11

Struts Development Using MyEclipse

IN THIS CHAPTER

- J2EE, Struts, and Eclipse 243
- Implementing a Web Application Using a Struts Module 245

"A framework is the final reinvention of a wheel."

—Anonymous

J2EE, Struts, and Eclipse

Chapter 9, "J2EE and the MyEclipse Plug-In," discussed many of the wizards and editors available within the MyEclipse plug-in, which is itself an aggregation of plug-ins to make J2EE development more convenient.

J2EE is a collection of distinct technologies. Some of the technologies have a direct application with system integration (CORBA, RMI), data connectivity (JCA), security (JAAS), XML (JAXP), and email (JavaMail), just to name a few. However, of all the technologies associated with J2EE servlets, JavaServer Pages and Enterprise JavaBeans appear to be the most popular. Due to the popularity of Web applications, servlets and JSPs have been used in many applications. This accumulation of experience has led to a recognition of the commonality of designs used for the presentation layer—which leads us to Struts.

Struts

Struts is not an official part of the J2EE architecture, but it rests on J2EE technologies—in particular, servlets and JSPs—to accomplish its goals. It codifies standard Web development practices and, once you understand how the pieces fit and why they fit the way they do, makes development of the presentation side of a Web application

simple. Struts is an open-source framework that defines a presentation architecture based on the Model-View-Controller design. The View component of the framework can be JSPs, Velocity, and/or XSLT, to name a few. The Controller component is a Struts-provided servlet that keeps a list of the objects you define, which drives the custom logic of your Web application. The Model component is your back-end system. It could be a custom datasource or a standard Java technology such as JDBC. Struts doesn't worry much about the model because that is the part of any system that is the most difficult to standardize.

Struts, as one of the more popular Web application frameworks, has the following moving parts:

- The `Action` class is the parent of a custom class that is responsible for handling incoming requests from the Web. This class is fine-grained in that it should handle one request. You are responsible for defining this class.
- An `ActionForm` is the parent of a custom class that maps directly to a form on some HTML page. The `ActionForm` could also be a superset of the fields, in case you have a form that spans multiple pages. You are responsible for defining this class.
- Zero or more HTML pages that send requests and zero or more JSPs (or other `Action objects`) that send responses back to the client. You are responsible for defining these pages and/or `Action objects`.
- The `ActionServlet` is the main switching point for the Struts framework. This is the only servlet needed for a Web application. The `ActionServlet` is responsible for receiving requests from the Web, determining which `Action` is responsible for a particular request, and routing the request to it. The configuration of the `ActionServlet` is defined in the Web application's `web.xml` file.
- The `struts-config.xml` file defines the configuration of a Struts-based application. It ties together the various pieces implemented by a developer. The `ActionServlet` uses this file to determine to which `Action` it should route incoming requests and with which `ActionForm objects`.

Also, support for dynamic forms (DynaActionForm), reusable presentation components (Tiles), and an external validator framework (the Validator framework) extends the capabilities of Struts even further.

For those of you already using frameworks such as Struts or WebWorks, you know using a proven design to implement your applications provides tremendous time savings. Many IDEs support Struts, and MyEclipse brings Struts support to Eclipse through the integration and extension of the Easy Struts plug-in, which was one of the early Eclipse plug-ins to support Struts.

Before installing MyEclipse, make sure you uninstall any existing Struts plug-ins. There may be conflicting configuration issues that are best avoided by using one plug-in task type at a time.

Finding and Installing MyEclipse and Tomcat

The "Downloading and Installing MyEclipse" section of Chapter 9 discusses the whys and wherefores of MyEclipse and walks you through the process of downloading and installing the plug-in, as well as downloading and installing Tomcat. If you have not installed either MyEclipse or Tomcat and need some help, refer to that section.

To use Tomcat within MyEclipse, you need to make sure MyEclipse is configured to support your version of Tomcat. If you have not yet configured MyEclipse to support Tomcat, again, refer to "Downloading and Installing MyEclipse" in Chapter 9 for the MyEclipse configuration information Tomcat needs to supply.

You are now ready to begin using MyEclipse to develop Struts applications.

Implementing a Web Application Using a Struts Module

Let's create a Web application to allow for the lookup of course information from a training company. An input page will ask for a course number, a course name, and an instructor name. If bad information is entered, the input page will let the user know which fields have a problem. If a course is found, the application will display information about it; otherwise, a "No course found" message will be displayed. You will put the main pieces together using the MyEclipse Struts wizards as well as the MyEclipse JSP editor.

The steps for creating a Struts application within MyEclipse are as follows:

1. Create a J2EE Web Module project.
2. Flag the Web Module project as Struts enabled.
3. Create the following three items in no particular order based on appropriateness:
   - Create an `ActionForm` that maps to the form in your starting HTML page. This step may be optional if the page you are coming from does not have a form.
   - Create an `Action`. If the `Action` processes an incoming form, use the `ActionForm` passed in as a method parameter.
   - Create a JSP that the `Action` can forward to, if necessary. An `Action` can execute and output HTML just like any other servlet, only it is not a servlet.
4. Update the `struts-config.xml` file to tie together the `ActionForm`, the `Action`, and the JSP (or any other target to which the `Action` might forward the request).
5. Deploy the Struts-based application to a Web server.

It is possible to create a Struts-based application with a JSP that gets routed back to itself, but it is not the intention of this chapter to go over Struts tips and tricks. Plenty of great Struts books are available that cover that kind of information. A few are listed in Appendix E, "Recommended Resources."

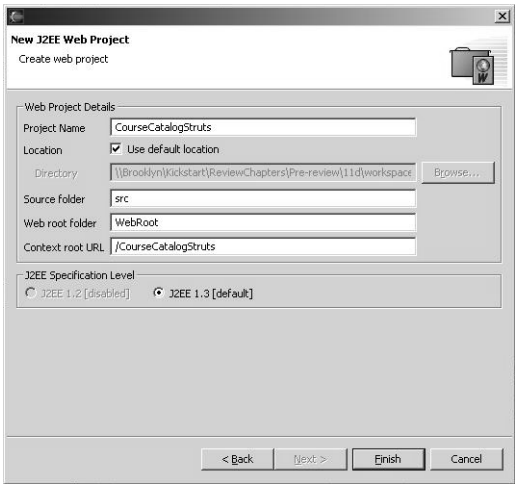

FIGURE 11.1 The New J2EE Web Project page, where you define the project name and the URL target for the project.

You start the process by creating a Web Module project and add Struts capabilities to it. Press Ctrl+N to open the New dialog; select Web Module Project beneath J2EE and click Next. Enter the project name CourseCatalogStruts (see Figure 11.1) and click Finish. If the Confirm Perspective Switch dialog opens asking for permission to switch to the MyEclipse perspective, click Yes. The project is once again set up the way a Web application should be: Under `WebRoot` you have a `WEB-INF` directory, a `lib` subdirectory, and a `web.xml` file to hold the configuration information needed by the Web server to properly deploy the application. There is also a `META-INF` directory, but it is really there for the use of the archiver, when you create and extract a WAR file.

The creation of a Web project is not sufficient to flag this project as a Struts-based project. Right-click the project and go to MyEclipse, Add Struts Capabilities. The New dialog opens on the Struts Support for MyEclipse Web Project page (see Figure 11.2). This page defines the information that will be inserted into `web.xml`. Struts Config Path is set to `/WEB-INF/struts-config.xml`. This path defines where the `ActionServlet` can find the Struts application configuration file. You can set this path to almost anywhere as long as a file exists at that location. In most circumstances, you will leave this value set to its default.

Leave the ActionServlet Name field set to `action`. Change the Base Package for New Classes field to `eclipse.kickstart.struts` and click Finish.

The Struts Wizard took care of the following tasks:

- The required Struts JAR files have been copied to `WEB-INF/lib`.
- The optional Tag Library Descriptor (TLD) files have been copied to `WEB-INF`. These files support the use of Struts custom JSP tags. In the development of a real application, you should create another directory under `WEB-INF` (for example, `tld`) where the TLD files should be moved. Web applications have a habit of aggregating support files, so the longer you can stay organized, the better. For this example, leave the files where the wizard placed them.

- The `struts-config.xml` file has been created with empty elements for the available tags that can contain information for use by the `ActionServlet`. The only missing tag is `plug-in`, but the MyEclipse Struts Editor will perform Code Assist if you type `<p` in the editor source page.
- The `web.xml` file, shown in Listing 11.1, has been updated with the `servlet` information needed by the Web server to load the `ActionServlet` when the Web server starts, and with default `init-params` for the `ActionServlet` to initialize itself. Also, a `servlet-mapping` is defined to route incoming resource requests that match the `*.do` pattern.

