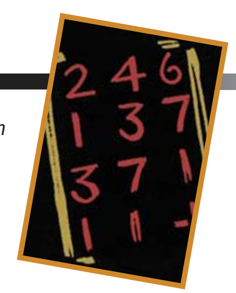


Editors: Paul F. Dubois, paul@pfdubois.com
George K. Thiruvathukal, gkt@cs.luc.edu
Konstantin Läufer, klaufer@luc.edu

A HIKE THROUGH POST-EJB J2EE WEB APPLICATION ARCHITECTURE

By Konstantin Läufer

SINCE THE MID-NINETIES, SERVER-BASED WEB APPLICATIONS HAVE EMERGED AS A CONVENIENT WAY TO PROVIDE FUNCTIONALITY TO A USER AUDIENCE WITHOUT ANY SPECIFIC software or system requirements except the need for a reasonably up-to-date Web browser. The tricky integration, installation, and configuration tasks are under the control of an expert on the server side. These advantages apply to both the general public and small research teams.

Typical Web applications have browser-based user interfaces for one or more user roles and might keep some information in persistent storage, such as a relational database. Since the early days of common gateway interface (CGI) scripting, technologies for developing Web applications have evolved by leaps and bounds to address growing expectations with respect to reliability, maintainability, extensibility, performance, scalability, and other goals.

In this article—the first in a planned series about Web application development—we take an exploratory hike through the architectural layers of a Web application built with state-of-the-art, widely used technologies. We focus on the upper layers that provide the application's user interface (we'll get into the lower tiers next time). By no means intended as a complete treatise on Web application development, this article is an overview meant to spark your interest and provide a starting point for further exploration.

Our running example—linear regression over a persistent set of points, implemented on the Java 2 Enterprise Edition (J2EE) platform—is simple enough to let us focus on the architecture, but rich enough to motivate a careful architectural discipline.

Architectural Goals and Overview

The typical Web application intended for production use has several specific goals:

- consistent visual styles, layout, and navigation throughout the application, each specified in a single place;
- support for internationalization;
- declarative validation of user input, specified in a single place;
- declarative component assembly, allowing the application to be reconfigured without modifying any code; and
- support for an object-oriented data model declaratively mapped to a relational database or other suitable persistent-storage technology.

In the old days, developers implemented the desired functionality as single or multiple CGI scripts without really worrying about these goals: the code responsible for such concerns was mixed together without much structural separation. Consequently, making any changes—for example, to the layout or the user-interface language—was extremely tedious and error-prone.

Today, professional Web developers stick to the same architectural disciplines and supporting technologies used by developers of other types of software. Various enterprise application development platforms have arisen, such as J2EE, .NET, and Zope; each has its own distinct characteristics, but all support similar architectural disciplines and blueprints.

The architectural blueprint shown in Figure 1 is typical: it has three major tiers (user interface, business functionality, and persistence) and multiple minor layers within each tier to support the specific Web application goals listed earlier. Underlying this blueprint is a *model-view-controller* architecture, which sets the tone for separating three key concerns: business functionality (the model), exchanging information with the user (the view), and dynamic behavior (the controller). For each tier, we can list the specific J2EE-related technology in our example to achieve the desired effect, without implying that any choice is superior to the alternatives. We use the term *post-EJB* (Enterprise Java Beans) for technologies that have recently emerged as more effective alternatives to J2EE-standard EJBs.

Core Functionality

Let's take a look at the application's core functionality. It must be capable of maintaining a set of points (including

 Copublished by the IEEE CS and the AIP 1521-9615/05/$20.00 © 2005 IEEE

adding, editing, and deleting them) and performing linear regression over the current set of points.

A good way to model this kind of basic "business" or domain functionality is in the form of a service interface with one or more service implementations. Figure 2 shows a service interface with methods that correspond to the expected capabilities, as well as some basic abstractions from our problem domain. The interface `RegressionResult` is an example of a *transfer object*, which aggregates multiple pieces of a result as a single object.

Assuming we have at least one implementation of our service interface—for example, a mock implementation for testing—our job is to expose the service's capabilities to the user. In other words, we've started in the middle of our overall architecture, so we need to build the user interface upward in successive layers.

