of ActionScript 2.0. Interfaces force the addition of functions onto the classes that implement that interface.

Our Command interface is defined as follows:

```
interface com.iterationtwo.amazonria.commands.Command
{
        function execute():Void;
}
```

This interface ensures that each command class that implements the interface must include an `execute()` function.

Individual Command classes are used to perform single pieces of business logic. For example, the following is a possible definition of the command used to perform a keyword search:

```
class com.iterationtwo.amazonria.commands.KeywordSearchRequestCommand implements
➥Command
```

```
{
        public function KeywordSearchRequestCommand()
        {
              delegate = new AmazonDelegate( this );
        }

        public function execute():Void
        {
              delegate.keywordSearchRequest();
        }

        public function keywordSearchRequest_Result( productInfo:ProductInfo )
        {

              new AmazonViewHelper().setUserMessage( "Found " +
              ➥productInfo.totalresults + " items in " + productInfo.totalpages
              ➥+ " pages" );
        }

        public function keywordSearchRequest_Status( status )
        {
              new AmazonViewHelper().setUserMessage( "The keyword search failed"
              ➥);
        }

        private var delegate:AmazonDelegate
}
```

As you can see from above, the `KeywordSearchRequestCommand` implements the Command interface, which enforces the inclusion of an `execute()` method. This class relationship is shown in Figure 6.7.

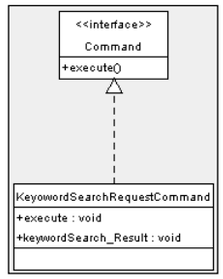

Figure 6.7 Command interface as a contract to concrete command classes.

The KeywordSearchRequestCommand constructor creates a new AmazonDelegate class and registers itself with the delegate as the response handler. This enables each command to take responsibility for the handling of service method results, such as a list of books.

The main `execute()` method of KeywordSearchRequestCommand uses the delegate created in the constructor to make a `keywordSearchRequest()` call to the server. Results from the service call are handled by the response handler:

```
public function keywordSearchRequest_Result( productInfo:ProductInfo )
```

This handler expects a ProductInfo Value Object from the server. ProductInfo contains an array of Details Value Objects. The result handler ensures that the results of the keyword search are reported to the user using a View Helper class that will be discussed in the next section.

In our Amazon RIA, failures are handled simply by tracing status messages to the Output panel in the following method:

```
public function keywordSearchRequest_Status( status )
```

Now that we have our individual commands all implementing the Command interface, we can refactor the Front Controller to uses these classes. Having the Front Controller deal with the Command interface means it needs not be aware of the internal workings of each individual command implementation. The AmazonController is able to assume that every command, because it implements the Command interface, will have an `execute()` method that it can call.

Our implementation of the Front Controller listens for appropriate events, enabling it to decide which command class it should be telling to execute. The Front Controller decides which command class to delegate to by a simple mapping between event names and concrete Command classes. This mapping means we no longer need to handle the events in their individual callback functions (for example, `keywordSearch()`). Instead, we can handle all events in the global `handleEvent()` function used by the EventDispatcher.

Here is our new Front Controller class, refactored to include the Command pattern.

```
class com.iterationtwo.amazonria.control.AmazonController
{
        public function AmazonController()
        {
              commands = new Array();
              initialiseCommands();
        }

        private function initialiseCommands()
        {
              addCommand( "authorSearch", new AuthorSearchRequestCommand() );
              addCommand( "keywordSearch", new KeywordSearchRequestCommand() );
        }
```

```
        public function handleEvent( event:Event )
        {
              executeCommand( event.type );
        }

        private function executeCommand( commandName:String )
        {
              var command:Command = getCommand( commandName );
              command.execute();
        }

        private function addCommand( commandName:String, commandRef:Command )
        {
              commands[ commandName ] = commandRef;
              EventBroadcaster.getInstance().addEventListener( commandName,
              ➥this );
        }

        private function getCommand ( commandName ):Command
        {
              var command:Command = commands[ commandName ];
        return command;
        }

        private var commands:Command;
}
```

The `AmazonController` uses `initialiseCommands()` to map the event names to the individual commands and registers those events with the EventBroadcaster. On receipt of an event in its `handleEvent()` function, the AmazonController retrieves the command delegated to manage that event and invokes its `execute()` function.

As can be seen, the Command pattern is a close friend of the Front Controller, taking responsibility for performing use-case specific work while letting the Front Controller get on with its job of capturing events and controlling the application workflow by orchestrating the commands.