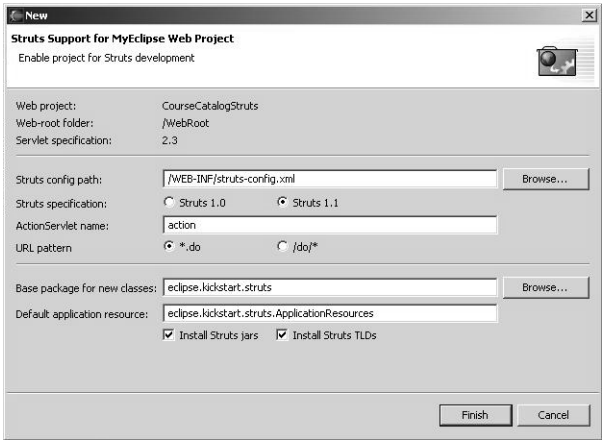

FIGURE 11.2 The New dialog's page for adding Struts support to an existing Web Module.

LISTING 11.1 `web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
➥"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <servlet>
        <servlet-name>action</servlet-name>
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
        <init-param>
            <param-name>config</param-name>
            <param-value>/WEB-INF/struts-config.xml</param-value>
        </init-param>
        <init-param>
            <param-name>debug</param-name>
            <param-value>3</param-value>
```

LISTING 11.1 Continued

```
        </init-param>
        <init-param>
            <param-name>detail</param-name>
            <param-value>3</param-value>
        </init-param>
        <load-on-startup>0</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
</web-app>
```

Now that the Struts project is created and properly initialized to support your development, you need to create an HTML page to accept input from the user, an `Action` class to manipulate the input and forward it to an output page, and an output HTML page to accept the data created by the `Action` class. Once again, MyEclipse comes to your aid with wizards to take care of creating stubbed versions of the files you need.

FIGURE 11.3
The `struts-config.xml` Outline view.

To open the Struts Wizard from the Package Explorer, open CourseCatalogStruts, WebRoot, WEB-INF and then double-click `struts-config.xml`. When the Struts editor opens, the Outline View displays a tree view of the contents of the file (see Figure 11.3). The Outline view is very important in the updating of the `struts-config.xml` file. Right-click `action-mappings` (the Struts element that ties together an input page, an action, and an output page) and select New Form, Action and JSP. This opens the New dialog at the Struts Form page (see Figure 11.4).

As a presentation-level piece, it is best to think of the functionality you are about to put together as satisfying a Use Case requirement for a larger application. For example, a user would navigate to the course search page to get information about a particular course, using a course number or description, or an instructor's name to discover which course he or she teaches. The input page, using a form, would take the user input and send it to a Struts `Action`, which would take care of retrieving course information. The Struts Form page collects the information needed by the code generator to create an initial input page and a `Form` class to contain the input data. The Struts Wizard uses the convention that every `Action` should have the granularity

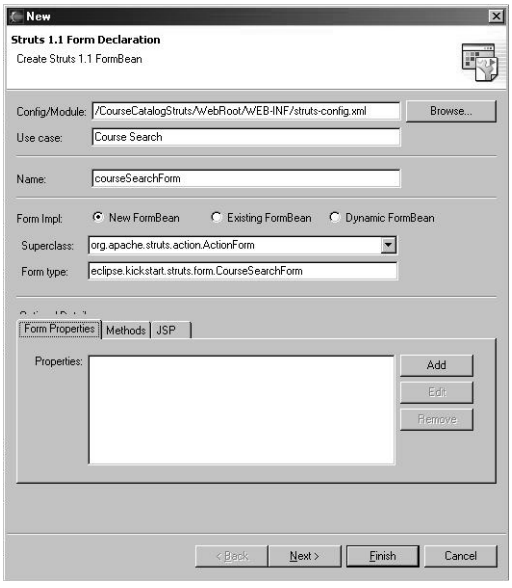

FIGURE 11.4 The Struts Form Wizard, complete with valid entries. (Due to resolution issues, the string Optional Details is covered by the Form Properties tab.)

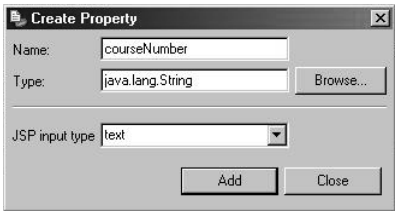

FIGURE 11.5 The Struts Create Property dialog with the `courseNumber` entry.

of a Use Case and therefore tries to name the `Form` and the `Action` after the Use Case. Because this is the Course Search Use Case, enter in the Use Case field `Course Search`. The Form Name field is automatically set to `courseSearchForm`, and the Form Type field is set to `eclipse.kickstart.struts.form.CourseSearchForm`. Using the drop-down button, change the Superclass setting from `<default>` to `org.apache.struts.action.ActionForm`.

At this point, the `Form` class is empty. To add form fields, click Add (found in the Form Properties section next to the empty text area). The dialog that appears allows you to set the name of the form field, the Java type to which it maps, an optional initial value, and the JSP input type (see Figure 11.5). One at a time, enter into the Name field `courseNumber`, `courseName`, and `instructor`, leaving Type set to `java.lang.String`, Initial Value set to blank, and the JSP Input Type set to `text`. Click Add until you have entered all three field names. When the dialog reopens for the fourth time, click Close. The Form Properties window lists `courseNumber`, `courseName`, and `instructor` as the defined form properties. Select the Methods tab and leave the top two selected methods, `validate()` and `reset()`, checked. Next, select the JSP tab and change the path from `/form/courseSearch.jsp` to `/courseSearch.jsp`. Click Next.

The next page of the wizard, Struts Action Declaration (see Figure 11.6), defines the `action-mapping` and `action` elements found in `struts-config.xml`. The entry in the Path field is the path alias used by the input HTML page to call the custom `Action`. In the HTML file, the path needs to end with `.do` if the Struts HTML tag library is *not* used. If the Struts HTML tag library is used, the Struts custom `form` tag will take care of adding `.do` to the form submission target. When you are done with the wizard (do not click Finish yet), you will find

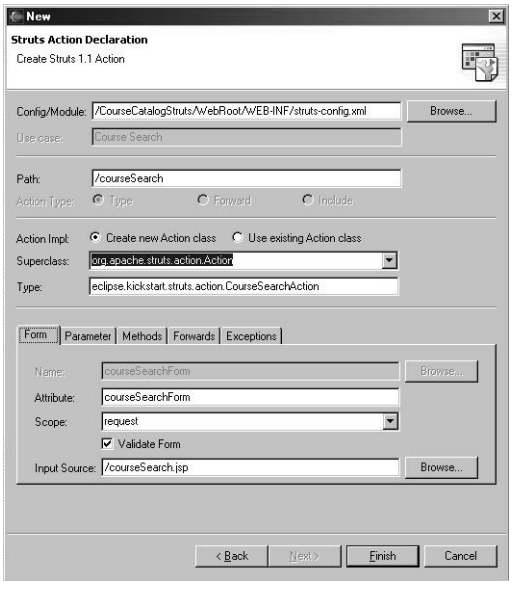

FIGURE 11.6 The Struts Action page displaying the ActionMapping information.

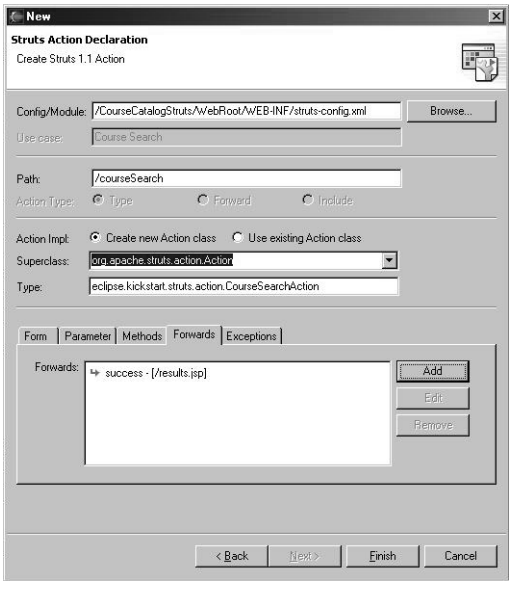

FIGURE 11.7 The Struts Action page with the "success" forward target set.

that the input JSP has a call to `/courseSearch` as the form action. Returning to the dialog, the Type field has been filled automatically with an acceptable Struts `Action` name based on the Use Case from the previous page, and the attribute name is filled in as well, using the form type name. The Scope dictates the scope object where the form will be placed when it is created by the Struts framework. The Scope defaults to `request`. This scope is acceptable because the search information is only valid for the initial search request.

Part of the responsibility of `struts-config.xml` is to map the input form page to which the `ActionServlet` should return if validation of the form fails. To match what was done in the previous dialog page, change the Input Source field on the Form tab from `/form/courseSearch.jsp` to just `/courseSearch.jsp` for user convenience. Leave the Parameter tab's Parameter field empty and make sure that the Methods tab has the `execute()` checked that takes in an `HttpServletRequest/HttpServletResponse` object (the first choice). The last tab you need to worry about, the Forwards tab, involves the JSP page to which the `Action` object should forward the response when it completes its task. The forward you define is specific to this `Action`; it is not a global forward. Click Add to open a dialog to enter a name for the forward target and the actual resource that should be called when the target name is used. In this case, enter the forward name `success` and the forward path `/results.jsp` (make sure it has a leading slash). Click Add to add the forward entry and then click Close to return to the New dialog (see Figure 11.7). Unfortunately, the wizard will not create a stubbed-out version of this file.

Close the New dialog by clicking Finish. The `struts-config.xml` file has two new entries: `form-beans` and `action-mapping`. The relevant code is shaded:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC
➥"-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
➥"http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
    <data-sources />
    <form-beans >
        <form-bean name="courseSearchForm"
➥type="eclipse.kickstart.struts.form.CourseSearchForm">
            <form-property name="instructor" type="java.lang.String" />
            <form-property name="courseName" type="java.lang.String" />
            <form-property name="courseNumber" type="java.lang.String" />
        </form-bean>

    </form-beans>

    <global-exceptions />
    <global-forwards />
    <action-mappings >
    <action
        attribute="courseSearchForm"
        input="/courseSearch.jsp"
        name="courseSearchForm"
        path="/courseSearch"
        type="eclipse.kickstart.struts.action.CourseSearchAction"
        unknown="false"
        validate="true">
    <forward
        name="success"
        path="/results.jsp"
        redirect="false"
        contextRelative="false" />
    </action>

    </action-mappings>

    <controller
        bufferSize="4096"
        debug="0"
        locale="false"
        nocache="false"
        inputForward="false" />
    <message-resources parameter="eclipse.kickstart.struts.ApplicationResources" />
</struts-config>
```

