[**Editor's Note:** The following excerpt is from Chapter 2 of the free eBook *The Developer Shortcut Guide to SUSE Linux* (Realtimepublishers) written by John Featherly and available at http://cc.realtimepublishers.com/portal.aspx?pubid=339.]

Chapter 2: Java Enterprise Applications

Java Enterprise Applications use J2EE (Java 2 Enterprise Edition) technology to create server based n-tier applications. N-tier design is an evolution of client/server design that can be thought of as 2-tier. The typical 3-tier application is made up of presentation, business logic, and back-end tiers. Java uses the term *JavaBean* to represent the component architecture for Java Objects. The mapping of JavaBean types to tiers in the 3-tier application consist of:

- Presentation tier to JSPs/Servlets
- Business logic tier to "session" JavaBeans
- Backend tier to "entity" JavaBeans

J2EE infrastructure provides *containers* for hosting components offering common resources and services. A complete J2EE application consists of multiple components hosted in containers that communicate and interact to achieve the overall goals of the application. The J2EE application developer writes the Java objects that are the core of all components. The developer also writes deployment descriptors that specify the parameters for packaging, installing, and running the component. The complete application package is bundled into a file called an Enterprise Archive (EAR). The Enterprise Archive can then be deployed to a J2EE Application Server, which unpacks it, sets up the components, and starts the application.

This chapter will look at the mechanics of writing the components and packaging the application and the Open Source tools available to assist the developer in both areas. It will also look at the Open Source J2EE Application Server product JBoss, developing a sample J2EE application and deploying it to a JBoss server.

The Sample Application

The sample application is a 3-tier application with a back-end data persistence tier, a middle application logic tier, and a user interface (UI) Web tier. To provide sample data and demonstrate functionality, the application records IP address and Ethernet hardware address relationships. The UI is a Web page that allows a user to look up a hardware address for a given IP address.

Writing the Application

The fundamental work of writing the application consists of writing Java code in .java files and compiling them into .class files, writing deployment and application descriptors in .xml files, and bundling the results together in .jar files and then into an .ear file. Although this process could be performed manually using a simple text editor, it is a complex undertaking. It is, of course, much more efficient to use developer tools.

Thus, this example will use the Open Source developer environment plug-in framework Eclipse that was introduced in Chapter 1. Eclipse comes with a standard Java plug-in, which you will use to support writing Java code for your components. You will make use of an additional plug-in for Eclipse to assist the J2EE tasks such as JavaBean templates and packaging definitions. The plug-in is called the JBoss Eclipse IDE. If you have installed Eclipse in a private directory on your SUSE Linux workstation, download the JBoss Eclipse IDE package from http://www.jboss.com and simply unpack the files into the Eclipse folder structure. If you have installed Eclipse on your workstation at the system level via YaST, you should use the software update feature internal to Eclipse. The SUSE Linux Professional 9.3 bundled version of Eclipse is version 3.0.1; thus, you will use JBoss Eclipse IDE version 1.4.1.e30. To do so, select Software Updates, Find and Install from the Help menu in Eclipse (see Figure 2.1).

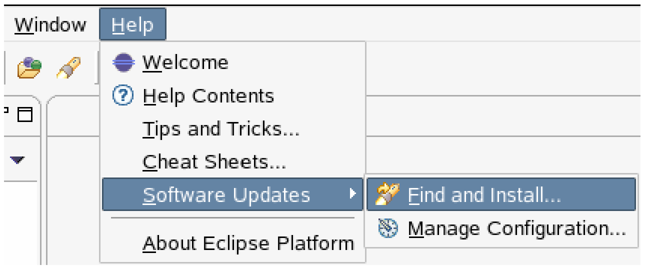
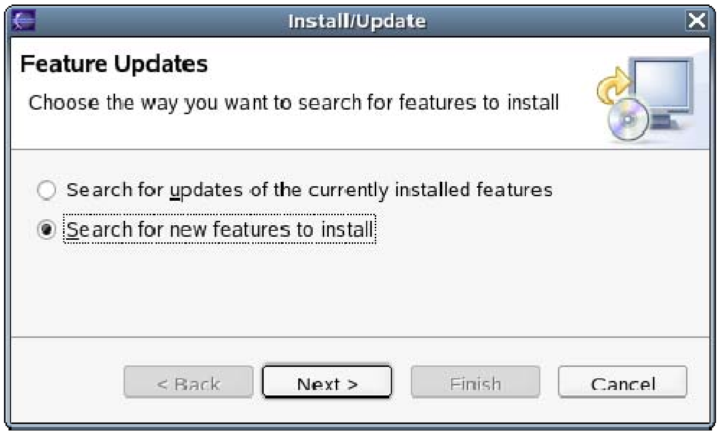


Figure 2.1: The first step in installing the JBoss IDE.

