Java design patterns 201

Skill Level: Introductory

Paul Monday
Developer

09 Apr 2002

Design patterns extend far beyond those described by the famous Gang of Four. In this tutorial, you will find out just how much. Veteran developer Paul Monday begins his discussion by exploring resources that newcomers to the study of design patterns often miss. Then he uses design patterns from these resources to implement a simple application. Finally, he switches his focus to how design patterns can help you to better understand software design and guides you through the reverse-engineering of a piece of technology, focusing on how it works from the perspective of patterns.

Section 1. About this tutorial

What is this tutorial about?

The *developerWorks* tutorial series on design patterns began with the tutorial "Java design patterns 101," which explored the history and common structure of published design patterns for the Java language. As a general introduction, that tutorial focused on some of the most well-known design patterns, those devised by the so-called Gang of Four. The Gang of Four (GOF) patterns apply to general computing problems and can be implemented for a variety of applications and systems. The GOF patterns are ubiquitous: in any modern, well-designed system, you will likely see one or more of these patterns implemented. Many people consider themselves well-versed in object-oriented design patterns if they know the GOF patterns. But design patterns extend far beyond the GOF.

In this tutorial, you will find out just how much you don't know about design patterns. We'll start by exploring a number of resources that newcomers to the study of design patterns often miss. These resources provide patterns that are invaluable to various

domains of computing, such as business applications, Web applications, and even Web design. Next, we'll use design patterns found through some of these resources to implement a simple application that allows a user to order movies from a Web site. After we've implemented the movie-ordering application, we'll move into the third part of the tutorial, where we'll focus on how design patterns can help us to better understand software design. To see for ourselves how the knowledge of design patterns translates to deeper understanding of software, we'll reverse-engineer a piece of technology, focusing on how it works from the perspective of patterns.

By the end of this tutorial, you will have a much greater appreciation of the wealth of knowledge available in design patterns. You will also know how to locate and disseminate the knowledge given to us in the GOF patterns, as well as several Java design pattern implementations outside of the realm of the GOF. Throughout the tutorial, we'll emphasize the ways that design patterns can assist you in learning about software.

Should I take this tutorial?

This tutorial expands upon the Java design patterns 101 tutorial, examining a broad range of design pattern resources and several specific design patterns in detail and implementation.

Although this tutorial is entirely Java-based, design pattern documentation outside of this tutorial is often in C++. To extend your knowledge of the patterns presented here, you will likely need an average skill level in C++.

See Resources for a listing of tutorials, articles, and other references that extend the material presented here.

Code samples and installation requirements

We'll use the following technologies and resources to complete the exercises in this tutorial:

- A heavily referenced set of text and online resources for design pattern documentation.
- The Java 2 platform, Standard Edition to implement the tutorial's two simple business pattern examples.
- A standard editor and a JDK to compile and run the examples.

- Tomcat 4.0.3 Servlet Engine to deploy the Web application and JSP Tag library examples. (Note that Tomcat has all the classes you will need for the J2EE portion of this tutorial.)

See Resources to download any of the above technologies that you do not have. I also suggest that you download the binaries and source code for the examples before you start the tutorial.

Section 2. Uncovering pattern resources

Uncovering pattern resources overview

The three most important elements to working with design patterns are:

- Knowing where to find design patterns
- Knowing how to apply design patterns to your software endeavors
- Recognizing when a design pattern appears in someone else's software

With a little practice, you'll find that it's fairly easy to locate a design pattern that seems applicable to your particular problem domain. In fact, you'll often find that there are too many applicable patterns for your problem domain. Once you've gathered a list of all of the potential patterns, deciding which one most aptly applies to your software design can be a challenge. When it comes to working with patterns, experience is key. With experience you'll learn where to look for patterns for a particular situation, and you'll learn how to blend a pattern into your software solution. As you advance in your career and experience as a software developer, you'll also learn to intuitively recognize the patterns in other people's designs.

In this section of the tutorial, we'll explore the ways you can extend your knowledge of design patterns beyond the GOF patterns. Specifically, we'll discuss a number of resources and techniques for tracking down design patterns for different problem domains. You'll find links for all the resources referenced here in Resources.

Online resources

Design pattern resources are available online and in book form. Oddly enough, the

online resources are some of the most difficult to track. When searching for patterns online, you'll find yourself visiting three types of Web sites:

- Corporate-sponsored sites
- Professional sites
- Educational sites

We'll start with a look at what each type of site has to offer.

Corporate-sponsored sites

Technology companies such as IBM and Sun Microsystems invest heavily in research and design, and often have a vested interest in documenting design patterns for their clients and customers.

Consider IBM's Patterns for e-business and Sun Microsystems's Java BluePrints. Both sites provide a substantial amount of information about design patterns. The patterns you'll find on these sites mirror the strengths of each company. IBM takes a holistic view of applications and their constituent parts, so the patterns you'll find on the IBM site emphasize integration from the ground up. Sun's patterns, like the company itself, are ingenious and technology oriented. Both sites offer critical resources for developing your software based on design patterns.

With further digging, you'll find that the patterns presented on many corporate sites are available in book form. Patterns are supposed to be stable entities. By definition, a pattern must be seen in software design three or more times to establish itself as a pattern. Therefore, any book about design patterns can be expected to have long shelf life.

Professional sites

Professional Web sites are typically built by consultants. A professional site is meant to show off the technological prowess of the individual who built it. It's rare that you'll find new patterns on such a Web site (although there are some notable exceptions), but you will find plenty of links to design pattern sites and resources.

Brian Foote's Web site offers an astonishing amount of information about design patterns. He has cataloged over 25 original patterns on his site, and published his own thoughts regarding patterns. Brian has also put out his design patterns reading list, which includes all the books covered in this tutorial plus about 20 more.

Other sites have sets of tools that are used for a particular domain, typically the domain that the consultant specializes in. One example of a more vertical site is Martijn van Welie's patterns site. Martijn is a Web usability specialist, therefore he has documented common UI patterns that aid him in his work.

Educational sites

The last category of online site is the ".edu". Educational sites are the most eclectic of the sites. Whereas with the professional sites you expect great content or links, educational sites vary widely in their offerings. Most educational sites with design pattern content center around the GOF patterns, since they are the most widely taught in college classes and the most widely explored by students.

Some educational sites do offer valuable content, especially for beginners. But you may have to dig a little deeper to find exactly what you're looking for in the educational domain.

Books

With all the online resources available -- my Google search on "Software Design Patterns" yielded 866,000 hits -- you might wonder why you would ever need to pick up a book. Simply put, books are the most efficient mechanism for uncovering patterns. If you're serious about patterns, you should probably read several patterns books a year.

In general, books about design patterns can be placed in the following categories:

- Books that implement patterns documented in other books
- Books that document original and undocumented design patterns
- Books that use design patterns to address a specific topic (such as concurrent threads or software architecture)
- Books that help you become better at building and recognizing patterns

Given the plethora of books available, we'll take a minute to discuss the criteria for determining which ones belong in your library.

Building a library

Books that implement patterns documented in other books are helpful when you're

just starting out. They'll help you get your arms around the concept of design patterns and how they apply to a given programming language. In general, however, you can get the same information on the Web, especially at the .edu sites. So, you may want to limit your selections from this category of books to one or two.

Books that document new or undocumented patterns are the most powerful asset in your library. Having ready access to such books lets you browse through pattern names, intent statements, and motivation statements at will. Frequently, your patterns research will be ad hoc, taking the form of "oh yeah" statements such as:

- *Oh yeah, I remember seeing a pattern like this in GOF!*
- *Oh yeah, I remember seeing a pattern like this, now where was it?*

Books facilitate this kind of thinking.

Building a library, continued

Books that use patterns to address a particular topic help you organize the ocean of patterns. These books focus on using design patterns to meet a specific goal, such as building a software architecture using patterns, or concurrent programming on the Java platform. Purchase them as you need them.

The final category of books is instructional. These books are geared to helping you become more advanced in your use of design patterns. They concentrate on such topics as the underlying dynamics of design patterns; the use of patterns to prevent the duplication of past mistakes; and how to know when a pattern is merely bending versus when it has truly broken.

Read these books to gain experienced instruction in working with design patterns. You should read at least one or two of them before you attempt to deploy a large-scale enterprise application.

Books summary

Once you begin digging, you'll find an abundance of information about design patterns, both online and in book form. Web sites cover a broad variety of topics and granularities of knowledge pertaining to design patterns, often very raw in format and presentation. While it may be time consuming to find exactly what you're looking for online, a good site can yield a treasure trove of information.

Books offer information about design patterns in a more polished and organized

form. Books tend to come in four forms: those that merely implement existing patterns; those that catalog new and existing patterns; those that use design patterns to teach you about a specific programming issue; and those that develop your knowledge of the practice of patterns. The most valuable books for the long term fall into the last three categories.

Section 3. Patterns at work

Patterns at work overview

In this section and the two that follow we'll uncover, evaluate, and use several design patterns to build a Web-based application. This application will allow users to order movies from a Web site. We'll use four design patterns to build the application: two business patterns, one Web presentation pattern, and one J2EE pattern. We'll start with a look at the business patterns.

Business patterns

Patterns specific to problems in vertical business applications can be hard to find. Fortunately, two books currently on the market, *Patterns for e-business: A Strategy for Reuse* and *SanFrancisco Design Patterns: Blueprints for Business Software*, focus almost entirely on business patterns. (See Resources for details on these titles.)