The `struts-config.xml` file has been modified, so save the file. Looking at the Outline view shows that the `form-beans` and `action-mappings` nodes have entries in them. For a more visual view of what you have just accomplished, click the Flow View tab located toward the bottom of the Struts editor window (see Figure 11.8). It displays the action as a flow from the starting input file (`/courseSearch.jsp`) to the `Action` object (`/courseSearch.do`) to the output target (`/results.jsp`).

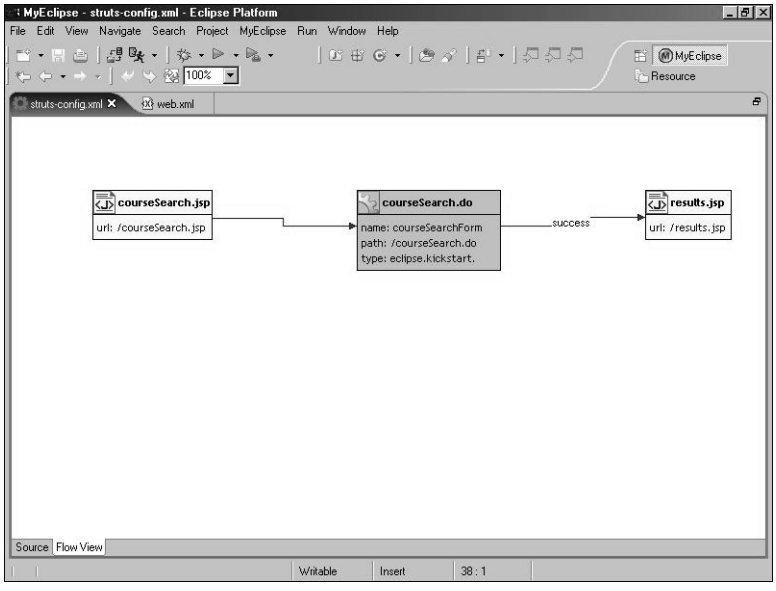

FIGURE 11.8 The Flow View displays the input, action, and output of the Struts components.

Implementing `CourseSearchAction`

From the Flow View, open `CourseSearchAction` by double-clicking `/courseSearch.do`. A cursory examination of the code shows that XDoclet has been hard at work generating the Struts files defined through the wizard. A look at the Javadoc for the class shows two XDoclet tags—`@struts:action` and `@struts:action-forward` (formatted to fit the page):

```
/**
 * MyEclipse Struts
 * Creation date: 06-07-2004
 *
 * XDoclet definition:
 * @struts:action path="/courseSearch" name="courseSearchForm"
 *                input="/courseSearch.jsp" validate="true"
 * @struts:action-forward name="/results.jsp" path="/results.jsp"
 */
```

The `execute()` method of `CourseSearchAction` has two lines of code: a line that assigns the generic `Form` object to a variable of type `CourseSearchForm`, and another line that throws an exception when the `Action` is called. Let's remove the exception and instead return the `ActionForward` that refers to your "success" target. The comments have been removed from the following code for brevity:

```
public class CourseSearchAction extends Action {

    public ActionForward execute(
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response) {
        CourseSearchForm courseSearchForm = (CourseSearchForm) form;

        return mapping.findForward("success");
    }

}
```

By the time this code is executed, Struts has taken care of filling and validating the form object and, after the code calls the search object, forwarding the request to `results.jsp` through the use of the alias `success`. The use of a forward name as an indirection to the forward target makes your implementation much more flexible. If you need to change your forward target, you just need to change the configuration file, not the code.

You have two things left to do to complete the example and run it within Tomcat: update the `CourseSearchForm` and create a basic output page. Let's start by updating the `CourseSearchForm` to make the `validate()` and `reset()` methods relevant. The XDoclet-created code throws an `UnsupportedOperationException` when `validate()` is called, but the logic should be a little more reasonable:

```
public ActionErrors validate(
    ActionMapping mapping,
    HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();

    if (instructor.trim().length() == 0) {
        errors.add("instructor", new ActionError("instructor.error"));
    }

    if (courseNumber.trim().length() == 0) {
        errors.add("courseNumber", new ActionError("courseNumber.error"));
    }
```

```
    if (courseName.trim().length() == 0) {
        errors.add("courseName", new ActionMessage("name.error"));
    }

    return errors;
}
```

Make `reset()` assign an empty string to the various fields:

```
public void reset(ActionMapping mapping, HttpServletRequest request) {

    courseNumber = "";
    courseName = "";
    instructor = "";
}
```

Press Ctrl+Shift+O to remove compile problems related to missing imports.

The `validate()` method is now well-behaved. It creates an `ActionMessage` object each time there is a validation problem, and the `ActionServlet` is free to return the `ActionErrors` object to `courseSearch.jsp`. Of course, each of the `ActionMessage` objects is referencing a key in the `ApplicationResources.properties` file, which you have not updated. `ApplicationResources.properties` is found in the `eclipse.kickstart.struts` package. The strings referenced in each of the `ActionMessage` constructors constitute the keys:

```
# Resources for parameter 'eclipse.kickstart.struts.ApplicationResources'
# Project P/CourseCatalogStruts
instructor.error=Instructor name cannot be blank.
courseNumber.error=Course number cannot be blank.
name.error=Course name cannot be blank.
```

The input page, `courseSearch.jsp`, is already instrumented to handle an incoming `ActionErrors` object. Open `courseSearch.jsp` in the JSP editor. Each of the individual `<html:text>` lines has an associated `<html:error>`.

Let's create the output file `results.jsp`. In the Package Explorer view, select `WebRoot` in the CourseCatalogStruts project. Press Ctrl+N to open the New dialog and then select J2EE, Web, JSP. Click Next.

The JSP Wizard page only needs three pieces of information: the location to where the JSP should be written (File Path), the name of the JSP (File Name), and the template file to be used to create a starting file (Template to Use):

- **File Path**—`/CourseCatalogStruts/WebRoot`
- **File Name**—`results.jsp`
- **Template to Use**—Standard JSP Using Struts 1.1

After you've provided this information, click Finish.

The new JSP, `results.jsp`, is filled with setup information. Look for the line

```
This a struts page. <br>
```

In place of this line, substitute the following:

```
Course Number: <bean:write name="courseSearchForm" property="courseNumber"/><br>
Course Name:   <bean:write name="courseSearchForm" property="courseName"/><br>
Instructor:    <bean:write name="courseSearchForm" property="instructor"/><br>
```

Now you can save `results.jsp`.