The first step is to work on the dialog layer, which determines how the user will interact with the application. From there, we can focus on the presentation layer, internationalization, and consistent navigation, layout, and visual styles.

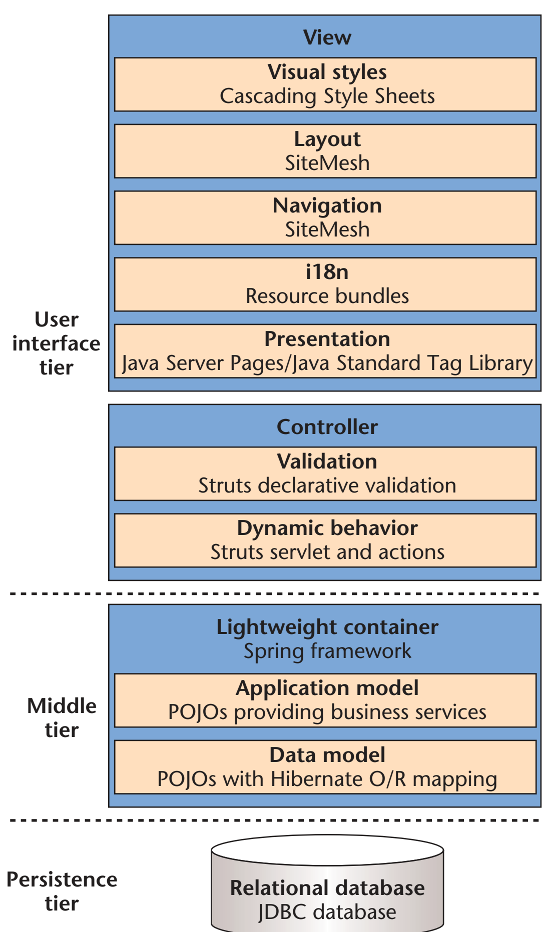


Figure 1. Architectural blueprint. The post-Enterprise Java Beans (EJB) Java 2 Enterprise Edition (J2EE) Web application architecture, with its typical tiers and layers, is a good starting point for addressing our architectural goals.

Dynamic Behavior

The dynamic model, shown as a UML state-machine diagram in Figure 3, helps us understand how the user interacts with the core functionality in an abstract fashion, without deciding (yet) what these interactions will look, sound, or feel like.

User interaction occurs only in one of the four *view states* (represented by rounded rectangles); because these states share three outgoing transitions, they're bundled in a composite state. Hollow circles represent *choice states*, which serve to branch on conditions, and solid circles represent *join states*, which allow multiple incoming transitions to continue along single paths.

Transitions are labeled *trigger [guard]/effect*. If the trigger—events such as a user's incoming HTTP request—is absent, the transition occurs automatically. If a guard is present, the transition occurs only if the guard is currently true. The effect is a piece of code executed as the transition occurs.

A particular scenario, such as the steps in Figure 4, shows a specific path through a dynamic model. In practice, it works well to start with a few scenarios, perhaps in the form of static (X)HTML pages, before trying to pin down the dynamic model.

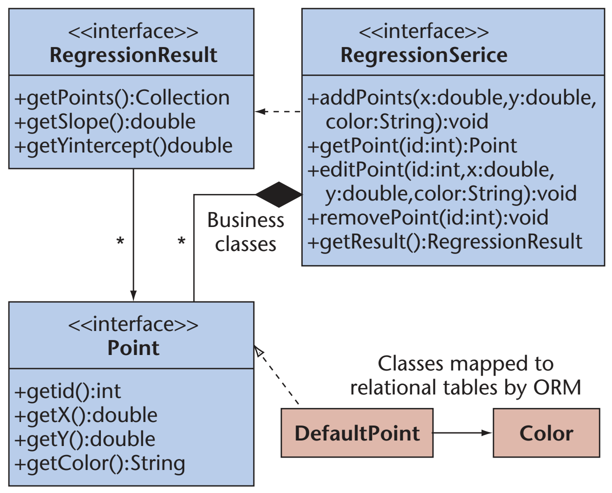


Figure 2. Service interface. The application and data models' class diagram shows the interfaces and classes that correspond to our Web application's core capabilities.