The business patterns each book describes vary considerably in terms of granularity. The first, *Patterns for e-business*, presents coarse-grained architectural patterns. These patterns are geared to helping you choose a path for structuring and architecting an e-business system. Because these patterns are architectural in nature, we won't implement any of them here. You might benefit from further study of these patterns, however, with particular attention to how the architectural patterns differ from a lower-level design pattern.

The patterns found in *SanFrancisco Design Patterns: Blueprints for Business Software* are lower-level design patterns, structured similarly to the GOF patterns. Two of these patterns, the Property Container pattern and the Simple Policy pattern, will be the basis of our movie-ordering application.

The Property Container pattern

Property Container is a *foundational* pattern. Its function is to ensure that an application, once built and deployed, can be dynamically extended.

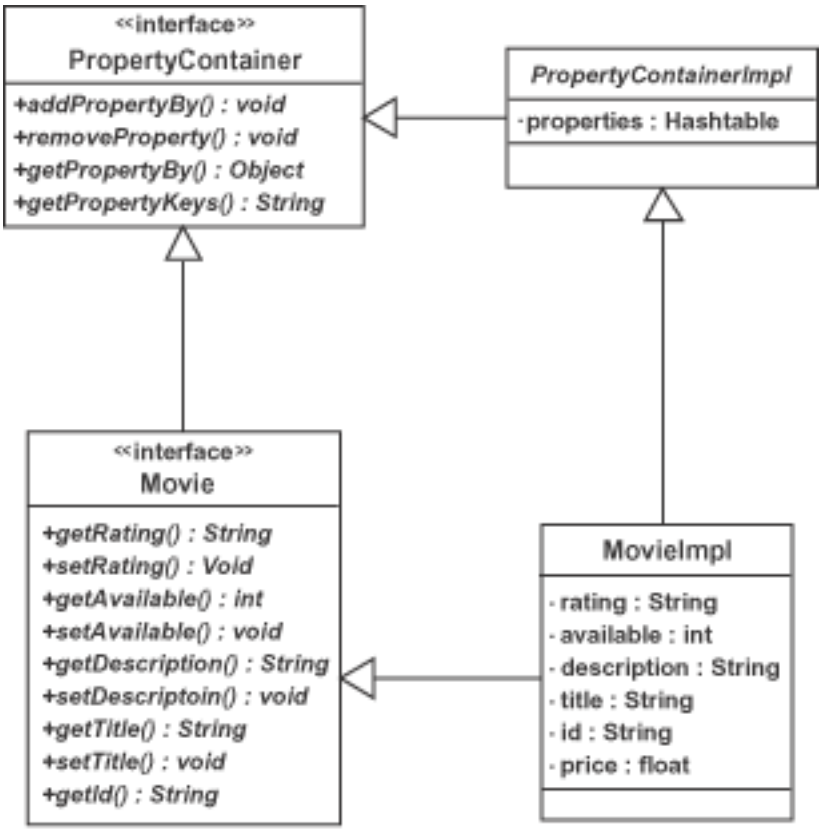
Consider the movie-ordering application. When designing a class to represent a movie, it's unlikely that you would concern yourself with all the attributes to describe a movie. But as a conscientious designer, you would want to ensure that the movie class could be extended with additional data as needed. Some extension mechanisms require a programmer to recompile the code and redeploy the classes to be extended. Property Container, on the other hand, gives us a mechanism to dynamically extend an object with additional attributes at run time. Of course, creating the Property Container is only the beginning; the application may also require modules that explicitly take advantage of the new property once it's been added.

A variety of mechanisms exist for making the `Movie` class a Property Container. Ours will be a fairly simple approach: we'll create a standard interface for the Property Container, then extend or implement that interface for all interfaces and classes that we want to be able to modify at run time.

In the next several sections we'll design a Property Container interface, then extend that interface to define the Movie interface that is the basis of our movie-ordering application. We'll also work with the concrete class implementations for the Property Container and Movie interfaces.

Designing a Property Container

In the following UML class diagram, notice that we have created a very simple Property Container interface and an abstract implementation from which classes can inherit:



As you can see, there are four methods to add, remove, and query properties in the Property Container. The properties are identified by a `String` key. In the concrete implementation we'll simply use a hashtable to store the properties as they're added to the `Movie` class.

We're working with a very simple Property Container design and implementation here, but the actual pattern has some critical extensions. These extensions illustrate the different ways the pattern can function in a more robust and extensive implementation. For example, if an object is in a hierarchy of run-time objects, the Property Container pattern allows for the traversal of that run-time hierarchy.

One example of a run-time hierarchy is a department contained within another department, which in turn is contained within a company. A query against a department's dynamic properties for an accounting code would make use of the hierarchical Property Container. The Property Container pattern would define the behavior for traversing the containment hierarchy for the specific property of the accounting code.

Implementing a Property Container

Our implementation of the Property Container will be very straightforward. To get started, take a look at the Property Container code below:

```
public abstract class PropertyContainerImpl
      implements PropertyContainer, Serializable
{

      protected Hashtable ivProperties = new Hashtable(2);

      public PropertyContainerImpl() {
      }

      /**
       * Add a property associated with a token name.
       * If the token already exists, the value will be replaced.
       * If the token does not exist, it will be added with the value.
       * @param value is an object that cannot be null
       * @param token is a key that can be used to retrieve the value
       */
      public void addPropertyBy(Object value, String token) {
          if(value==null || token==null) return;
          if(ivProperties.containsKey(token)){
              ivProperties.remove(token);
          }
          ivProperties.put(token, value);
      }

      /**
       * Retrieve a value by a particular token.
       * @param token is a key that can be used to retrieve the value
       * @return Object is the value associated with the token.  It
       *   will not be null.
       */
      public Object getPropertyBy(String token) {
          if(token==null) return null;
          return ivProperties.get(token);
      }

      /**
       * Retrieve all property keys currently in use.
       * @return String[] is an array of all valid token names.
       */
      public String[] getPropertyKeys() {
          String keys[] = null;
          synchronized(ivProperties){
              int s = ivProperties.size();
              keys = new String[s];
              Enumeration e = ivProperties.keys();
              int i = 0;
              while(e.hasMoreElements()){
                  keys[i] = (String)e.nextElement();
                  i++;
              }
          }
          return keys;
      }

      /**
       * Remove a value associated with a particular token.
       * @param token is a key associated with a value that was added
       */
      public void removeProperty(String token) {
          if(token==null) return;
```

```
            ivProperties.remove(token);
        }

}
```

As you can see, our Property Container uses a `Hashtable` to store property values. The key is determined by the person adding the property. You will typically want to standardize the keys in some way, such as the package name and class responsible for adding the property.

Implementing the Movie class

The next step is to use the Property Container interface and abstract class implementation to implement the Movie interface. We'll use classic, simple JavaBean patterns to devise the properties for our Movie interface implementation. The implementation, `MovieImpl`, will inherit from the abstract `PropertyContainerImpl` implementation in the previous panel.

```
public class MovieImpl extends PropertyContainerImpl
      implements Movie, Serializable
{

      private int available;

      private String description;

      private float price;

      private String rating;

      private String title;

      private String id;

      public MovieImpl() {
      }

      public int getAvailable() {
          return this.available;
      }

      public void setAvailable(int available) {
          this.available = available;
      }

      public String getDescription() {
          return this.description;
      }

      public void setDescription(String description) {
          this.description = description;
      }

      public float getPrice() {
          return this.price;
```

```
        }

        public void setPrice(float price) {
            this.price = price;
        }

        public String getRating() {
            return this.rating;
        }

        public void setRating(String rating) {
            this.rating = rating;
        }

        public String getTitle() {
            return this.title;
        }

        public void setTitle(String title) {
            this.title = title;
        }

        public String getId() {
            return id;
        }

        public void setId(String id) {
            this.id = id;
        }

}
```

You should find no surprises in our `MovieImpl` class implementation. Recall that all the behavior for the Property Container is actually contained in the `PropertyContainerImpl` superclass. The remainder of the code is simply JavaBean patterns, which fill in the primary attributes of the `MovieImpl` class.

Property Container demo

After creating several `Movie` objects and placing them in a vector, we may find that some of the movies have additional attributes. For example, one of the movies may not be released yet. Rather than extending the class, we can add the `releasedate` property using the inherited `addPropertyBy` method on the `Movie` class, as shown here:

```
            Movie temp = (Movie)movies.elementAt(1);
            temp.addPropertyBy(
                new GregorianCalendar(2002, 5, 22),
                "releasedate");
```

A test program that populates two `Movie` objects and adds a release date property is in the package

`com.stereobeacon.patterns.business.propertycontainer`. To run the program, use the `com.stereobeacon.patterns.business.propertycontainer.TestPropertyContainer` class.

Pros and cons of the Property Container pattern

There are some disadvantages to using the Property Container pattern. With the off-the-shelf property container implementation, you lose strong typing. Also, the interface to the class is not entirely descriptive of the contents, and you'll probably have to modify a user interface to take proper advantage of the added attribute. If you're using the method of serialization to a database, the Property Container itself could cause some headaches as well.

Despite the drawbacks, the Property Container pattern is well suited for certain types of applications, especially when coupled with the ability to traverse a containment hierarchy.

We'll put the Property Container aside for now, as we concentrate on integrating another essential pattern into our application.

The Simple Policy pattern

The Simple Policy pattern is similar to the GOF's Strategy design pattern, but specialized to deal with the realities of business. The Simple Policy pattern allows us to dynamically set and enforce access policies for our application.