Before you deploy, check the form target in `courseSearch.jsp`. The version of MyEclipse used in this chapter did not fill in the form action with the Struts action declared in the Struts Wizard. The form action in `courseSearch.jsp` should read `/courseSearch`. If it does not, update it to do so.

If you still have Tomcat running from the previous example, stop it by selecting the running guy with the smiling face behind him. Click the downward-pointing triangle and click Tomcat 5, Stop.

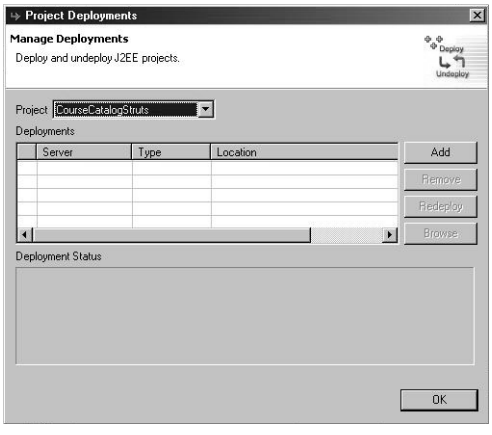

FIGURE 11.9 The Project Deployments dialog with no available deployment configurations.

From the Package Explorer, right-click the CourseCatalogStruts project and then click MyEclipse, Add and Remove Project Deployment to open the Project Deployments dialog. The Project Deployments dialog displays available projects and their deployment profiles. Select CourseCatalogStruts in the Project field if it is not the current project (see Figure 11.9). No deployment profiles are available for it, so you will have to create one the same way you created one for CallCenterWeb: by clicking Add and, when the New Deployment dialog opens, selecting Server Tomcat 5 and the deploy type Exploded Archive. The Deploy Location field will display the directory where the Web app will be installed (see Figure 11.10). Click Finish. Adding the Deployment profile for the first time takes care of doing the initial deployment. Click OK to complete the deployment and close the Project Deployments dialog.

Restart Tomcat by selecting Tomcat 5, Start from the toolbar. You are ready to go when Tomcat displays the following message:

```
INFO: Server startup in 18667 ms
```

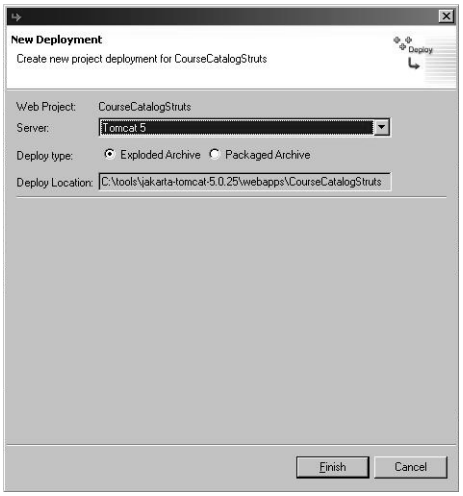

FIGURE 11.10 The completed deployment configuration for Tomcat 5.

Open a Web browser and point it to your Struts application:

```
http://localhost:8080/CourseCatalogStruts/
courseSearch.jsp
```

The HTML page displays a very plain look and feel, with just the three fields and two buttons. Without entering any information in the fields, click Submit. The `validate()` method will create `ActionMessage` objects for each of the fields, and the `ActionServlet` will call `courseSearch.jsp` to handle displaying the error messages contained in the `ActionErrors` object (see Figure 11.11). The `CourseSearchAction` has not been called as of yet.

Enter information in the various fields and click Submit. The `results.jsp` page displays the entered information (see Figure 11.12). Success! Your first Struts Web application using MyEclipse is done.

SHOP TALK

Struts and the Art of Layers and Indirection

The exercise you just ran through brings to mind a number of issues involved in the use of frameworks and indirection.

First of all, it is important to remember that a framework encapsulates many best practices, but it cannot encapsulate them all. Struts is no different and makes it a point to remind developers that they are responsible for the "M" part of the MVC design used in Struts. The model is the core of the application and needs the most work. The Struts example uses a form object to store the information from the input page, but it does not pass the form to any kind of search service, for the sake of brevity. However, even if there were a datasource of some kind, the `Action` should not pass the form in anyway. Passing the form would increase the coupling between the presentation and the data source layers. It would be up to the designer/implementer of the search service to create an object, or objects, to hold lookup criteria to increase the cohesion but decrease the coupling. Not knowing how a client is going to use the service would encourage the service to use this `SearchCriteria` object to do its job generically. It is the difference between having an `Action` do this:

```
// Cast the incoming form
SearchForm strutsSearchFromObject = (SearchForm)form;

// Get the search service to look up a course
searchService = … // Retrieve the SearchService from somewhere…
Course [] course = searchService.searchFor(strutsSearchFormObject);
```

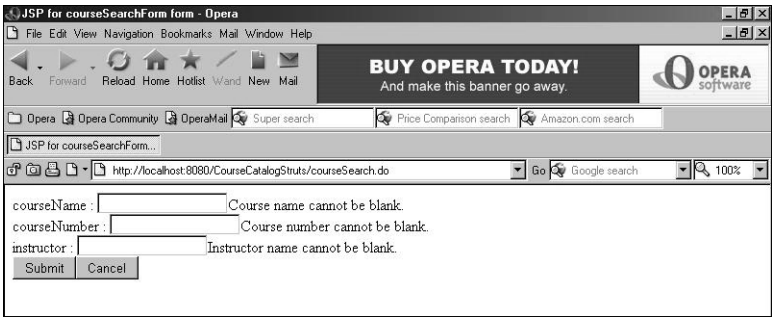

FIGURE 11.11 The course search page displaying error messages next to each field.

Struts is interesting enough that quite a bit of what you may have just discovered has less to do with Eclipse and more to do with Struts. The MyEclipse Wizards copied all the files you needed, created an input file, the `ActionForm`, the `Action`, and the `ApplicationResource` file. The MyEclipse JSP Wizard also has built-in templates to create JSP files that directly support the Struts tag libraries and Struts forms.

or this:

```
// Cast the incoming form
SearchForm strutsSearchFormObject = (SearchForm)form;

// Get the search service to look up a course
searchService = … // Retrieve the SearchService from somewhere…

// Copy the form information into a search service-specific object.
strutsSearchFormObject.copyTo(searchServiceCriteriaobject);
Course [] course = searchService.searchFor(searchServiceCriteriaobject);
```

By including the `CourseSearchForm` in the API of the `CourseSearchService` implies that the service is only being used by the `Action` and no one else, which is always a possibility, but not likely.

Does this mean that you would have to have code in various places in your `Action` to copy the data from the form to the criteria object? Not at all. Because the object controlling the data is the form, you would implement a `copyTo()` method in the form that would recognize the criteria object and take care of copying any data into it.

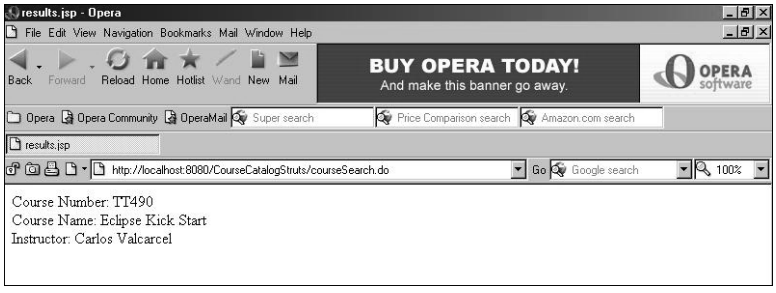

FIGURE 11.12 The output from `results.jsp`.

For the next few additions to the current example, you should stop Tomcat and close all your editor windows by pressing Ctrl+Shift+F4. The additions have to do with scenarios such as the following:

- All `Action objects` need to call a particular forward if certain information is entered.
- When an exception is thrown from any `Action`, the Web application should always display the same error page based on exception type.
- You will remove the code-based form altogether and use a configuration-driven form instead.
- You will configure a JDBC source for use by the Struts application.

MyEclipse gives you direct Struts support for these scenarios, in the creation of global forwards, global exceptions, `DynaActionForms`, and datasources. You will add one of each of these to the CourseCatalogStruts project, and you will modify the `CourseSearchAction` to use them all.