The one task that our framework isn't very good at right now, however, is preparing the results that the command has returned to it from the server for the view. The View Helper pattern comes along to make the job of each Command easier in this regard. We shall now discuss the View Helper in more detail.

View Helper

One of the key tenets of user interface development is that of "separation of content and code." This idea is central to many development practices that ensure that the responsibilities of the user interface designer and the application

developer are kept discrete. Changes to the user interface in the way information is presented to the user should not have any negative effects on the implementation of the application.

Looking at this from the other direction, a trap that many developers fall into is that of embedding business logic in the presentation logic. For web application developers, placing business logic as script intermingled with HTML has been a bad habit that many have not realized until too late that they should break. Java has always promoted the separation of business logic into tag libraries and external classes, ColdFusion has promoted tag libraries and more recently ColdFusion Components (which are essentially external Java classes), and .NET now recommends the use of "code-behinds" to separate business logic from the HTML content and layout in an ASPX page.

Problem

With a Flash user interface, the temptation for developers to scatter code around the Timeline or place code on movie clips and components does little to help achieve a clean separation. As the size of an application scales up, it is important that a solution is found to solve this recurring presentation problem.

Interweaving presentation and business logic in this manner is the foundation of a system that becomes less flexible, less reusable, and less resilient to change. Modularity of the application becomes dramatically reduced, while separation of the roles in the development team becomes increasingly difficult.

Solution

The view (user interface) should be responsible only for the formatting of data. All responsibility for the fetching of data, the processing of data, and the preparation of data in a model ready to be consumed by the view should be implemented in a standalone class known as a View Helper. The View Helper stores an intermediate data model and serves as an adapter for the business data on behalf of the view.

The View Helper is able to decouple an application's presentation tier from the business tier. A business tier service call, such as "fetch book information by title," may be used in several different contexts. A shopping cart view may want simply to display a summary of the book information, whereas a product view may want to display a detailed page with all the information about the book. Specific View Helpers, such as a ShoppingCartViewHelper or ProductViewHelper, can extract the appropriate information from a book object and set the various elements on the page—dynamic text fields or movie

clips containing images, for instance—accordingly, preparing the same model for different views.

Additionally, it is often the case that to invoke a service call, we need to fetch some information from the view. For instance, to search for a book by author, we need to take the author's name from somewhere on the view. In a search page, the name may be provided in a search box, whereas a product view may make a hyperlink out of the author's name to initiate the search. In this scenario, the View Helper would be used to fetch the author's name from a particular view.

Described above, the View Helper forms a contract between the user interface developer and the application developer. The application developer doesn't care how to display the information pertaining to a product or how to fetch the author's name to perform a search. Likewise, the user interface developer doesn't care how a book is stored on the server or how it is fetched; he or she simply cares that it has a title, an author, a jacket cover, and an ISBN number, so that he or she can display its details on screen.

This contract between the user interface and the business logic is achieved using the View Helper pattern.

ActionScript 2.0 Implementation

In our Amazon RIA, we create the AmazonViewHelper class to retrieve information from the user interface to pass to service requests or to display book information back on the user interface.

Part of its implementation is as follows:

```
class com.iterationtwo.amazonria.view.AmazonViewHelper
{
        public function AmazonViewHelper()
        {
        }

        public function getKeywordRequest():KeywordRequest
        {
              var keywordRequest:KeywordRequest = new KeywordRequest();

              keywordRequest.keyword = getSearchTerm();
              keywordRequest.page = "1";
              keywordRequest.mode = "books";
              keywordRequest.type = "heavy";

              return keywordRequest;
        }

        public function setUserMessage( userMessage:String )
```

```
        {
            _root.userMessage.text = userMessage;
        }

        private function getSearchTerm():String
        {
            return _root.searchField.text;
        }
    }
```

The `getKeywordRequest()` function in AmazonViewHelper is used by the AmazonDelegate to create and populate a KeywordRequest Value Object. This Value Object is required to perform a keyword search service. In our example implementation, we have hard-coded some of the values while retrieving the search text from the entry field on the user interface via the `getSearchTerm()` private function.

The `setUserMessage()` function is used to update a status message on the user interface and can be called by any Command class to advise the user as to the results of any user gestures.