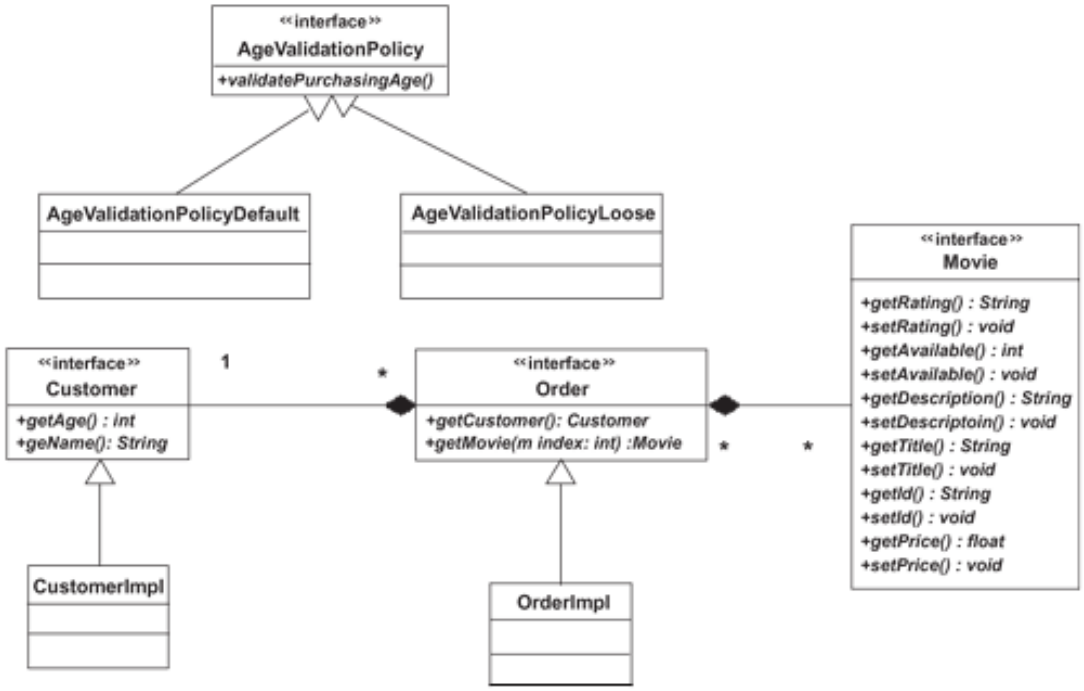
For example, say that an online movie store wants to enforce a policy to restrict the sale of movies to minors, based on movie ratings. The store executives haven't settled on a single policy, however, so our movie-ordering application has to be flexible enough to change policies with ease. Furthermore, movie ratings occasionally change, so the application must be able to take into account new ratings scales or criteria when an order is placed. All of this means that the rules for validating that a particular customer is old enough to order a particular movie could be considered *volatile* code. Rather than having to recompile new code every time the store policy or industry ratings system changes, it would be easier to isolate the volatile code and allow it to change dynamically. This is essentially what the Simple Policy pattern does.

Designing a Simple Policy

The `Order` class contains the volatile validation logic. This logic is extracted out into concrete classes that implement a single interface. In our case, the interface is named `AgeValidationPolicy` and there are two different policy implementations, as follows:

- `AgeValidationPolicyDefault` follows the U.S. movie rating recommendations (for example, a PG-13 movie cannot be sold to anyone under age 13).
- `AgeValidationPolicyLoose` says that anything up to and including R-rated movies can be purchased by individuals of any age.

In addition to the policy classes, we have an `Order` object that contains a customer and one or more movies, as shown below:



Not shown in the diagram is a simple registry, which implements the GOF's Singleton pattern (see Resources). To keep things simple, the registry is a concrete implementation of the Property Container pattern. When an order is created, the `Order` object locates the validation policy in the registry, then passes the customer information and movie order to the validation policy. The policy throws a run-time error if the customer is unable to purchase the movies due to a policy violation.

The run-time registry is convenient because it lets us change the validation policy on the fly. So, if the store executives knew that the store's sales were going to be audited by a movie watchdog group, they could run the default policy. Once the audit was complete, they could replace the default policy with the "loose" policy, thus increasing sales. Integrating the Simple Policy pattern into our application ensures that policy changes don't require recompilation, just that the new policy be registered in the run-time registry.

Implementing a Simple Policy

The interesting portion of the `Order` class is where the validation occurs. To demonstrate how the validation policy works, we'll look at the constructor on the concrete `Order` implementation:

```
/** Creates a new instance of OrderImpl */
public OrderImpl(Customer customer, Movie[] movies) {
    AgeValidationPolicy policy =
        (AgeValidationPolicy)
            PolicyRegistry.getInstance().getPropertyBy(
                AgeValidationPolicy.Token
            );
    if(policy==null) policy = new AgeValidationPolicyDefault();
    policy.validatePurchasingAge(customer, movies);
    this.customer = customer;
    this.movie = movies;
}
```

Upon attempting to create an instance of an order, the constructor retrieves the Singleton policy registry and gets the validation policy by a predefined token. If no validation policy is registered, the default policy is used. We then call the `validatePurchasingAge` method on the policy. If the policy fails, it returns a run-time exception and the constructor will also fail. Otherwise, we go ahead and set the customer and movie array.

Simple Policy demo

Once we're in the order constructor, we assume that someone is maintaining the registry with the age validation policy. The test program `com.stereobeacon.patterns.business.simplepolicy.TestSimplePolicy` demonstrates the run-time exchange of policies. To run the demo, we first add the default age validation to the registry. Once added, the default policy will be the one retrieved by the `AgeValidationPolicy.Token` (the token is simply the string "com.stereobeacon.patterns.business.simplepolicy.AgeValidationPolicy").

```
    public static void main(String[] args) {
        PolicyRegistry.getInstance().addPropertyBy(
            new AgeValidationPolicyDefault(), AgeValidationPolicy.Token
            );

        Movie[] movies = createMovies();
        Customer customer = new CustomerImpl();
        customer.setAge(17);
        customer.setName("Paul");

        try {
            Order o = new OrderImpl(customer, movies);
            System.out.println("Successful Order Creation");
        } catch(ValidationException exception){
            System.out.println("Could not create Order due to Validation Exception");
            System.out.println("Message is:  "+exception.getMessage());
        }

        PolicyRegistry.getInstance().addPropertyBy(
            new AgeValidationPolicyLoose(), AgeValidationPolicy.Token
            );

        try {
            Order o = new OrderImpl(customer, movies);
            System.out.println("Successful Order Creation");
        } catch(ValidationException exception){
            System.out.println("Could not create Order due to Validation Exception");
            System.out.println("Message is:  "+exception.getMessage());
        }

    }
```

Once the policy is added, we populate some movies and create a 17-year-old customer, Paul. Clearly, our customer should not be able to order R-rated movies. Upon creating the first order, we'll receive a validation exception. Next, we change the registry and use the loose validation policy, which allows anyone to order R-rated movies. The second order will successfully create, even with the same movies and customer. And with that, we've dynamically changed the validation policy at run time.

Notes about the Simple Policy pattern

Like the Property Container implementation, our implementation of the Simple Policy pattern is very simple, offering only a glimpse of what the pattern can do. Policies are very powerful when associated with containment hierarchies.

Take the example of a chain of movie rental stores owned by a parent company. The parent company sets a default sales policy for all branches, but each branch is also empowered to implement its own sales policy. Thus, a branch could set an individual policy that would be contained by the larger company policy at run time. The algorithm would then be set to check the store for a default movie sales policy; if one did not exist the parent company would be queried for the default policy,

otherwise the store's individual policy would be used.

Patterns summary

In this section we implemented two design patterns from *SanFrancisco Design Patterns*. These patterns were defined for a specific type of application, in this case business applications.

Integrating the Property Container into our application allows us to add attributes, such as release date, to our `Movie` class at run time, making the overall movie catalog easier to update. The Simple Policy implementation isolates validation logic, letting us change our sales policy at will. We'll use the code from both of these examples in the next section, as we continue building our Web-based movie-ordering application.

See Resources to learn more about these two design patterns and others to be found in *SanFrancisco Design Patterns*.

Section 4. A Web presentation pattern

A Web presentation pattern overview

In this section we're concerned with ensuring that our application is well-presented in terms of Web usability. To enhance the usability of our application, we'll implement the Repeated Menu design pattern.

At first glance, you might think a Web presentation design pattern is outside of the realm of Java programming. After all, what respectable Java programmer increases his HTML toolkit? But the Repeated Menu pattern is perfect for implementing with Java ServerPages (JSP) tag libraries, which will come in handy as we further develop our application in the next section. More and more it is becoming the responsibility of the Java programmer to implement complex and repeatable Web designs.

You'll find the Repeated Menu pattern documented on Martijn Welie's Web site. See Resources for a link to Martijn's site.