Adding Global Forwards

Let's create a global forward that the `CourseSearchAction` will use when an unknown course number is entered. A *global forward* is the mapping of a named resource that is available to all action-mapping elements. To create a global forward using MyEclipse, open `struts-config.xml`, go to the Outline view, right-click the `global-forward` node, and select New Forward to open the Struts 1.1 Forward Declaration dialog. You can declare global and local forwards using this dialog. By clicking a particular forward scope, you cause the wizard to assign the forward information either to a particular `Action` or to a global-forward element. For this example, ensure that Global Forward is the selected radio button in Forward Scope. Enter the forward name `unknown` and enter the forward path `/unknown.jsp` (see Figure 11.13). Click Finish. The `struts-config.xml` file is not saved by the wizard on completion, so save the file by pressing Ctrl+S. The `global-forward` element generated by the wizard displays the following information:

```
<global-forwards >
<forward
    name="unknown"
    path="/unknown.jsp"
    redirect="false"
    contextRelative="false" />

</global-forwards>
```

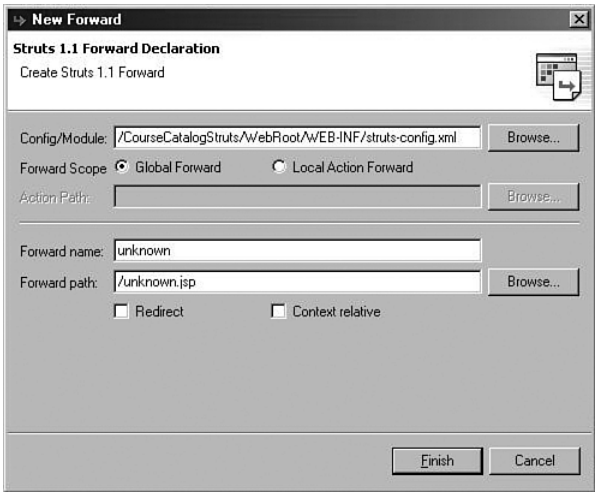

FIGURE 11.13 The New Forward dialog with a complete name and path.

Add a conditional to the `Action` so that all course numbers but one will cause the `unknown.jsp` page to be displayed (you will create `unknown.jsp` next). Open `CourseSearchAction`, located in the `eclipse.kickstart.struts.action` package, in the Java editor. Update `execute()` to check for course number TT490. If the right course number comes in, `results.jsp` is called and all other course numbers get `unknown.jsp`. Change `execute()` to check for the incoming course number and return the proper `ActionForward`:

```
public ActionForward execute(
    ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response) {
    CourseSearchForm customerSearchForm = (CourseSearchForm) form;

    ActionForward forward = null;
    if (customerSearchForm.getCourseNumber().equalsIgnoreCase("TT490")) {
```

```
            forward = mapping.findForward("success");
        } else {
            forward = mapping.findForward("unknown");
        }

        return forward;
    }
```

With the following pieces in place, you can run the Struts application again. Start Tomcat, open you browser, navigate to `http://localhost:8080/CourseCatalogStruts/courseSearch.jsp`, enter "TT490" as the course number, and supply any information in the remaining two fields. Click Submit to see the usual `results.jsp` page. To see the call to `unknown.jsp`, click the Back button on your browser and change TT490 to TT491, or any other string that strikes your fancy. The result is an error from Struts to Tomcat stating that the request resource is unavailable. It is unavailable because you have not written `unknown.jsp`.

To create `unknown.jsp` in the Package Explorer, view select `WebRoot` in the CourseCatalogStruts project. Press Ctrl+N to open the New dialog and then select J2EE, Web, JSP. Click Next to go to the JSP Wizard page. Make sure File Path is set to `/CourseCatalogStruts/WebRoot`, File Name is set to `unknown.jsp`, and Template to Use is set to Standard JSP using Struts 1.1. Click Finish.

When the JSP editor opens, look for the following line:

```
This a struts page. <br>
```

In that line's place, put the following:

```
The following course was not found in the course catalog:<br>
<bean:write name="courseSearchForm" property="courseNumber" />
```

Redeploy the application either by clicking the button next to the MyEclipse Application Server button or by right-clicking the CourseCatalogStruts project and selecting Add and Remove Project Deployments. When the Project Deployments dialog opens, select CourseCatalogStruts as the project, select the Tomcat 5 server entry in the Deployments list, and click Redeploy.

Once again, open your browser and go to `http://localhost:8080/CourseCatalogStruts/courseSearch.jsp`. Enter a random course number and arbitrary information into the two remaining fields. When you click Submit, the `ActionServlet` will execute the `CatalogSearchAction`, which will ask the `ActionServlet` to forward the request to `unknown.jsp`, which in turn displays your page (see Figure 11.14).

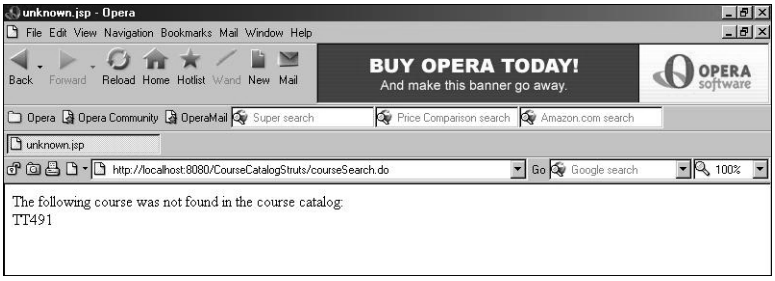

FIGURE 11.14 The browser displaying output from `unknown.jsp`.

Replacing Java-Based Forms with `DynaActionForm`

The next Struts technology supported in MyEclipse is the `DynaActionForm`. You can use `DynaActionForm` as a partial or complete replacement of your form, but when you use it as a complete replacement, you lose the ability to programmatically control the form's validation. An alternative to programmatic validation would be the Validator framework, which is part of the Jakarta Commons project. The Validator framework defines a reusable and extendible rule-based validation framework that lets you declare validation rules in an XML file as well as allows you to write your own rules that can be added to the framework. It is not directly related to Struts and therefore can be used to validate any kind of object. However, MyEclipse does not give you any direct support for the Validator framework. If you decide that the Validator framework is appropriate for your work, you would have to add the supporting pieces by hand. You will not be adding Validator support to any of the succeeding examples.

Let's look at both uses of the `DynaActionForm` and how MyEclipse supports them. The first one you will implement is the total replacement version. Rename the current form by right-clicking on `CourseSearchForm` and selecting from the pop-up menu Refactor, Rename. When the Rename Compilation Unit dialog opens, enter a new name of `CourseSearchFormSave` and make sure that no check boxes are selected. Click OK.

At this stage, you have a number of editors open. Close them all by using the keyboard shortcut Ctrl+Shift+F4. From the Package Explorer, double-click `CourseCatalogStruts`, `WebRoot`, `WEB-INF`, `struts-config.xml`. Delete the entry for `CourseSearchForm` by deleting the `form-bean` element between the `form-beans` tags. The new entry should be an open and close tag for `form-beans`. Save the `struts-config.xml` file:

```
<form-beans >
</form-beans>
```

In the Outline view, right-click `form-beans` and select New Form from the pop-up menu. When the New dialog opens, it opens on the same page you used when you first created the Struts application. Enter in the Use Case field `Course Search`. This will fill in the Form Name field with `courseSearchForm`, which is the name that will be entered in `struts-config.xml` as

the form name. In Form Impl, click Dynamic FormBean, which changes the Dynamic Type field to use `org.apache.struts.action.DynaActionForm`. Click Add in the Form Properties section and enter the three field names `courseNumber`, `courseName`, and `instructor`, all with a type of `java.lang.String`. Remember to click Close when the Form Property dialog opens for the fourth time. Click Finish. The `struts-config.xml` file has been changed but not saved, so save the file.

The new `form-beans` entry now contains a type of `org.apache.struts.action.DynaActionForm` and three `form-property` elements, one for each of the form fields you entered. Take a look at your project: `CourseSearchForm` is not defined anywhere. In order for any code to access the form, it must now cast the form to be an object of type `DynaActionForm` and use the form's API to access the data:

```
<form-beans >
    <form-bean name="courseSearchForm" type="org.apache.struts.action.DynaActionForm">
        <form-property name="instructor" type="java.lang.String" />
        <form-property name="courseName" type="java.lang.String" />
        <form-property name="courseNumber" type="java.lang.String" />
    </form-bean>
</form-beans>
```

Open `CourseSearchAction` and change any references to `CourseSearchForm` to `DynaActionForm`. Remember to let the editor help you in typing out `DynaActionForm` (Ctrl+spacebar) and in correcting the import list (Ctrl+Shift+O). Also, change `if()` to look up the course number field using `DynaActionForm.get()`:

```
    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
        DynaActionForm customerSearchForm = (DynaActionForm) form;

        ActionForward forward = null;
        String courseNumber = (String) customerSearchForm.get("courseNumber");
        if (courseNumber.equalsIgnoreCase("TT490")) {
            forward = mapping.findForward("success");
        } else {
            forward = mapping.findForward("unknown");
        }

        return forward;
    }
```

Save the file. Redeploy the Web application, restarting Tomcat if necessary. Your results should be the same as before: When you enter TT490 as the course number, the `results.jsp` page is displayed, and when you enter anything else as a course number, you get `unknown.jsp`. A different behavior will be when you leave any or all of the fields as blank, the `ActionServlet` does not send the form back to `courseSearch.jsp`; instead, it goes to either `results.jsp` or `unknown.jsp`.