Web Application Frameworks

We still need to choose a suitable technology for the user interface tier of our Web application. The continual proliferation of Web application frameworks makes this choice increasingly difficult.

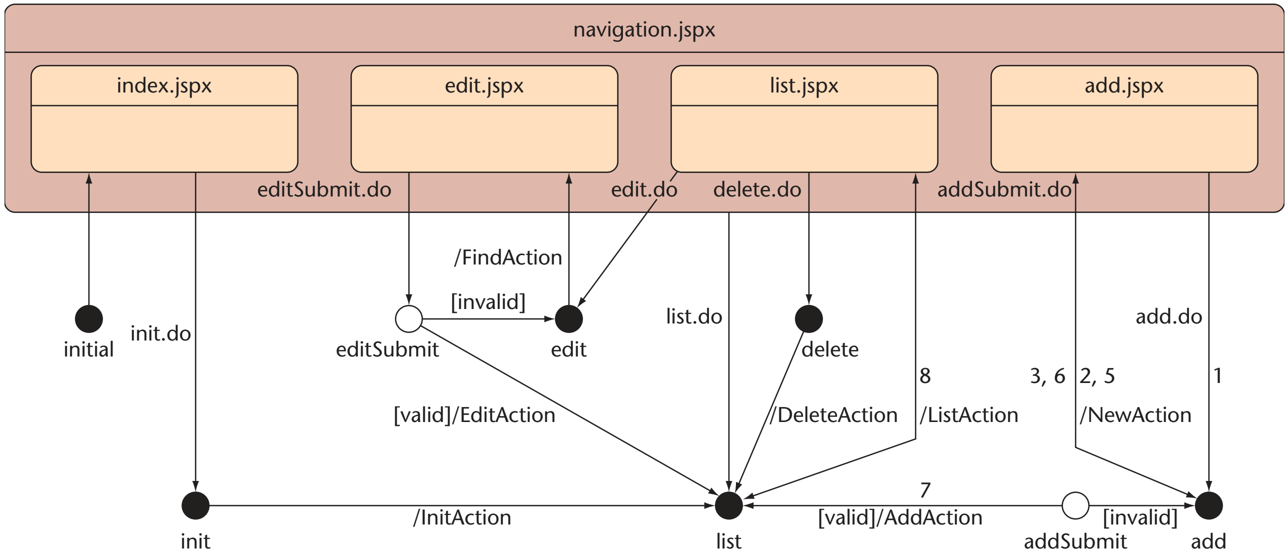


Figure 3. Dynamic behavior model. Hollow circles represent choice states, and solid circles represent join states. The rounded rectangles are the four view states.

In our example, we'll settle on the widely used, well-documented Jakarta Struts framework (http://struts.apache.org/), which supports a model-view-controller architecture. The controller is a configurable servlet, the model is any set of suitable Java classes, and the views are any suitable document-centric components, such as Java Server Pages (JSPs), that ultimately produce (X)HTML content a Web browser can display.

All incoming HTTP Web requests go straight to the controller, which assembles incoming Web form data into *form beans* and passes them to an *action* (a small application-specific controller plug-in). The action typically examines the incoming form bean, interacts with the model, and populates the form bean with information to send back to the user; the controller then forwards the request to another action or view for final presentation to the user. When mapping state-machine terminology to Struts components, "triggers" map to incoming requests, "effects" map to actions, "trigger arguments" map to form beans, and "view states" map to views. This correspondence isn't exact because Web applications' concurrency semantics differ from state-machine diagrams, but the structure is close enough to be quite helpful.

Controller Configuration

The Struts servlet configuration file, usually called `small struts-config.xml`, has several sections. Most importantly, the action entries correspond closely to the state-machine diagram and represent all transitions except those coming out of view states, which are implemented as links or form actions in the actual views. For example, the following entries represent the transitions involved in the scenario shown in Figure 4, which are numbered in Figure 3 in the order they occur in the scenario:

```
<action path="/add"                   <- trigger
  type="points.web.NewAction"         <- effect
  name="pointForm"                    <- argument
  scope="request"
  validate="false">
  <forward name="success"
    path="/WEB-INF/jsp/add.jspx"/> <- target
</action>

<action path="/addSubmit"
  type="points.web.AddAction"
  name="pointForm"
  validate="true"
  input="/add.do"
  scope="request">
  <forward name="success"
    path="/list.do"/>
</action>

<action path="/list"
  type="points.web.ListAction"
  name="listForm"
  scope="request">
  <forward name="success"
    path="/WEB-INF/jsp/list.jspx"/>
</action>
```