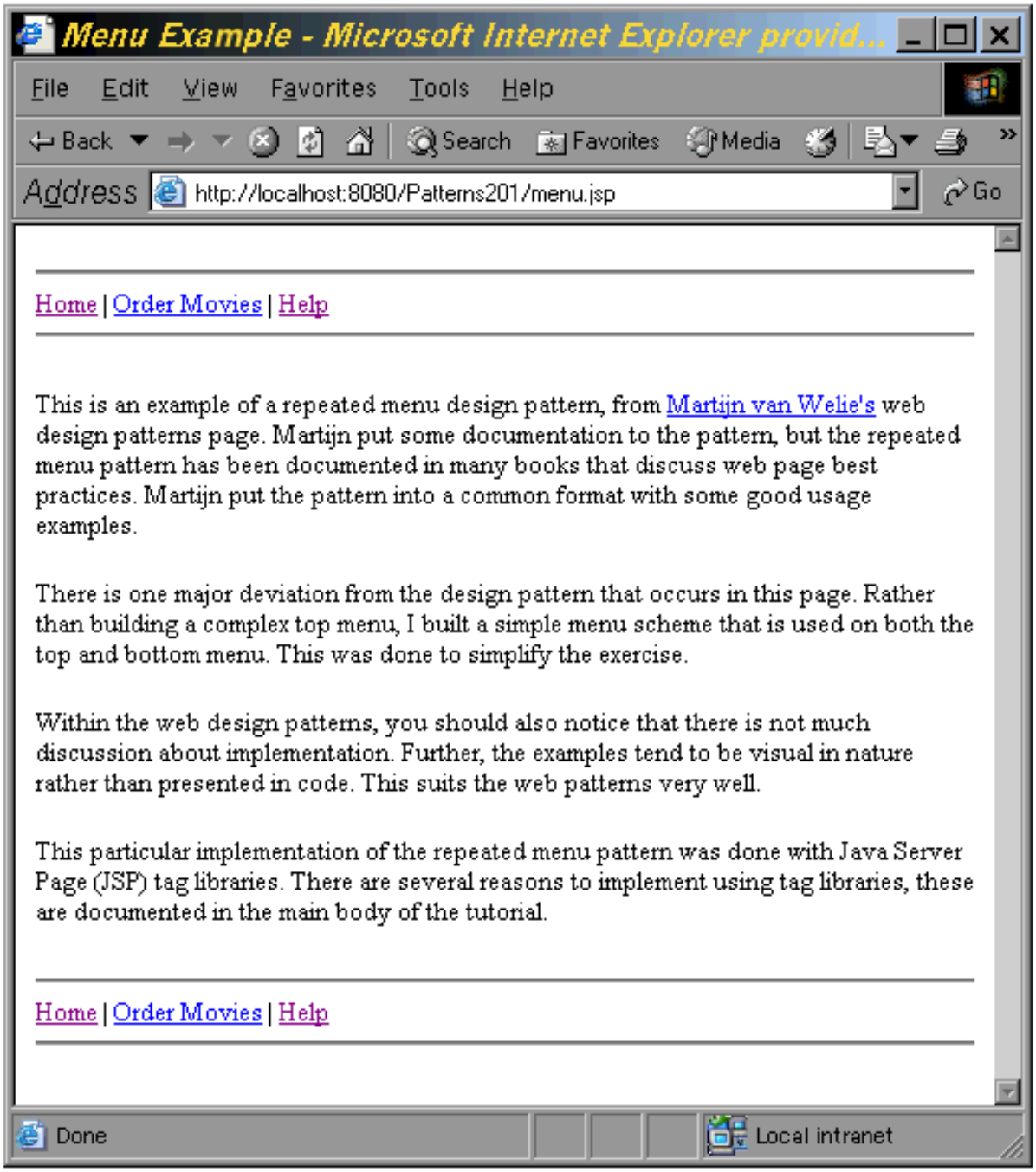
The Repeated Menu pattern

The Repeated Menu design pattern simply states that a menu should be repeated on a Web page if the contents of the page would otherwise become too long for a single screen. While this may seem to be a common-sense rule, many Web sites do not adhere to it (you will, of course, notice that this tutorial does an excellent job of using the Repeated Menu design pattern).

Web presentation design patterns are frequently coded in HTML, though patterns such as the Repeated Menu are particularly well suited for JSP tag libraries. By moving the menu implementation into a single tag, we can reuse the code as much as possible without cluttering up the HTML code or creating a maintenance nightmare.

A Repeated Menu example

The following simple image illustrates the nature of the repeated menu:



In the browser window you see that the menu for the Web site is repeated on both the top and the bottom of the page. This allows the user to easily access the menu based on where his eyes are on the page. Further, if the page were to scroll, only one menu would be visible at a time. This, again, helps the user navigate the Web site.

Coding the repeated menu

The repeated menu uses a custom-built JSP tag and is extremely simple to implement, especially for a Web page designer. Rather than coding HTML blocks for the top and bottom menu, all a Web page coder needs to do is include the menu tag library, then add tags where the menus need to be. To include the tag library, the following line is placed at the top of the page (this assumes you're using a .war file for distributing the tag library and that you have a correct descriptor file packaged with the file).

```
<%@ taglib uri="menutags" prefix="menu" %>
```

The body of the JSP tag looks like this:

```
<BODY>
<menu:topmenu/>
<p> content here...</p>
<menu:bottommenu/>
</BODY>
```

The menus can be repeated as often as you like.

Maintaining the repeated menu

The repeated menu is easy to maintain because the code is encapsulated in a single Java class, as shown below:

```
public class MenuPattern extends TagSupport {
      private static final String homeLink = "<a
href=\"menu.jsp\">Home</a>";
      private static final String movieLink = "<a
href=\"ordermovieplaceholder.jsp\">Order Movies</a>";
      private static final String helpLink = "<a
href=\"help.jsp\">Help</a>";

      public int doStartTag() throws JspException
      {
          try {
              pageContext.getOut().println("<hr>");
              pageContext.getOut().println(homeLink);
              pageContext.getOut().println(" | ");
              pageContext.getOut().println(movieLink);
              pageContext.getOut().println(" | ");
              pageContext.getOut().println(helpLink);
              pageContext.getOut().println("<hr>");
          } catch(IOException ioe) {
              throw new JspException(
```

```
                "Error: IOException while writing to client"
                + ioe.getMessage());
            }
            return SKIP_BODY;
        }

        public int doEndTag() throws JspException
        {
            return EVAL_PAGE;
        }

}
```

The above class extends the `TagSupport` class, which is a part of the `javax.servlet.jsp.tagext` package. Upon seeing the start of the tag, the JSP engine calls the `doStartTag` method. The method writes the menu to the output stream. To change a menu for all of the pages on a Web site (anywhere from 1 to 1,000,000), we need only to change this one piece of code.

JSP tag libraries are ideal for implementing many Web presentation design patterns and best practices. Tag libraries both make the Web page coder's job easier and greatly simplify Web page maintenance.

Section 5. A J2EE design pattern

A J2EE design pattern overview

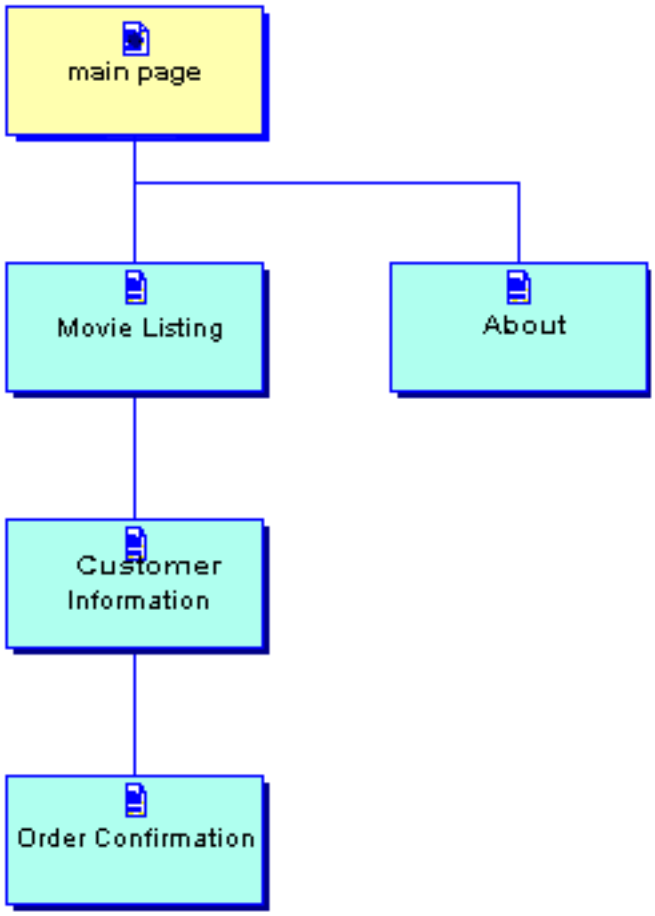
The Front Controller design pattern returns us to the realm of more traditional, back-end design patterns. Front Controller is based on the Model-View-Controller (MVC) architectural pattern, but is tailored for Web-based, enterprise-level applications.

Front Controller is one of the J2EE design patterns available from Sun Microsystems. Its primary function is to create a centrally controlled linking mechanism for managing page views. Integrating the Front Controller design pattern into our movie-ordering application gives us greater control over how users navigate through our Web application.

You'll find links in Resources for accessing information about the Front Controller design pattern in book form and on the Java BluePrints Web site.