Using a `DynaActionForm`, as convenient as the previous example portrays it to be, does lose validation that can be replaced using the Validator framework. Another option is to create a subclass of `DynaActionForm`, in conjunction with the definitions in `struts-config.xml`, and add `validate()` to the subclass. The use of this combination entails the following tasks:

- Subclassing `DynaActionForm` and adding `validate()`.
- Changing the `form-bean` type entry to the subclass.

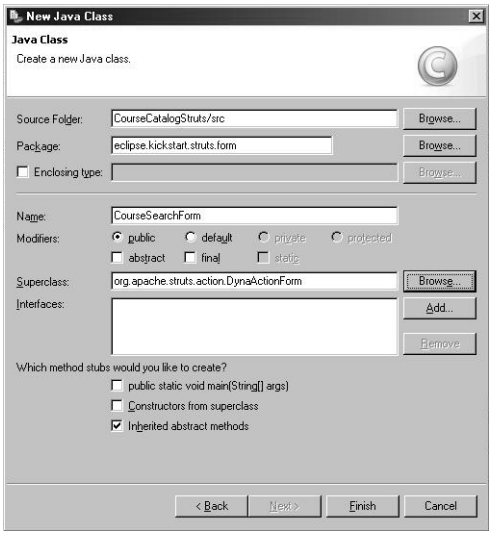

FIGURE 11.15 The New Java Class dialog with the information needed to generate the `DynaActionForm` subclass.

Let's create the hybrid `DynaActionForm` class. By rights, if you were doing this from scratch, you would use the MyEclipse Form Wizard to get you to where we are now, and the wizard would perform the following steps. Select the `eclipse.kickstart.struts.form` package in the CourseCatalogStruts project. Press Ctrl+N to open the New Java Class dialog; then select Java, Class and click Next. Enter in the Name field `CourseSearchForm` and click the Browse button next to the Superclass field. Entering the string `dyn` in the Superclass Selection dialog will bring `DynaActionForm` to the top of the list. Select it and click OK. The New dialog now has enough information to generate the stub class (see Figure 11.15). Click Finish.

When the `CourseSearchForm` opens in the Java editor, right-click anywhere in the editor and from the pop-up menu select Source, Override/Implement Methods. The Override/Implement Methods dialog lists all the concrete and abstract methods available to your subclass. Scroll and open the ActionForm node, check `validate(ActionMapping, HttpservletRequest)`, and click OK (see Figure 11.16).

Change `validate()` to perform a check on the form fields by getting the current field value from itself using the `get()` method and the desired field name:

```
public ActionErrors validate(
        ActionMapping mapping,
        HttpServletRequest request) {
        ActionErrors errors = new ActionErrors();

        String field = (String) get("instructor");
        if (field.trim().length() == 0) {
            errors.add("instructor", new ActionMessage("instructor.error"));
        }
```

```
        field = (String) get("courseNumber");
        if (field.trim().length() == 0) {
            errors.add("courseNumber", new ActionMessage("courseNumber.error"));
        }

        field = (String) get("courseName");
        if (field.trim().length() == 0) {
            errors.add("courseName", new ActionMessage("name.error"));
        }

        return errors;
    }
```

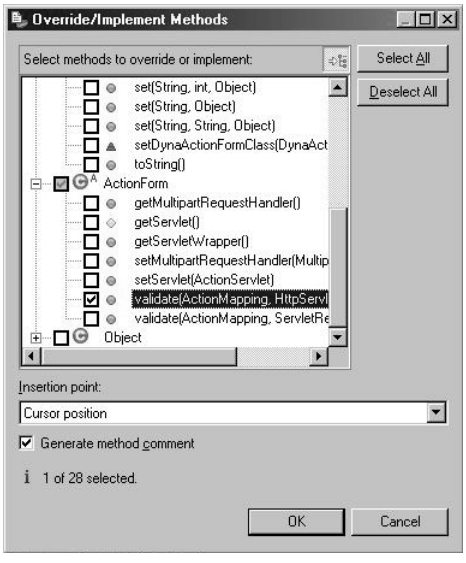

FIGURE 11.16 The list of available methods for `CourseSearchForm` to implement or override.

Now that you have a subclass of `DynaActionForm`, add it to `struts-config.xml` in the `form-bean` element type attribute (the shaded code):

```
    <form-bean
        dynamic="true"
        name="courseSearchForm"
        type="eclipse.kickstart.struts.form.CourseSearchForm">
        <form-property name="instructor" type="java.lang.String" />
        <form-property name="courseNumber" type="java.lang.String" />
        <form-property name="courseName" type="java.lang.String" />
    </form-bean>
```

Redeploy the application by selecting Add and Remove Project Deployments or by clicking the Redeployment button in the toolbar. Once you are redeployed, restart Tomcat, open your browser, and navigate to `http://localhost:8080/CourseCatalogStruts/courseSearch.jsp`. Click Submit without entering any information. The error messages will again be displayed.

Adding Global Exceptions

Action forwards define explicit targets that can be called by `Action` objects. Global exceptions define the flow for an exception that is not handled by your Web application. When an exception is caught by the `ActionServlet` and an exception element is defined within `struts-config.xml`, the framework retrieves the error message from the file where the messages are defined, wraps the string in an `ActionMessage` object, puts the `ActionMessage` object within an `ActionErrors` object, and passes the `ActionErrors` object to the forward target defined within the `exception` element. Let's create a global `exception` element that handles `IOExceptions` by sending an error message to an error page.

SHOP TALK

`DynaActionForm` Versus Subclassing `ActionForm`

If the `DynaActionForm` is so good, why is the `ActionForm` still supported in Struts? Why bother using one over the other? The answer is one of convenience: If you prefer to update configuration files instead of code, the `DynaActionForm` is a great way to go. Conventional wisdom goes like this: When a change needs to be made, you update `struts-config.xml` and continue on your way. Subclasses of `ActionForm` need to have code changed, which means that you have to take the file out of source control, you need to open an IDE to make the change, you need to run your tests to make sure that the change doesn't break anything or introduce new bugs, and so on.

I have to admit to feeling ambivalent about the `DynaActionForm`. As much as the `ActionForm` can appear to be more work, this is not as big a deal as it seems. If you are going to make a change to a form using `struts-config.xml`, you still need to take the file out of source control, you still need to update code somewhere to use the new field (or not use a deleted field), and you still need to run your tests to make sure that the change to the configuration file doesn't break anything else. (In my opinion, configuration file changes can create bugs that are much harder to track down than bugs created by code changes, unless you have good unit tests in place to prove otherwise.)

But what about the hassle of creating the Java class that maps to the form, defines the fields and their types, as well as the getter and setter methods? As you have seen in previous chapters, Eclipse lets you define a class using a wizard and takes care of generating the code to wrap access to instance fields from the editor. Also, defining the form using MyEclipse means the plug-in creates the `ActionForm` subclass for you with minimal effort.

There is one other advantage to creating the form as a subclass of `ActionForm`: The subclass's API to the instance fields is type-safe, whereas the `DynaActionForm` relies on you casting the various accesses to the data to the proper type.

Close all your editor windows except for `struts-config.xml` (double-click it to open it in the MyEclipse Struts editor if it is not already opened). In the Outline view, right-click `global-exceptions` and select from the pop-up menu New Exception to open the New Struts Exception dialog.

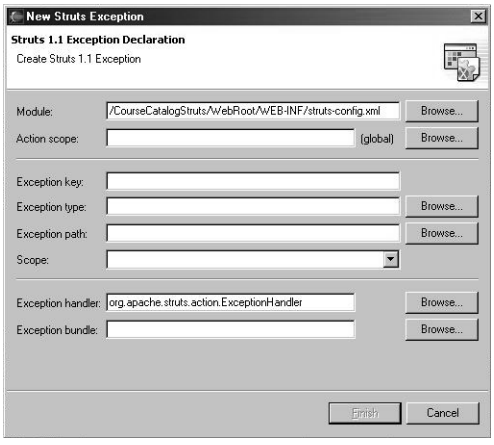

FIGURE 11.17 The New Struts Exception dialog.