The following action from the middle of our scenario processes the incoming form data and passes it to the business service method it invokes:

```
public class AddAction ... {
```

```
public ActionForward execute(
  ActionMapping mapping,
  ActionForm pointForm,
  HttpServletRequest request,
  HttpServletResponse response) ... { ...
  // obtain arguments from form bean
  double x = Double.parseDouble(BeanUtils
    .getProperty(pointForm, PROPERTY_X));
  double y = Double.parseDouble(BeanUtils
    .getProperty(pointForm, PROPERTY_Y));
String color = BeanUtils
    .getProperty(pointForm, PROPERTY_COLOR);
// interact with model
Util.getRegressionService(this)
    .addPoint(x, y, color);
// provide informative message
request.setAttribute(
    ATTRIBUTE MESSAGE KEY, "add.message");
// trigger state transition
return
    mapping.findForward(FORWARD_SUCCESS);
```

Form beans are defined either programmatically as Java classes or declaratively in the Struts configuration file (the latter is much more convenient and usually the better choice). Note that the first four properties are for information coming from the user, whereas the last one is for sending a selection list of valid colors to the user:

```
<form-bean name="pointForm"
  type="org.apache.struts.validator.
    DynaValidatorForm">
  <form-property type="int"
    name="id" initial="-1"/>
  <form-property type="double"
    name="x" initial="0"/>
  <form-property type="double"
    name="y" initial="0"/>
  <form-property type="java.lang.String"
    name="color"/>
  <form-property type="java.util.ArrayList"
    name="colors"/>
</form-bean>
```

Although form beans are considered controller components, they're used to ship information back and forth between the views and the controller. We'll see in the next section how views refer to form beans.

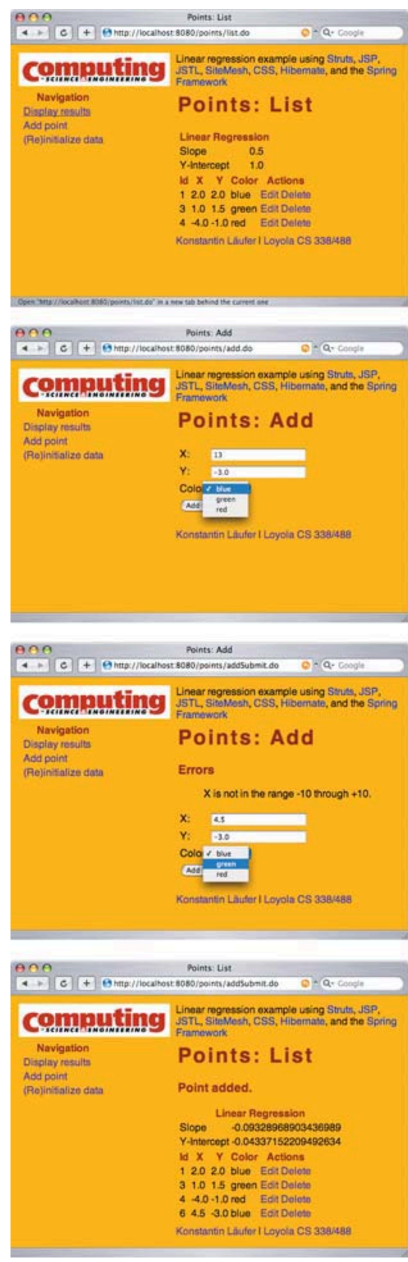


Figure 4. Screenshot sequence. Using *CiSE*'s Web page, we run through a typical scenario of a user's interaction with the application: attempting to add an (initially invalid) point and displaying the results.