Static versus dynamic navigation

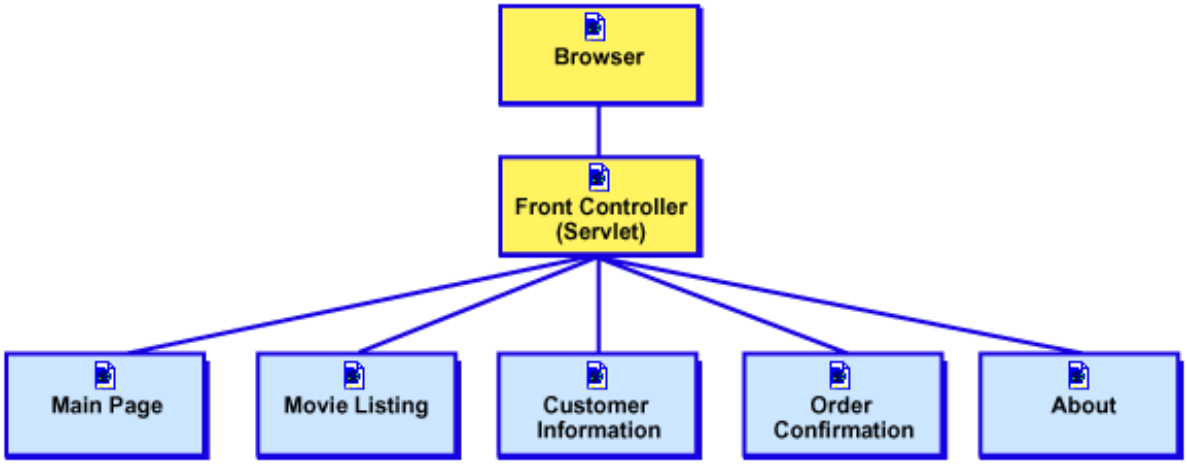
Traditionally, Web sites are statically linked. When viewing a Web page, links to related Web pages are embedded in the page itself, as shown in the following diagram:



This model is fine for Web sites that are structured like books, where the flow is linear and one-directional, and the page data does not change based on user input. But the book-based model is unrealistic for enterprise applications. In an enterprise application, page views are based more on the underlying state of the data on the page than on a static model. Given all the variables of an enterprise transaction, it is difficult to predict how each user will navigate the system. Enterprise applications generally need to employ more than one data model, so as to offer more than one navigational flow to users. By creating multiple navigational options and centralizing page management with a Front Controller mechanism, you can create a more dynamic, serviceable Web application.

Front Controller navigation

The Front Controller is a servlet whose task is to delegate business model information and view handling to subcomponents (that is, other servlets). The following diagram shows how the Front Controller manages page-view navigation:



In addition to allowing for a more sophisticated navigation system, the Front Controller provides a central place for such common operations as security checks, auditing, logging, or checking user session notifications received from an outside source.

Before we get into the details of the Front Controller mechanism, we'll look at the content and navigation setup for our movie-ordering application.

Page and navigation setup

Our movie-ordering application is fairly simple, since it contains only six pages. The first three pages listed below are accessible from the repeated menu. Regardless of how a user accesses a page, all page requests are routed through the Front Controller. The pages are as follows:

- The **Main Page** serves as our storefront.
- The **Movie Listing** page lists movies available from the movie database.
- The **About** page provides information about the application.
- The **Customer Information** page is presented if an order is sent but no customer information is available.

- The **Order Confirmation** page is presented if an order is successful.
- The **Order Rejection** page is presented if an order fails due to a validation error.

Placing an order

Here's an example of how a user might place an order through the movie-ordering application:

1. The user enters the Main Page.
2. The user chooses the "Order Movies" link and is presented with the Movie Listing page.
3. The user chooses one or more movies and places an order.
4. Because no customer information is available (that is, the user is unregistered and has no customer ID), the user is sent to the Customer Information page.
5. Once the customer ID has been created, the user's order is created and the validation logic is triggered. If the user is too young to order the movie, the Order Rejection page is presented. Otherwise, he receives the Order Confirmation page.

Now we can talk about what's happening on the back-end, with the Front Controller mechanism.

Document links

When using a Front Controller, links do not refer to a specific target, such as "home.jsp" or "movies.jsp". Rather, they're used to indicate an action that the user wants to perform, such as going to the main page, altering the movie order, or viewing the list of available movies. The Front Controller then makes any necessary adjustments to the data model and interprets the action into a concrete page view, which it returns to the user.

So, for example, the menu link

```
<a href="/Patterns201/frontcontroller/home">Home</a>
```

lets us know that the user wants to enter the main page. Based on the accumulated session data, we know that the user is already in the movie area of the application, which means he has already been through the main page once. So he must be returning to the main page to either restart his session or get more information about the application. At this point the Front Controller can easily present the user with a new page to help him with his current session, rather than simply returning him to the main page, where he will find only basic information that may not suit his particular need.

The Front Controller uses the `home` portion of the above link to determine the request, and the servlet engine uses the `/Patterns201/frontcontroller` portion of the link to route the request to the Front Controller servlet. The routing of the request based on `/Patterns201/frontcontroller` is a function of the application deployment descriptor used by the servlet engine.

The flow of operations

Upon receiving a request via a link or a Submit button, a typical Front Controller is responsible for the following:

- Performing centralized operations such as auditing, logging, and security checks
- Determining how the submitted page, including any form elements passed with the page, relate to the underlying data model
- Using a *flow manager* to determine what the next page view should be for the user; this is based on the current page combined with the state of the data model and any additional current page context that can be used
- Combining the new page view with the business data to construct a new page for the client

The Front Controller servlet

As previously mentioned, the Front Controller is a servlet whose task is to delegate business model information and view handling to other servlets. The code for the Front Controller servlet is as follows:

```
public class FrontController extends HttpServlet {
```

```
      public void init(ServletConfig config) throws ServletException {
          super.init(config);
      }

      public void destroy() {
      }

      protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
      throws ServletException, java.io.IOException {
          DataHandler.processRequest(request);
          ViewHandler.getNextView(request);
          HttpSession session = request.getSession();
          String newUrl = (String)session.getValue(WebKeys.CurrentPage);

          getServletConfig().getServletContext()
              .getRequestDispatcher(newUrl)
              .forward(request, response);
      }

      protected void doGet(HttpServletRequest request,
HttpServletResponse response)
      throws ServletException, java.io.IOException {
          processRequest(request, response);
      }

      protected void doPost(HttpServletRequest request,
HttpServletResponse response)
      throws ServletException, java.io.IOException {
          processRequest(request, response);
      }

      public String getServletInfo() {
          return "Short description";
      }

}
```

How the Front Controller works for our application

The Front Controller's main processing occurs in the `processRequest` method. Since our movie-ordering application requires no centralized operations, coding the Front Controller is very simple. Upon receiving an order request, we use a data handler to process changes to the data model, then we request a new view from the view handler.

To make things even more simple, this application does not require that we combine incoming data with the page view to construct a new page. This would be a more advanced, though not uncommon, implementation of a Front Controller mechanism. In our application, the view handler simply returns a JSP page built using scriptlets.

In the sections that follow we'll look at each of the three main components of the Front Controller setup for our application: the JSP implementation; the data handler implementation; and the view handler implementation.

The JSP implementation

The Front Controller mechanism is built around JSP technology. JSP pages use data held in the user's session to display data. The pages themselves would be more robust if we'd written them using JSP tag libraries and powered them with servlets. But building the pages out of scriptlets allows us to keep all our code in one place, so we've gone with that simpler solution.

```
<%@ taglib uri="menutags" prefix="menu" %>
<HTML>
<HEAD>
      <TITLE>Movie Order Page</TITLE>
</HEAD>
<BODY>
<menu:movietopmenu/>
<h1>Choose the movies you want to order</h1>
<%@ page
import="com.stereobeacon.patterns.business.propertycontainer.*" %>
<%@ page
import="com.stereobeacon.patterns.Web.frontcontroller.WebKeys" %>
<%@ page import="java.util.Hashtable" %>
<%@ page import="java.util.Enumeration" %>

<form method="post">
<input type="hidden" name="page" value="ordermovies">
<hr>
<%
      Hashtable movies = (Hashtable)session.getValue(WebKeys.Movies);
      Enumeration e = movies.elements();

      while(e.hasMoreElements()){
          Movie cur = (Movie)e.nextElement();
%>
<p>
<h2><% out.println(cur.getTitle()); %></h2>
<b>Description: </b><%
out.println(cur.getDescription()); %><br/>
<b>Price: </b><% out.println(cur.getPrice()); %><br/>
<b>Rating: </b><% out.println(cur.getRating()); %><br/>
</p>
<p>
Check to order:
<input type=CHECKBOX name=<% out.print(cur.getId()); %>
value="Checked" >

<%
      }
%>
</p>
<p>
<input type="submit" value="Submit">
</p>
</form>
</BODY>
</HTML>
```

Notes about the JSP pages

Our Front Controller implementation employs a typical scriptlet-based JSP page. The only unusual element is the `menu` tag at the top, which functions to incorporate repeated menus into our design. In the code above, the JSP page retrieves the list of movies stored in the session and iterates through them, placing the contents into a form. The user can then check the movies he wants to order and press the Submit button. The movies are populated by the Front Controller's data handler. Once the page is submitted, the view handler is responsible for determining the next page to display.

Of special note is the following line of code:

```
<input type="hidden" name="page" value="ordermovies">
```

The combination of the URL and the hidden field gives us the view context from which the page was submitted. With this information, we can manipulate the data model appropriately. In this case, once a user has entered into a relative URL ending in `/movies`, he is in a portion of the application where his next view will be based entirely on the underlying data model.