The Struts 1.1 Exception Declaration page is divided up into three sections (see Figure 11.17):

- The location of the exception definition and its scope (global to all `Actions` or local to one)
- The type of exception this definition will handle, the error message to be displayed when it occurs, and the path to where the exception will be forwarded
- The type of the Java object that will handle the incoming exception and a resource file where it can look up the error message using the key from the previous section

All the fields in the wizard are required except for Action Scope and Exception Bundle. The exception will be assigned to the `global-exception` element if no action scope is declared; otherwise, the exception will be local to a particular action. The exception bundle defines a properties file that contains the error messages to be used when an exception is caught; if no file is declared, the file declared in `message-resources` is used.

Set the fields in the New Struts Exception dialog as follows:

- **Action Scope**—Leave this field blank.
- **Exception Key**—`ioe.error`.
- **Exception Type**—`java.io.IOException`. (Click the Browse button and use the dialog to find it.)
- **Exception Path**—`/error.jsp`.
- **Exception Handler**— `org.apache.struts.action.ExceptionHandler`.
- **Exception Bundle**—Leave this field blank.

Click Finish and save `struts-config.xml`. The `global-exceptions` element is ready for use. The `exception` element does not declare the use of `org.apache.struts.action.ExceptionHandler` because it is the default:

```
<global-exceptions >
    <exception
        key="ioe.error"
        path="/error.jsp"
        type="java.io.IOException" />

</global-exceptions>
```

For this first example of Struts-handled exceptions, you need to create an error page. Select CourseCatalogStruts, WebRoot and then press Ctrl+N to open the New dialog. Select J2EE, Web, JSP and click Next. In the JSP Wizard page, change the File Name field to `error.jsp` and leave the remaining two fields alone (Template to Use should be set to Standard JSP Using Struts 1.1). Click Finish.

In the `error.jsp` file, look for the line that reads

```
This a struts page. <br>
```

In place of this line, put in code to retrieve the `ActionErrors` object that is created when a `java.io.IOException` is caught:

```
<h1>Oh, no!  An error happened!</h1>
<html:errors />
```

Open the `ApplicationResources.properties` file found under CourseCatalogStruts, `src`, `eclipse.kickstart.struts`. Add a message with a key of `ioe.error` and save the file:

```
ioe.error=Major error trying to retrieve the data from the main server.<br>
```

The only thing left to do is have the `CourseSearchAction` throw an exception. Open `CourseSearchAction` in the Java editor and modify the condition to throw an `java.io.IOException` when a course number of "foobar" is entered. Make sure you add the `throws Exception` clause to `execute()`:

```
public ActionForward execute(
    ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response) throws Exception {
    DynaActionForm customerSearchForm = (DynaActionForm) form;

    String courseNumber = (String) customerSearchForm.get("courseNumber");
    ActionForward forward = null;
    if (courseNumber.equalsIgnoreCase("TT490")) {
        forward = mapping.findForward("success");
```

```
        } else if (courseNumber.equalsIgnoreCase("foobar")) {
            throw new IOException("Read/write error!");
        } else {
            forward = mapping.findForward("unknown");
        }

        return forward;
    }
```

Stop Tomcat. The `struts-config.xml` file has been changed, and the only way for `ActionServlet` to reread the file is to be reloaded. Redeploy the CourseCatalogStruts application using the Add and Remove Project Deployments dialog. Restart Tomcat within Eclipse, and when Tomcat has completed its startup, open your browser and navigate to `http://localhost:8080/CourseCatalogStruts/courseSearch.jsp`. Enter `foobar` into the course number field, some arbitrary strings into the remaining two fields, and click Submit. The `error.jsp` page will display with the message from the `ApplicationResources.properties` file (see Figure 11.18).

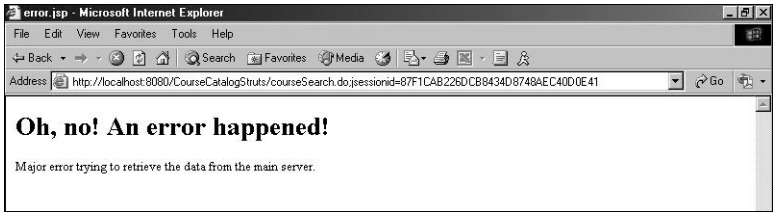

FIGURE 11.18 The `error.jsp` page displaying the error message from the thrown `IOException`.

If you subclassed `org.apache.struts.action.ExceptionHandler` to create your own exception handler, you could take the incoming exception and perform some intermediate operation before forwarding the request to the target forward.

Configuring a Struts Datasource

Another element defined within `struts-config.xml` is `data-source`. The definition of `data-source` allows an `Action` to access a JDBC datasource from within your application. Because a previous chapter defined a Hypersonic database, you will deploy your Struts-based application to JBoss and then you will modify `CatalogSearchAction` to access the `data-source` defined in `struts-config.xml`.

If you have not installed and downloaded JBoss, refer to the "Downloading and Installing JBoss" section of Chapter 9. It will walk you through the process of downloading and installing JBoss, the configuration of the JBoss-supplied Hypersonic database, the execution of the SQL to create a table, and the insertion of a few rows into the table.

On the assumption that you have JBoss downloaded, installed, and configured with its Hypersonic database set up with data, let's once again walk through creating a deployment configuration so that you can deploy to JBoss and test whether your Web application is still functioning.

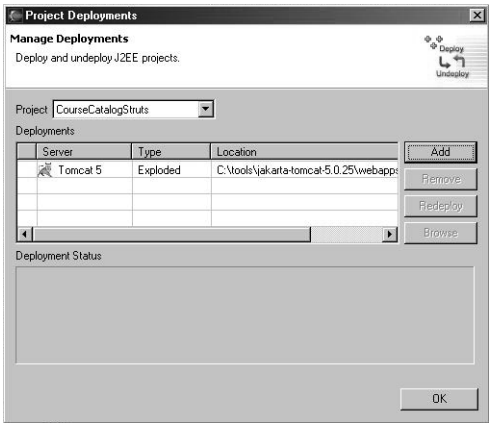

FIGURE 11.19 The Project Deployments dialog with the Tomcat deployment configuration.

Open the Project Deployments dialog by clicking the button next to the MyEclipse app server button on the toolbar, or you can right-click the CourseCatalogStruts project and select MyEclipse, Add and Remove Project Deployments from the pop-up menu. When the Project Deployments dialog opens, the Tomcat 5 deployment configuration appears as its only entry (see Figure 11.19). Select CourseCatalogStruts as the project and click Add. When the New Deployment dialog opens, select JBoss 3 as the server and select the deploy type Exploded Archive. Click Finish to return to the Project Deployments dialog, which now lists two deployment configurations: Tomcat 5 and JBoss 3. Click OK.

Once you selected JBoss as a deployment target, MyEclipse took care of deploying the application to it. If the configuration and deployment went well, you can call your Web application from a browser and see the input and output pages as you did before. To check the application, stop Tomcat (from the toolbar, select Tomcat 5, Stop) and then start JBoss (from the toolbar, select JBoss 3, Start). You cannot have Tomcat and JBoss running at the same time using their default configurations because they both use port 8080. JBoss is ready to go when you see the message telling you how long it took JBoss to start:

```
18:40:14,934 INFO  [Server] JBoss (MX MicroKernel) [3.2.3 (build:
➥CVSTag=JBoss_3_2_3 date=200311301445)] Started in 26s:137ms
```

Open your browser and navigate to the usual spot, `http://localhost:8080/CourseCatalogStruts/courseSearch.jsp`. To make sure your application is functioning, try the following scenarios:

1. Enter a course number of `TT490` with any instructor and any course name. When you click Submit, you should get the `results.jsp` output. Click the Back button on your browser. Leave the instructor and name fields alone for the rest of the scenarios.
2. Change the course number to `TT491` and click Submit. The output from `unknown.jsp` should appear. Click the Back button.
3. Change the course number to `foobar` and click Submit. The `error.jsp` page will display the "Oh, no! An error happened!" page.

If you were developing this as a real Web application, you would have had test fixtures in place to run these tests for you.

Before you can configure the Struts datasource, you have to put the JDBC driver you intend to use someplace where the Struts framework can find it and use it. If you had an external database running, this example could have been run from within Tomcat, as any JDBC driver will do as long as it is available to Struts. The use of JBoss is purely one of convenience.

Right-click `CourseCatalogStruts`, `WebRoot`, `WEB-INF`, `lib` and then select Import from the pop-up menu. When the Import dialog opens, select File System as the import source and click Next.