Exchanging Information with the User

At this point, we're still missing the view components, which define how our application exchanges information with the user. We must choose a suitable visual presentation mechanism for our Web application (although, in principle, the exchange could also be aural, tactile, and so on). Struts will work with several view components, including Velocity (a Java-based template engine) and XML/XSTL, but it provides particularly good support for JSPs in the form of spe-

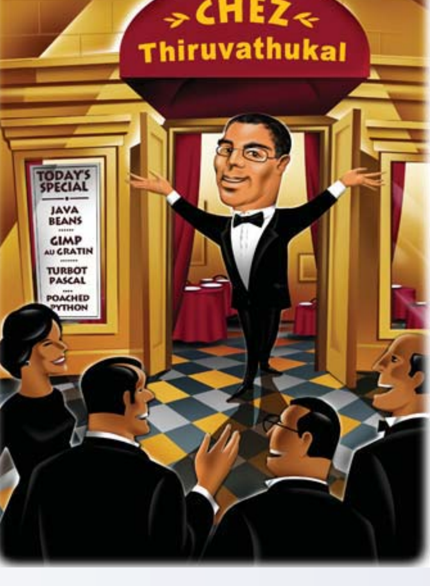
Chéz Thiruvathukal

Thunder and Lightning Using S5

At a recent Chicago Python User Group meeting (ChiPy), I noticed several folks using an innovative Web-based approach for so-called *lightning* presentations, wherein the speaker talks for five to 10 minutes about a technical topic in a nutshell. I'm under the distinct impression that a lightning talk, aside from being quick, is supposed to leave the listener in a "wow!" state at the end, wondering how such a great technical presentation can be given in 10 minutes or less.

S5 (www.meyerweb.com/eric/tools/s5/) is a standards-based presentation framework designed to work entirely within the comfort and safety of a Web browser. S5 itself is based on Opera Show, which was designed to work with the Opera Web browser (www.opera.com). Because many of us have fled to Mozilla Firefox in droves, Opera Show was a promising technology that went largely unnoticed in the browser world. The S5 authors started with Opera Show and extended it to include XHTML (the XML-enabled HTML) and Cascading Style Sheets (CSS), meaning that you can now customize the slide background, paragraph styles, and overall navigation using ordinary HTML and CSS markups.

Creating an S5 presentation is a straightforward process. You simply use a text editor or HTML editor (Mozilla Composer or Dreamweaver both work) to create your presentation. Then you link to the CSS for S5 and reference the CSS *slide* class on each section of your document intended to be a slide. S5 takes care of the rest, allowing you to switch between text and slideshow modes with straightforward JavaScript. (You need not know one iota about JavaScript to use S5 effectively.)



I think S5 and approaches based on it are likely to become compelling choices, especially for scientific, mathematical, and technical presentations. In my initial experiments, S5 worked great for text and code examples. It should also work with Math Markup Language (MathML) for rendering equations, which is claimed to be part of the next Mozilla Firefox release. To me, the best part is being ale to make effective presentations—minus the bloat—that work on all platforms in a compelling way. It seems like a clear win for everyone!

Soapbox: Open Source Not Supported?

I attended a meeting with some faculty colleagues recently, in which I overheard something interesting: "Free and open-source software is not supported." My first thought was to uninstall Linux from my computer. It's not supported, so I can't possibly rely on it for serious work. Who you gonna call? Linux-busters?

Needless to say, there continues to be a perception that

cialized tag libraries that coexist with the Java Standard Tag Library (JSTL).

Let's focus on a single set of concerns: what we have to present to the user and receive back. We're not worried (yet) about navigation, layout, or visual styles, which we'll get into later in a systematic way. At this stage, our view components all look very plain, as you can see in Figure 5 (Figure 6 shows this view's JSP source). Note how submission of the embedded form generates a request to a resource called `addSubmit`, hence the transition labeled "3, 6" in Figure 4.

By design, these plain views don't provide access to other functionality. The cool thing is that we can achieve the look from Figure 3 without having to touch these views again!