The data handler implementation

The data handler portion of the Front Controller is responsible for receiving the current request and working with the data model to bring the data to a new state. Typically, the data handler will do further delegation to fulfill user requests. We've kept things more simple for the sake of this example.

```
public static void processRequest(HttpServletRequest request){
    // retrieve the session, it is where we store our data model
    //  for use by other portions of the Web application
    HttpSession session = request.getSession();
    // retrieve the initialized flag from the session, it is set
    //  when the data is populated.
    Boolean initialized =
    (Boolean)session.getAttribute(WebKeys.Initialized);
    if(initialized==null || !initialized.booleanValue()){
        // the session has not been initialized with data so create
        // the movies and place them in the user session
        Hashtable movies = createMovies();
        session.setAttribute(WebKeys.Movies, movies);
        session.setAttribute(WebKeys.Initialized, new Boolean(true));
    }

    // retrieve the path info for the URL we are at
    String path = request.getPathInfo();
```

```
        // if the path isn't null and we are in the movie Web, then
        //  there may be changes to the data model (the other pages
        //  do not make changes to the data)
        if(path != null && path.endsWith("movies")){
            // retrieve the hidden field from the page, this will
            //  be null if it is the first time a movie page is requested.
            String page = request.getParameter("page");
            if(page==null || page.equals("ordermovies")){
                // if the hidden field is not available (implying it is
                //  our first time here, or we are on the ordermovies
                //  page), then go through the list of movies to determine
                //  if any of them were ordered by checking the checkbox.
                Enumeration e = request.getParameterNames();
                Vector ordered = new Vector(2);
                while(e.hasMoreElements()){
                    String parmName = (String)e.nextElement();
                    if(!parmName.equals("page")){
                        String parmValue = request.getParameter(parmName);
                        ordered.add(parmName);
                    }
                }
                // if any of the checkboxes were checked, set the ordered
                //  movies into the session for later use
                if(ordered.size()> 0){
                    session.setAttribute(WebKeys.MoviesOrdered, ordered);
                }
            } else if(page.equals("customerinfo")) {
                // if we were on the customer info page, get the
                //  data from it and check that it is not null.  Then
                //  put the customer data into the session for later
                //  retrieval.  if one of the fields is null, no
                //  customer is stored in the session.
                String name = request.getParameter("customername");
                String age = request.getParameter("age");
                if(name!=null && age!=null){
                    Customer customer = new CustomerImpl();
                    customer.setName(name);
                    customer.setAge(Integer.parseInt(age));
                    session.setAttribute(WebKeys.Customer, customer);
                }
            } else if(page.equals("failed") || page.equals("successful")){
                // if we were on a failure or successful order page, then
                //  check if the reset box was checked, if it was, then
                //  clear out the session data.
                if(request.getParameter("reset")!=null){
                    session.removeAttribute(WebKeys.MoviesOrdered);
                    session.removeAttribute(WebKeys.Customer);
                    session.removeAttribute(WebKeys.OrderInformation);
                    session.removeAttribute(WebKeys.OrderException);
                }
            }

            // now check if all of the necessary information exists
            //  to build an order.
            Vector v = (Vector)session.getAttribute(WebKeys.MoviesOrdered);
            Customer c = (Customer)session.getAttribute(WebKeys.Customer);
            if(v!=null && c!=null){
                Hashtable movies =
                (Hashtable)session.getAttribute(WebKeys.Movies);
                Movie[] m = new Movie[v.size()-1];
                for(int i=0 ; i<m.length ; i++){
                    String id = (String)v.elementAt(i);
                    m[i] = (Movie)movies.get(id);
                }
```

```
                try {
                    // create the new order.  Remember that the validation
                    //  logic will run when we create the order, thus possibly
                    //  sending a validation exception.
                    Order o = new OrderImpl(c, m);
                    // the order was successful, place the information in
                    //  the session for later use.
                    session.setAttribute(WebKeys.OrderInformation, o);
                } catch(ValidationException valid){
                    session.setAttribute(WebKeys.OrderException, valid);
                }
            }

        }
    }
```

Notes about the data handler

Here's how the above data handler implementation works:

1. If the data is not initialized, the data handler initializes it and places flags in the session to indicate the movies are initialized. The data handler also places the movies into the session to be retrieved by the JSP pages and the view handler.

2. If the user is in the movies portion of the application, the data handler checks to see if he is on the Order Movies page. If he is, the data handler determines whether any values (movies) have been selected. If so, it builds a vector of movies ordered and places the vector in the session for later processing.

3. If the user is on the Customer Information page, the data handler determines whether he has completely filled in the required information. If so, the data handler places a customer instance into the session for later use.

4. Once a Failed or Successful Customer Information page has been shown, the user has the option to check a box to reset the session. If this box is checked, the data handler clears all of the session data.

5. Once the user has created a customer ID and placed a movie order, the data handler creates a new movie order. If the new order successfully creates, the data handler places the order information in the session. If a validation exception is thrown, the data handler places a flag in the session.

Notice that all of these mechanisms are based entirely on the data model. When a user exits a page view, the next page view is undetermined. The data handler has only made the appropriate data model changes and any exceptions available to the session. The action of moving the user from one page to the next takes place elsewhere in the servlet (or in JSP pages).

The view handler implementation

The view handler must look at the current session context and combine it with the data that is available in the session to determine which page view should occur next. Upon determining the next view, the view handler places an attribute in the session, requesting the next page. The Front Controller servlet will then delegate this request to its request dispatcher.

Here's our view handler implementation.

```
public static void getNextView(HttpServletRequest request){
    // retrieve the session, this will be used to store and
    //  retrieve model and view state
    HttpSession session = request.getSession();

    // get the URL path
    String path = request.getPathInfo();
    if(path==null){
        // if the path is null, it is a URL: /Patterns201/frontcontroller
        //  so take the user to the entry page
        session.setAttribute(WebKeys.CurrentPage, "/movieentry.jsp");
    } else if(path.endsWith("movies")){
        // if the path is /Patterns201/frontcontroller/movies, then the
        //  user is in the movie portion of the Web where the current
        //  view is based entirely on the datamodel.

        // retrieve the list of ordered movies
        Vector moviesOrdered = (Vector)session.getAttribute(
            WebKeys.MoviesOrdered);
        if(moviesOrdered==null){
            // if there is not an order, then the user needs to order
            //  some movies before they go further
            session.setAttribute(WebKeys.CurrentPage, "/ordermovies.jsp");
        } else {
            // if there is an order, retrieve the customer information
            Object customer = session.getAttribute(WebKeys.Customer);
            if(customer==null){
                // if there is no customer information, prompt the
                //  user for customer information
                session.setAttribute(
                    WebKeys.CurrentPage, "/customerinformation.jsp");
            } else {
                // if there is customer information, check for either
                //  a valid order, or an exception
                Order o = (Order)session.getAttribute(
                    WebKeys.OrderInformation);
                ValidationException ve = (ValidationException)
                    session.getAttribute(WebKeys.OrderException);
                if(o!=null){
```

```
                        session.setAttribute(
                            WebKeys.CurrentPage, "/successfulorder.jsp");
                    } else if (ve!=null) {
                        session.setAttribute(
                            WebKeys.CurrentPage, "/failedorder.jsp");
                    }
                }
            }
        } else if(path.endsWith("about")){
            // if the about link was pressed, return the about page
            session.setAttribute(WebKeys.CurrentPage, "/about.jsp");
        } else {
            // by default, put up the entry page
            session.setAttribute(WebKeys.CurrentPage, "/movieentry.jsp");
        }
    }
```

Notes about the view handler

The view handler implementation first determines if the user is in the Main page, the movie-ordering portion of the application, or the About page. If the user is in the movie-ordering portion of the application, the view handler must determine the state of the data contained in the session to know what page it should display next. If no order has been placed, the view handler will display the Movie Order page. If the user has made a movie selection but provided no customer information, the view handler will display the Customer Information page. If the combination of customer information and movie selection creates a validation error, the view handler will display the error page.

With all its components in order, we're now ready to deploy our movie-ordering application.

Deploying the application

To facilitate easy deployment of the application, everything is bundled into a Web archive (.war) file. To install the application, take the .war file from the `/dist` directory of the downloaded source code and place it in your Tomcat installation `/webapps` directory. Upon starting Tomcat, you should be able to get to the application by typing the URL: `http://localhost:8080/Patterns201/frontcontroller`.

A J2EE design pattern summary

In this portion of the tutorial we've built and deployed a complete Web-based application for ordering movies. We used four design patterns to build the

application: Property Container, Simple Policy, Repeated Menu, and Front Controller. We also used a variety of technologies to deploy the application: HTML, JSP technology, servlets, and Java code.

As you can see, the logic of this type of application can quickly become complex, because we're essentially dealing with a state machine. Using design patterns lets us collapse this logic into simpler and more easily understood commands and strategies.

In the final section of the tutorial we'll reverse the way we use design patterns. Rather than learning patterns and employing them to build an application, we'll look at how knowledge of design patterns can help us understand and (hopefully) better employ an existing application.

Section 6. Using patterns to discover software

Using patterns to discover software overview

Many people fail to immediately recognize the role of software patterns in helping programmers decrypt a software solution. This is unfortunate because much of a developer, designer, or architect's day-to-day work revolves around understanding how existing software works. You should be familiar with at least one of the following scenarios:

- An *architect* looks at existing software to determine how it may influence his own architectures.
- A *designer* looks at legacy systems to determine what portions can be efficiently brought forward into a new architecture.
- A *programmer* looks at other people's software to locate bugs and maintain existing code.