The View Helper pattern abstracts the user interface implementation from the command performing the business logic. It enables the creative designer and client-side ActionScript developer to work in tandem, with the View Helper defining the "contract" between the members of the team.

RIA development necessarily involves collaboration between the user interface designer and the developer responsible for implementing the Command classes in ActionScript. On a large-scale RIA development, it is unlikely that these will be the same person. It is our experience at iteration::two that the Command class developer will create the View Helper classes and provide empty methods that he requires to fetch information from the user interface or to populate the user interface. The user interface designer can consequently "flesh out" these methods to work with his or her particular user interface design. If the design changes, then the View Helper methods can be updated, and the Command classes remain oblivious to the user interface changes.

This also makes it much easier to test the command logic independently of the user interface. We discuss test-driven development and unit-testing of ActionScript in *Reality J2EE: Architecting for Macromedia Flash MX* (Macromedia Press, 2003), and we encourage the reader to adopt this practice.

Pattern Framework in Action

So far we have looked at a small number of design patterns in the context of

RIA development and discussed the problems that each pattern circumvents. Remember that a pattern is a recognized solution to the problem, but it says nothing about the implementation of that solution. In developing your own RIAs, it is likely that you will use the implementations given here as starting points for your own implementations of the patterns.

The collaboration of the patterns given here is the makings of a framework for the rapid development of Rich Internet Applications. Let's quickly summarize how the patterns all interoperate, from the front to the back.

Everything starts with a user gesture; perhaps the user has entered a keyword into a search box and clicked on the search button. Clicking the button generates an event that has been previously registered with the EventBroadcaster class. This event now arrives at the AmazonController, which is acting as a single point of contact for all user gestures. The controller recognizes the event and dispatches the responsibility for handling that event to a specific Command class, calling the `execute()` method on the Command class. The Command class, recognizing that it's handling a keyword search, asks for help from the appropriate AmazonViewHelper to get the keyword that was typed into the search box on the UI. The Command class then creates a KeywordRequest Value Object, which it passes as an argument to the `keywordSearchRequest()` method on the AmazonDelegate. This method on the Business Delegate requires the Amazon Service, which the Business Delegate locates simply by asking the ServiceLocator class for "AmazonService." Oblivious to whether this is a Web Service, a Java class, or even a dummy back-end being used during development, the Business Delegate uses the KeywordRequest Value Object to request a list of matching books.

The service on the server-side returns a ProductInfo Value Object. Because the Business Delegate might be asked to use the keyword search method in many different contexts, rather than make any guesses about what it should do with the results it has, it simply passes them back to the Command class that called it. The Command class, on receiving the results, then uses an AmazonViewHelper class to update the view to the search results page before using a method on the View Helper to set the results to those it has received from the Business Delegate.

From clicking "search" and having typed in an author, our pattern-based framework has collaborated to fetch the results from the server and display them back on the UI.

More importantly, with this framework in place, development can proceed at an increasing pace. Each new use-case requires the creation of a Command class, reusing or creating Value Objects and methods on the Business Delegate

as required.

Though our application may grow in complexity, our code doesn't.

What We Have Learned

Developing enterprise RIAs needs not be complex when the power of ActionScript 2.0 is leveraged to implement a collaboration of tried-and-tested software design patterns.

The pattern framework that has been presented is a loose coupling of patterns—each Rich Internet Application that you develop may use a simplified pattern framework or may demand that other design problems are solved using other design patterns, such as the Mediator, the Observer, or an implementation of the Factory pattern. It is as important to know when not to use a design pattern as it is to know when one can improve the design of your existing code. Refactoring your Rich Internet Application toward a pattern-based architecture is a way of ensuring that your intent is expressed clearly.

There are numerous other considerations in the correct engineering of Rich Internet Applications, many of which are solved by features of the Flash MX Professional 2004 environment, and many of which are solved simply through the re-application of experience gained in traditional enterprise software development.

The Amazon RIA developed in this chapter, and the full source code and Flash movies, are available for download at `http://aspatterns.iterationtwo.com/`, where the team at iteration::two openly share their experiences in Rich Internet Application engineering.

We hope that many of you share our experiences through the web site and are encouraged to explore how ActionScript 2.0 and design patterns can help you to better engineer your own Rich Internet Applications.

Further Reading

Webster, Steven. *Reality J2EE: Architecting for Macromedia Flash MX*, Macromedia Press, 2003.

Alur, Deepak, Crupi, John, and Malks, Dan. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, 2003.