Internationalization

You probably noticed that the view source in Figure 6 includes absolutely no literal text at all. We facilitate internationalization (i18n, the process of adapting software for world-wide audiences) by using resource bundles, which contain literal, possibly parameterized text that view components can look up with a key. The resources used in the *add point* view are defined as

```
global.title=Points:
global.heading=Points:
# ...
add.title=Add
add.heading=Add
add.submit=Add Point
# ...
pointForm.prompt.x=X:
pointForm.prompt.y=Y:
pointForm.prompt.color=Color:
```

commercial support doesn't exist for Linux and other free or open-source software (FOSS) solutions. In recent years, however, a sea change has apparently gone unnoticed. Unlike the early FOSS years, in which there truly was limited commercial support, we simply can't say the same thing today. Companies such as IBM have stepped in with consulting services to provide support options for companies and organizations with limited technical expertise. Smaller players have a presence as well—LinuxCare, for example. Moreover, an entire community is willing to help, especially when you ask the right questions. I've found that's the quickest and least expensive option, even if it means I have to RTFM (Read the "Bloody" Manual, according to *The Hackers Dictionary*).

Later in that same meeting, I also learned that Linux's success (or lack thereof) is intrinsically connected to its presence on the desktop. Although I'm now using Linux almost exclusively and find it to be great, the user experience clearly isn't for everyone. In my view, the desktop isn't the central battleground. I don't see Windows going away anytime soon, nor do I think this would be a necessarily good thing.

As you already know from reading *CiSE*, Linux has a huge impact in the supercomputing cluster and center world, but this isn't an important arena for "regular" consumers (which is where the real money is). Nevertheless, the tide is slowly turning. Motorola announced recently that almost all of its mid-range phones (the ones most of us actually want) are going to run Linux. Most people who would not otherwise switch to Linux have to do it now to accomplish something besides desktop computing. Given the choice between "successful and relatively unknown" or "known not to be successful," I'll take the former.

Books as Living Works

Technical books, especially those covering open-source technologies, become obsolete quickly. I wish I weren't speaking from experience, but I've been bitten by this very phenomenon. The content in the two books I coauthored remains relevant, but it's no longer consistent with the software and frameworks covered within. I find myself wishing I could update the book—mostly for my own ego but also to help my readers, many of whom use it to try to understand the code available on my Web pages.

The *Plone Live* book project by Michel Pelletier and Munwar Shariff demonstrates a new approach that effectively combines print and Web media. (We covered the essence of Plone and content management systems in an earlier column, "Plone and Content Management," *Computing in Science & Engineering*, vol. 6, no. 4, pp. 88–95.) Pelletier and Shariff make regular content additions and incorporate corrections based on "bug reports," which identify something bad that happened to a good sentence or a code-related issue. At any given time, it appears that a reasonably current version of the book can be purchased in printed form—what a great idea!

Years ago, when I was working in the research center at a major Chicago-based printing company at the advent of the World Wide Web, the interest in on-demand printing was significant. It strikes me that the economics for publishers and authors would work out a lot better if you could purchase a book that is relevant, timely, and accurate, at any given time. Publishers could produce books in shorter-run formats, with subsequent runs serving as snapshots of the latest live book. Readers are more likely to purchase a printed book or, heck, even a subscription to a magazine, with the understanding that the work will be kept up to date with some frequency. Although the margins would certainly be lower in this new publishing world, I think this could be a win-win-win for authors, publishers, and readers alike.

To internationalize our application, we can simply create additional resource bundles with the same keys but translations of the original values to the target languages.

Validation of User Responses

Validation is a concern that arises in conjunction with user response but that is actually handled within the controller. As with several other things, we have a choice between *programmatic* and *declarative* along one dimension, and between *server-side* and *client-side* along another. By default, validation of user responses occurs on the server, but you can do it in the browser, using JavaScript, by inserting the element <html:javascript/> in your JSP view source. If you choose declarative validation in Struts, you can draw from several predefined validation "recipes." For example, we would implement the range check in our scenario as