Extensive knowledge of design patterns helps with all of these tasks, in addition to the task of building brand-new software and enhancing existing software with well-structured designs.

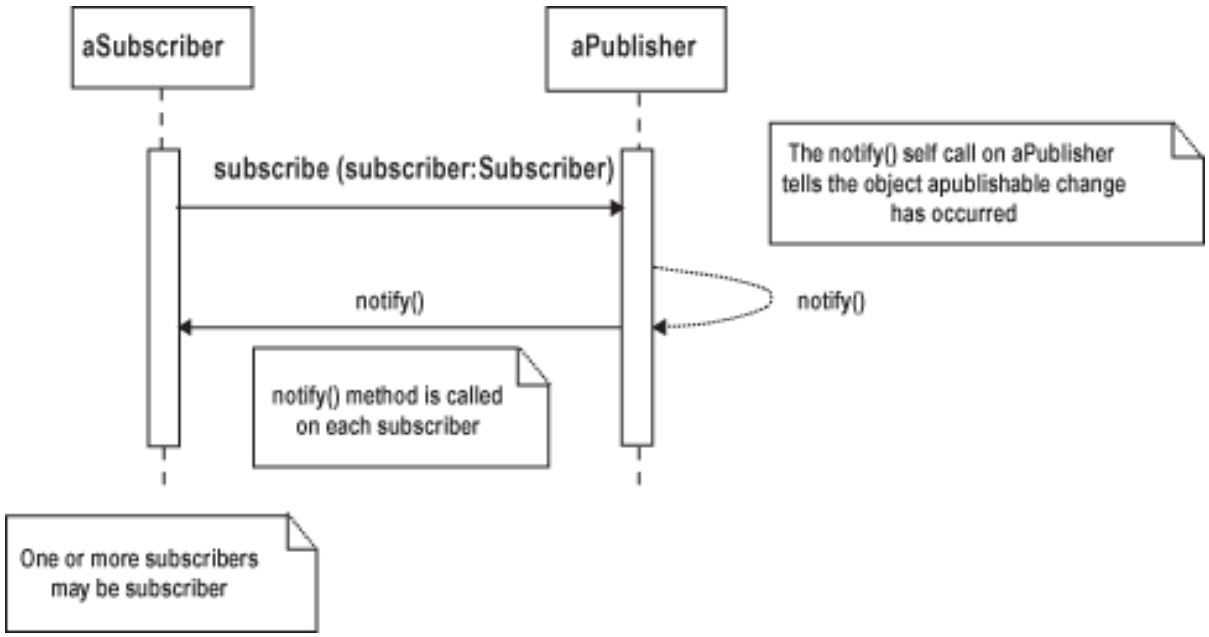
In this section of the tutorial, we'll learn about the general Publish/Subscribe (P/S) design pattern and the more specific Event Channel design pattern. We will then

apply what we know about these patterns to the Java Message Service (JMS). By the end of this section, you will have some sense of how JMS is similar to P/S and the Event Channel pattern, as well as where it differs.

Publish/Subscribe pattern

At the heart of most of today's event-based systems is the P/S mechanism. It is similar in nature to the GOF's Observer pattern. The problem that P/S addresses occurs when an object holds data on which other objects depend. For example, when I get in a bad mood, it would be nice if I could publish that information to those who might be impacted by my mood. My family would certainly subscribe to that data, and some might respond by staying away from me!

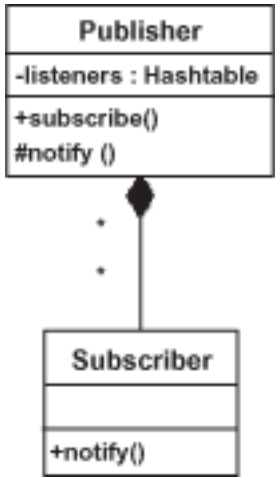
To implement P/S, an object (the publisher) has an interface that clients (subscribers) use to subscribe to changes. Upon subscribing, the publisher adds the subscriber to an internal list of clients who will be informed of data changes. When the object state changes, the publisher goes through the list of subscribers informing each one, as shown in the UML sequence diagram below:



Basic P/S implementation

A P/S system involves two classes, as shown in the following diagram. The `Subscriber` typically implements an interface with a `notify` (or similarly named) method on it. The `Publisher` typically has a public `Subscribe` (or similarly

named) method that takes a `Subscriber` as an argument.

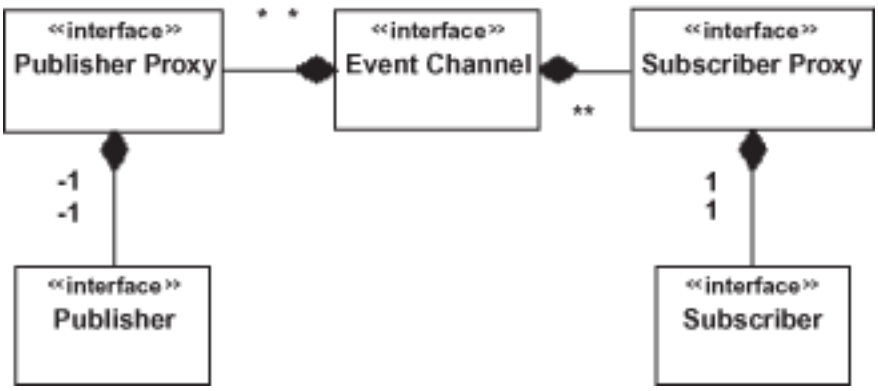


A variety of mechanisms are used within the `Publisher` to keep track of the `Subscriber` s. In the diagram above, a hashtable is used; other implementations use vectors or arrays. Typically, a P/S system also has some way to start the event-firing mechanism. In the `Publisher` shown above, a protected `notify` method tells the `Publisher` to fire an event to the `Subscribers`.

The Event Channel pattern

The Event Channel pattern from the Object Management Group (OMG) extends P/S to function more appropriately in a distributed environment. An *event channel* creates a centralized channel for events. This central channel allows objects to publish events or subscribe to events. The event channel mechanism greatly enhances the basic P/S pattern in that a subscriber can receive published events from more than one object, even though it is registered with only a single channel.

The Event Channel pattern uses proxy objects to subscribe to the event channel and a proxy object to publish events to the channel. The use of proxies allows the proxy to exist outside of the process boundary of the actual publisher or subscriber. A conceptual view of an event channel is shown below:

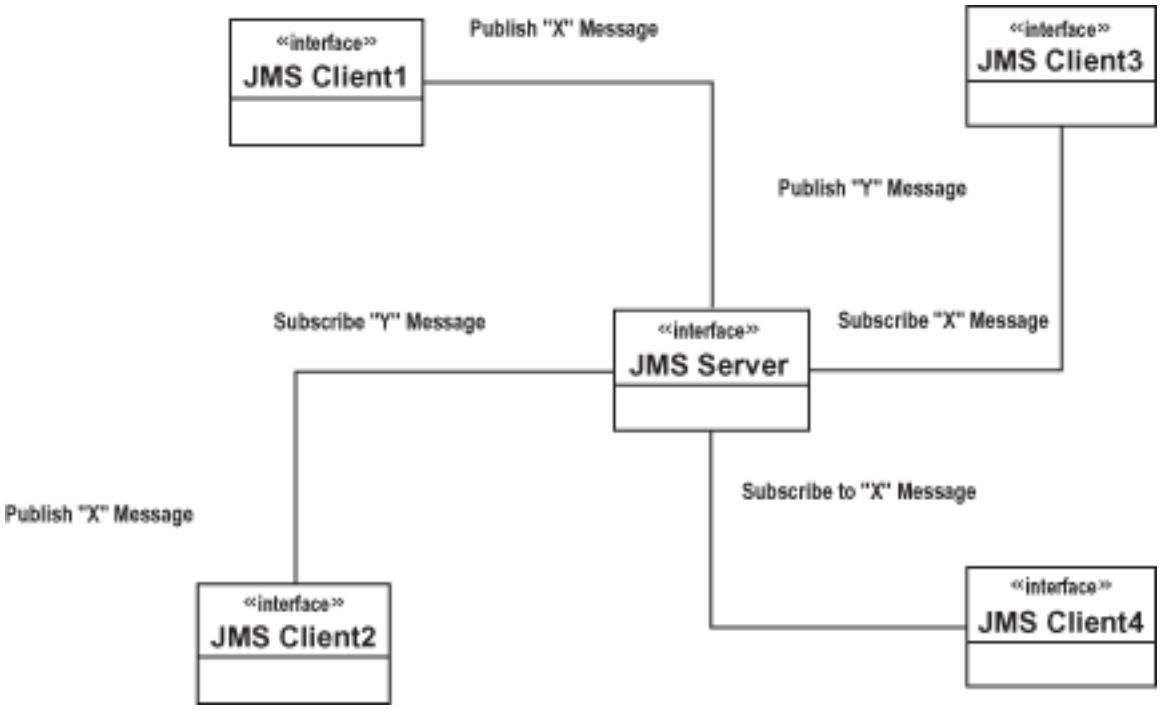


The Event Channel pattern employs *strongly typed events*; this means a subscriber can expect certain types of event data back when registered for a particular event. It also lets the subscriber both push and pull events, rather than only being able to have events pushed to it.

Understanding the Java Message Service

What we now know about the P/S and Event Channel patterns should help you quickly get your arms around the Java Message Service (JMS). The implementation details of the JMS differ from the above descriptions, but conceptually and by design JMS reflects the P/S and Event Channel design patterns.