On the File System page, you need to navigate to the JBoss directory where the HSQL driver is located, which is `<JBoss install directory>\server\default\lib`. Click the top Browse button and navigate to the proper directory. When you have found the `lib` directory, click OK to enter the path into the From Directory field. The left window lists `lib` as a folder, and the right window lists the JAR files the directory contains. Put a check next to `hsqldb.jar`. The Into Folder field must read `CourseCatalogStruts/WebRoot/WEB-INF/lib` (see Figure 11.20). Click Finish. The `hsqldb.jar` file is now located in your project `WEB-INF/lib` directory.

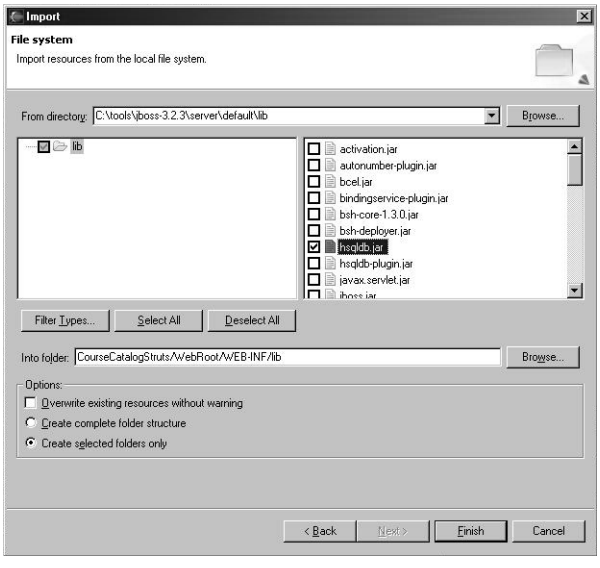

FIGURE 11.20 The File System import page with the required data.

One last thing before you can configure the `struts-config.xml` file: You need to put the HSQL driver JAR file in the build path of your project. The import should have taken care of that, but to be sure, from the Package Explorer open the Properties dialog by right-clicking the CourseCatalogStruts project name and selecting Properties from the pop-up menu. When

the Properties dialog opens, select Java Build Path from the left and the Libraries tab from the right. The `hsqldb.jar` file should be one of the JAR files in the build path of your project. If it is not, click Add External JARs and navigate to your project workspace `WEB-INF/lib` directory and include it. If the `hsqldb.jar` file is there, click OK to close the Properties dialog.

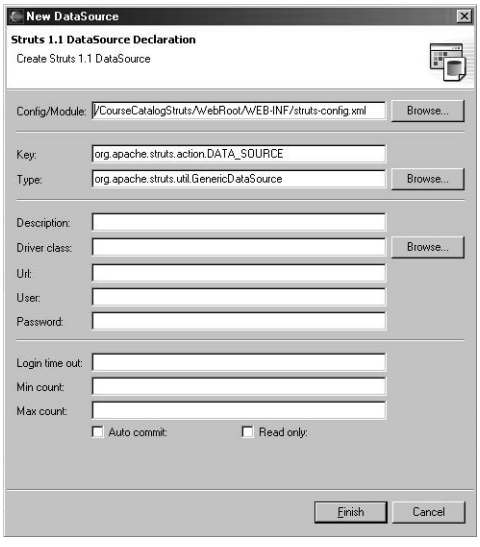

FIGURE 11.21 The New DataSource dialog displaying the Struts 1.1 DataSource Declaration page.

Open `struts-config.xml`. In the Outline view, right-click `data-sources` and select New DataSource from the pop-up menu to open the New DataSource dialog, where you define the Struts DataSource (see Figure 11.21). The first three fields are already filled and should be set as follows:

- **Config/Module Path**—`/CourseCatalogStruts/WebRoot/WEB-INF/struts-config.xml`
- **Key**—`org.apache.struts.action.DATA_SOURCE`
- **Type**—`org.apache.struts.util.GenericDataSource`

The next section contains JDBC driver information:

- **Description**—This field can be left blank.
- **Driver Class**—`org.hsqldb.jdbcDriver`. (Use the Browse button to find it using the Superclass selection dialog.)
- **Url**—`jdbc:hsqldb:hsql://localhost:1701`. (This information is taken from the `hsqldb-ds.xml` file supplied with JBoss.)
- **User**—sa.
- **Password**—Leave this field blank.

The third section is datasource-configuration data:

- **Login Time Out**—Leave this field blank.
- **Min Count**—1. (This is the minimum number of connections to use.)
- **Max Count**—5. (This is the maximum number of connections to use.)
- **Auto Commit**—Check this option.
- **Read Only**—Leave this option unchecked.

Once all the preceding information is entered, click Finish. The `data-sources` element now contains the information needed to configure the default Struts datasource, `GenericDataSource`, at runtime:

```
<data-sources >
    <data-source>
        <set-property property="password" value="" />
        <set-property property="minCount" value="1" />
        <set-property property="maxCount" value="5" />
        <set-property property="user" value="sa" />
        <set-property property="driverClass" value="org.hsqldb.jdbcDriver" />
        <set-property property="description" value="" />
        <set-property property="url" value="jdbc:hsqldb:hsql://localhost:1701" />
        <set-property property="readOnly" value="false" />
        <set-property property="autoCommit" value="true" />
        <set-property property="loginTimeout" value="" />
    </data-source>

</data-sources>
```

Now that Struts has a JDBC datasource available for use by the `Action`, let's update the code for `CourseSearchAction` to use it. The first conditional is the only piece that has changed. No matter which instructor name is entered, the code will always use the name field from the customer table in the Hypersonic database:

```
    if (courseNumber.equalsIgnoreCase("TT490")) {
        DataSource ds = getDataSource(request);
        Connection con = ds.getConnection();
        Statement stmt = null;
        ResultSet rs = null;
        try {
            stmt = con.createStatement();
            rs = stmt.executeQuery(
                 "select * from customer where phone_number='1-222-333-4444'");
            rs.next();
            String name = rs.getString("name");
            customerSearchForm.set("instructor", name);
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            con.close();
        }
        forward = mapping.findForward("success");
```

```
} else if (courseNumber.equalsIgnoreCase("foobar")) {
    throw new IOException("Read/write error!");
} else {
    forward = mapping.findForward("unknown");
}
```

Stop JBoss, redeploy CourseCatalogStruts, and restart JBoss. Open your browser to `http://localhost:8080/CourseCatalogStruts/courseSearch.jsp` and enter `TT490` for the course number, `John Smith` as the instructor, and `Eclipse` for the course name. Click Submit. The `results.jsp` output should have an instructor name of Ronald Weasley (see Figure 11.22).

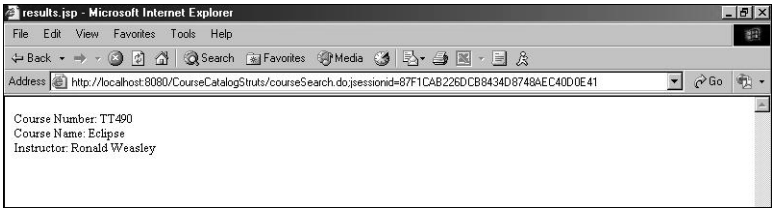

FIGURE 11.22 The final output of `results.jsp` showing the substitution of the name retrieved from the Hypersonic database.

Two Struts framework areas you are not going to try out using MyEclipse are the creation of `controller` components and the creation of `message-resources` because there is no direct support for either technology within MyEclipse.

In Brief

Struts developments using MyEclipse is easier than pulling the various pieces of the Struts framework together by hand. Not all aspects of Struts are supported within the plug-in, specifically the Validator framework and the `controller` configuration within `struts-config.xml`, but overall the support is solid and the various Struts pieces work with a minimum of effort.

- Struts projects are created from Web projects. Adding Struts capabilities to a Web project is accomplished by right-clicking the project and selecting MyEclipse, Add Struts Capabilities.
- Deployment of a Struts application to Tomcat is handled by a deployment wizard. The wizard works in conjunction with application server information entered in the Preferences dialog under MyEclipse, Application Servers.

- Global forwards are available to all `Actions` defined within `struts-config.xml`.
- `DynaActionForm` allows you to define a form without having to write an associated Java class, but you lose validation capability unless you create a subclass of `DynaActionForm` and include `validate()` or use the Validator framework.
- The `global-exceptions` element is supported by a MyEclipse wizard. Exceptions thrown by `Action objects` are caught by the `ActionServlet` and redirected to the forward target defined in the `global-exception` element.
- JDBC datasources are supported through a MyEclipse wizard that understands how to configure the default Struts JDBC datasource `GenericDataSource`.