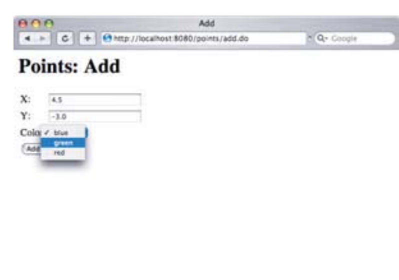


Figure 5. Screenshot. A bare-bones view component for adding a point as seen in the browser.

```
<field property="x"
  depends="required,double,doubleRange">
```

```
<jsp:root
xmlns:jsp="http://java.sun.com/JSP/Page"
   .../>
 <jsp:directive.page
    contentType="text/html"/>
 <html:xhtml/>
 <html:html locale="true">
   <head>
    <title><fmt:message key="add.title"/>
     </title>
    <html:base/>
  </head>
  <body>
   <h1>
    <fmt:message key="global.heading"/>
    <jsp:text> </jsp:text>
    <fmt:message key="add.heading"/>
   </h1>
   <html:errors/>
   <html:form action="/addSubmit">
    <table class="addedit">
     <tr>
      <td>
       <fmt:message
         key="pointForm.prompt.x"/>
      </td>
      <td><html:text property="x"/></td>
     </tr>
     <!— similarly for property y —>
     <tr>
      <td>
       <fmt:message
         key="pointForm.prompt.color"/>
      </td>
      <td>
       <html:select property="color">
        <html:options property="colors"/>
       </html:select>
      </td>
     </tr>
     <tr>
      <td>
       <html:submit>
        <fmt:message key="add.submit"/>
       </html:submit>
      </td>
     </tr>
    </table>
   </html:form>
  </body>
 </html:html>
</jsp:root>
```

Figure 6. Code from Figure 5. The Java Server Page source of a bare-bones view component defines the (X)HTML form for adding a point.

```
  <arg0 key="pointForm.fieldName.x"/>
  <arg1 name="doubleRange" key="${var:min}"
    resource="false"/>
  <arg2 name="doubleRange" key="${var:max}"
    resource="false"/>
  <var>
    <var-name>min</var-name>
    <var-value>-10</var-value>
  </var>
  <var>
    <var-name>max</var-name>
    <var-value>+10</var-value>
  </var>
</field>
```

This application-specific validation descriptor ensures that the coordinates of a point are doubles that fall within the range –10 … +10.

Finishing up the Interface

To finish up our user interface, we need to add consistent navigation, layout, and visual styles. A Web page layout and decoration framework such as SiteMesh (www.opensymphony.com/sitemesh/) can help. SiteMesh lets you specify decorators declaratively without having to touch the original views at all. It just needs a servlet filter entry in the Web application deployment descriptor (`web.xml`), which enables it to apply SiteMesh decorators in response to certain sets of requests.

Consistent Navigation

For consistent navigation, we can simply add the following navigation snippet to our overall layout decorator (shown in the next subsection):

```
<table>
  <tr>
    <th>
      <fmt:message key="navigation.
         heading"/>
    </th>
  </tr>
  <tr>
    <td>
      <html:link action="/list">
```

```
<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
   .../>
  <jsp:directive.page
    contentType="text/html"/>
  <html>
   <head>
    <title><decorator:title/></title>
    <decorator:head/>
   </head>
   <body>
    <table>
     <tr>
      <td class="left">
       <table>
        <tr>
         <td class="logo">
          <c:url value="/images/logo.gif"
            var="logo"/>
         <a href="http://cise.aip.org/cise/">
           <img class="logo" src="${logo}"/>
         </a>
        </td>
       </tr>
      <tr>
       <td class="navmenu">
       <!— navigation gets inserted here —>
       <page:applyDecorator name="panel"
        page="/WEB-
          INF/decorators/navmenu.jspx"/>
         </td>
        </tr>
       </table>
      </td>
     <td class="right">
      <table>
       <tr>
        <td class="header">
         <fmt:message key="header.text"/>
        </td>
       </tr>
      <tr>
       <td class="content">
        <!— original view gets inserted here —>
        <decorator:body/>
       </td>
      </tr>
     <tr>
      <td class="footer">
       <fmt:message key="footer.text"/>
      </td>
    </tr>
   </table>
</td>
</tr>
</table>
</body>
</html>
</jsp:root>
```

Figure 7. Layout decorator. The source of this decorator contains the (X)HTML elements for the overall layout and navigation placed on top of the plain views.

```
        <fmt:message key="navigation.link.
           list"/>
      </html:link>
    </td>
  </tr>
  <!— ...other navigation links... —>
</table>
```

Now we can easily make global changes to the navigation by modifying this section of the layout decorator.