Right off, you will the notice similarities between the conceptual documentation for JMS and the two design patterns. Most conceptual overviews of JMS include a diagram like the one that follows:



Clues in the code

The above overview gives you your first hint of the similarities between JMS and the P/S and Event Channel patterns. Browsing through the code for all three systems will further confirm your suspicions. Sessions with JMS subscribe to topics and publish to topics. Subscriptions and publications occur against queues and topics. For example, the following code is a subset of code that creates a connection to a topic and subsequently publishes a message to that topic:

```
TopicPublisher publisher = publishSession.createPublisher(topic);
TextMessage message = publishSession.createTextMessage();
message.setText("Hello");
publisher.publish(message);
```

Subscribers are similarly handled; they locate the JMS topic and subscribe to it, as shown below:

```
TopicSubscriber subscriber = subscribeSession.createSubscriber(topic);
subscriber.setMessageListener(this);
```

This code is greatly abbreviated and meant to illustrate a point: you don't have to know everything about JMS, or even compile any code, to understand that you're

working with the general P/S design pattern and that JMS more closely resembles the Event Channel design pattern.

Learning with design patterns

Even a brief understanding of the P/S and Event Channel patterns let us quickly grasp the underlying structure of JMS. Further exploration will yield further discovery. You will find that there are many differences, as well as likenesses between Event Channel and JMS implementations. For example, JMS includes point-to-point event delivery as well as the many-to-many delivery enabled by event channels. Conversely, the Event Channel pattern allows for strong typing whereas JMS does not.

These are fine points, but they add up to a deeper understanding of how JMS works. This understanding will become most beneficial when applied to your own software designs.

P/S and Event Channel design patterns summary

In this section of the tutorial, we learned the basics (the very basics) of the P/S and Event Channel design patterns. We then applied what we'd learned about these patterns to JMS. From a quick conceptual overview diagram and some code samples we could see that JMS was basically a P/S mechanism with many attributes similar to an event channel.

The important thing to take away from this section is this: the more design patterns you know and the deeper your understanding thereof, the stronger your grasp of software design will be. Understanding design patterns lets us see similarities and differences in software that we may never have seen before.

Section 7. Wrapup

Summary

Throughout this tutorial we have discussed design patterns that go beyond the basic GOF patterns. We first started by discussing resources you may not have previously

known about. These resources include approximately 866,000 Web sites and dozens of formal print publications.

Next, we implemented two patterns that can be used in business applications (Property Container and Simple Policy), as well as patterns that can be built with J2EE technologies (Repeated Menu and Front Controller). We then wove these four pattern implementations together into a single Web-based application.

Finally, we took a high-level view of the much-used P/S and Event Channel patterns. Using these patterns we quickly grasped the conceptual design of JMS.

The list of available patterns and pattern implementations is always growing. To be vital in software design, development, and architecture, you must continually increase your awareness of design patterns. Start by seeking out new resources, both on the Web and in book form. Then use the patterns you've found, both in your own software design and in your observation of existing software systems.

Advanced exercises

The following are interesting exercises to extend both our example Web application and your knowledge of the patterns presented in this tutorial:

1. Extend the Property Container pattern to use a run-time containment mechanism that will query "parents" (such as a parent company) for a property.
2. Develop additional age-validation policies, register them, and use them in the movie-ordering application.
3. Implement a version of the P/S pattern for the movie-ordering application.
4. Learn more about the Event Channel design pattern and implement a version of it in the movie-ordering application.

Downloads

Description	Name	Size	Download method
Source code	j-patterns201.zip	140KB	HTTP

Information about download methods

Resources

Learn

- Design pattern Web sites
  - Brian Foote's Web site (http://www.laputan.org/foote/papers.html) is a virtual treasure trove for the student of design patterns. (At the time we went to press, this Web site was unavailable. Welcome to the ever-changing world of the Web!)
  - Martijn van Welie's patterns site (http://www.welie.com) contains many patterns that are applicable to user-interface and Web-usability design.
  - You'll find the Simple Policy and Property Container patterns on IBM's Patterns for e-business site (http://www.ibm.com/developerworks/patterns/index.html).
  - The Front Controller pattern is one of several useful J2EE design patterns you'll find on Sun Microsystems's Java BluePrints site (http://java.sun.com/blueprints/index.html).
  - The Object Management Group (http://www.omg.org) is responsible for the Event Channel design pattern.
  - The Patterns for e-business Web site has been updated with new Access Integration designs (http://www.ibm.com/developerworks/patterns/access/index.html) that enable users to access multiple back-end systems through a single sign-on process, either from a browser or now from pervasive device clients. Access Integration also lets you personalize content based on user role, identity, and preferences. Learn more about these powerful designs.
- Articles and tutorials
  - The tutorial " Java design patterns 101 " (*developerWorks*, January 2002, http://www.ibm.com/developerworks/education/r-jpatt.html) is an introduction to design patterns. Find out why patterns are useful and important for object-oriented design and development, and how patterns are documented, categorized, and cataloged. The tutorial includes examples of important patterns and implementations.
  - Paul Monday's recent tutorial " Java event delivery techniques " (*developerWorks*, February 2002, http://www.ibm.com/developerworks/education/r-jdel.html) expands upon the brief discussion of JMS found here.

- Malcolm Davis's article " Struts, an open-source MVC implementation " (developerWorks, February 2001, http://www.ibm.com/developerworks/java/library/j-struts/) introduces the MVC architecture, on which the Front Controller design pattern is based. This article also deals extensively with JMS and Servlets technologies.
- Embrace the dark side of programming with design patterns! " A taste of Bitter Java " (developerWorks, March 2002, http://www.ibm.com/developerworks/java/library/j-bitterjava/) is a preview of Bruce Tate's upcoming book about antipatterns, which he describes as "common traps for developers with dire consequences".
- Patterns architects discuss emerging Web services technology (http://www.ibm.com/developerworks/patterns/guidelines/web-services.pdf) and its effect on Pattern solutions.

- Recommended books
  - Start with the book that launched all the rest: Design Patterns: Elements of Reusable Object-Oriented Software (Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, http://devworks.krcinfo.com/WebForms/ProductDetails.aspx?ProductID=0201633612).
  - One of the GOF, John Vlissides, also wrote the book Pattern Hatching (Design Patterns Applied) (Addison-Wesley, 1998, http://www.amazon.com/exec/obidos/ASIN/0201432935/qid=1015942491/sr=1-9/ref=sr_1_
  - The Pattern Almanac 2000 (Rising, Addison-Wesley, 2000, http://www.aw.com/catalog/academic/product/1,4096,0201615673,00.html#summary) is an essential resource for the student of design patterns.
  - Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects (Schmidt, Stal, Rohnert, Buschmann, Wiley, 2000, http://devworks.krcinfo.com/WebForms/ProductDetails.aspx?ProductID=0471606952) is one of those books that uses patterns to help you learn about a particular area of software development. It's also a fine addition to any patterns library.
  - Along the same lines (and first in the series) is the book Pattern-Oriented Software Architecture, Volume 1: A System of Patterns (Buschmann, Meunier, Rohnert, Sommerlad, Stal, Wiley, 1998, http://www.wiley.com/Corporate/Website/Objects/Products/0,,104675,00.html).
  - The upcoming Framework Process Patterns: Lessons Learned Developing Application Frameworks (Carey, Carlson, Addison-Wesley, 2002, http://www.amazon.com/exec/obidos/ASIN/0201731320/qid=1015908003/sr=1-2/ref=sr_1_ is for the advanced study of design patterns. It discusses older design

patterns, how they're derived, and newer patterns that apply specifically to building frameworks. One of the patterns, "Missed it by That Much," addresses the fine line between being able to apply an existing pattern and finding a new pattern.

- Also from James Carey and Brent Carlson is SanFrancisco Design Patterns: Blueprints for Business Software (Addison-Wesley, 2000, http://cseng.aw.com/book/0,3828,0201616440,00.html)
- Patterns for e-business: A Strategy for Reuse (Adams, Galambos, Koushik, Vasudeva, IBM Press, 2001, http://www.mcpressonline.com/ibmpress/5206.htm) is available in book form.
- And so is the book that spawned the Java BluePrints site, Designing Enterprise Applications with the Java 2 Platform (Enterprise Edition) (Kassem, Addison-Wesley, 2000, http://www.digitalguru.com/product_detail.asp?catalog%5Fname=Books&product%5Fid=02

- Additional resources
  - You'll find hundreds of articles about every aspect of Java programming in the IBM *developerWorks* Java technology zone (http://www.ibm.com/developerworks/java/).
  - See the developerWorks Java technology tutorials page (http://www-105.ibm.com/developerworks/education.nsf/dw/java-onlinecourse-bytitle?Open for a complete listing of more free Java tutorials from *developerWorks*.
  - See the Guide to developer kits from IBM (http://www.ibm.com/developerworks/library/i-tools.html) for a listing of the latest IBM developer toolsets.

Get products and technologies

- Downloads
  - The Java 2 platform, Standard Edition (http://java.sun.com/j2se/) is available from Sun Microsystems.
  - The Tomcat 4.0.3 Servlet Engine (http://jakarta.apache.org/tomcat/) contains all the J2EE functionality you need to complete this tutorial.
  - The Netbeans (http://www.netbeans.org) development environment was used to develop all the examples used here.

About the author

Paul Monday
Paul Monday is a contributing developerWorks author.