Consistent Layout

The layout decorator puts our original views in the middle of a table that adds a logo and lines up the various view panels. Once again, we aren't touching the original plain JSP/JSTL views at all. Figure 7 shows the layout decorator's source.

Consistent Visual Styles

To finish up the user interface, we still need to add some visual styles, but we want them to be consistent across our application. This is where Cascading Style Sheets (CSS) really help. This fragment of our sample CSS applies style attributes to certain (X)HTML elements:

```
BODY {
  background: orange;
  font-family: sans-serif;
  color: black;
}
H1, H2, H3, H4, H5, H6 { color: maroon;
}
```

The remaining challenge is to associate the style sheet with our views without touching them. We achieve this by adding a top-most SiteMesh decorator that simply adds a link to the CSS to every view:

```
<html>
```

Further Reading

All developers have personal favorite development environments, ranging from vi or Emacs to Eclipse or IntellijIDEA. I've had good experiences with a combination of tools:

- Eclipse (www.eclipse.org)
- Sysdeo Tomcat plug-in (www.sysdeo.com/eclipse/tomcatPlugin.html)
- XMLBuddy (www.xmlbuddy.com)
- CSS Editor plug-in (http://csseditor.sourceforge.net)
- Spring IDE (www.springide.org/project/)
- Tomcat (http://jakarta.apache.org/tomcat/)
- Concurrent Versions System (CVS; www.cvshome.org)

Interested in some of the technologies described in this article? Check out these links:

- Struts (http://struts.apache.org)
- Java Server Pages (http://java.sun.com/products/jsp/)
- Java Standard Tag Library (http://java.sun.com/products/jsp/jstl/)
- SiteMesh (www.opensymphony.com/sitemesh)
- Cascading Style Sheets (www.w3.org/Style/CSS/)
- Spring (www.springframework.org)
- Hibernate (www.hibernate.org)

I've also found the following references to be extremely useful:

- R. Johnson and J. Höller, *Expert One-on-One J2EE Development without EJB*, Wrox, 2004
- Core J2EE Patterns (http://java.sun.com/blueprints/corej2eepatterns)
- AppFuse (http://appfuse.dev.java.net), a more industrial-strength Web application blueprint

Finally, for your enjoyment, you can use anonymous CVS to check out the complete code of the linear regression example (module LinearRegression) from :pserver:anonymous@cvs.cs.luc.edu:/root/laufer/cise.

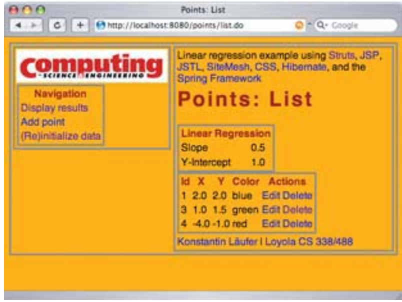


Figure 8. Visible table structure of a decorated view. The decorators add a logo and navigation on the left, and a header and footer on the right (in the middle of the right table column).

```
<head>
  <title>
    <fmt:message key="global.title"/>
    <jsp:text> </jsp:text>
    <decorator:title/>
  </title>
  <decorator:head/>
    <link rel="stylesheet"
      href="${pageContext
      .request.contextPath}/cise.css"/>
</head>
<body>
    <page:applyDecorator name="layout">
      <decorator:body/>
    </page:applyDecorator>
  </body>
</html>
```

Figure 8 shows the final result, with the visible table structure resulting from the layout decorator.

We started in the middle of our architecture and made it all the way to the top by using a combination of technologies to separate and confine our various issues into small, manageable, clean architectural layers. Such a separation of concerns facilitates modular development and helps produce more maintainable software.

Next time, we'll head for the bottom, looking at how object-relational mappings and lightweight containers can ease the burden of storing our data persistently.

Acknowledgments

I thank Igor Stoyanov, now at ThoughtWorks, for outstanding contributions as a student, challenging discussions, and keeping me informed on emerging technologies.

Konstantin Läufer is a professor of computer science and an associate dean of the graduate school at Loyola University Chicago. His research interests include programming languages, software architecture and frameworks, and educational technology. Läufer has a PhD in computer science from the Courant Institute at New York University. Contact him through www.cs.luc.edu/users/laufer.