Next, select the *Search for new features to install* option in the subsequent dialog box, as Figure 2.2 shows.



Feature Updates

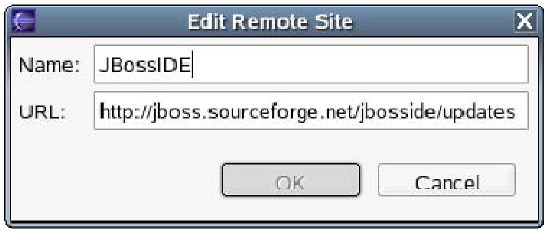
Choose the way you want to search for features to install

○ Search for updates of the currently installed features

● Search for new features to install

Figure 2.2: Selecting the option to search for new features.

Finally, add a new remote site with then name JBossIDE and the URL http://jboss.sourceforge.net/jbosside/updates as Figure 2.3 shows.



Name: JBossIDE
URL: http://jboss.sourceforge.net/jbosside/updates

Figure 2.3: Specifying the name and URL of the JBoss IDE.

Pick the appropriate version of Eclipse (3.0 is the version if you installed it from the SUSE Linux Professional 9.3 distribution), and click Finish. The JBoss Eclipse IDE package will be installed from the SourceForge site. You are now ready to use Eclipse to develop your J2EE application.

✎ This installation should be done while running Eclipse as root because the system Eclipse folder will be updated.

Creating the Entity Bean

The JBoss Eclipse IDE plug-in provides templates to assist in creating JavaBean components. There are templates for Entity Beans, Message Driven Beans, and Session Beans. Before you create the entity bean for your application, a J2EE project must be created. Select J2EE 1.4 Project under JBoss-IDE from the New Project selection dialog box (see Figure 2.4).

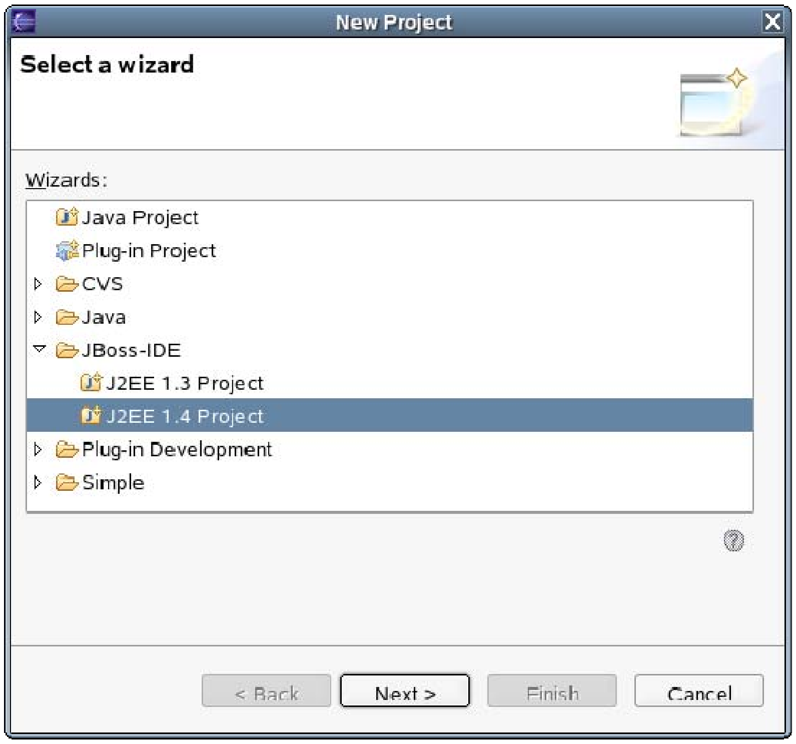
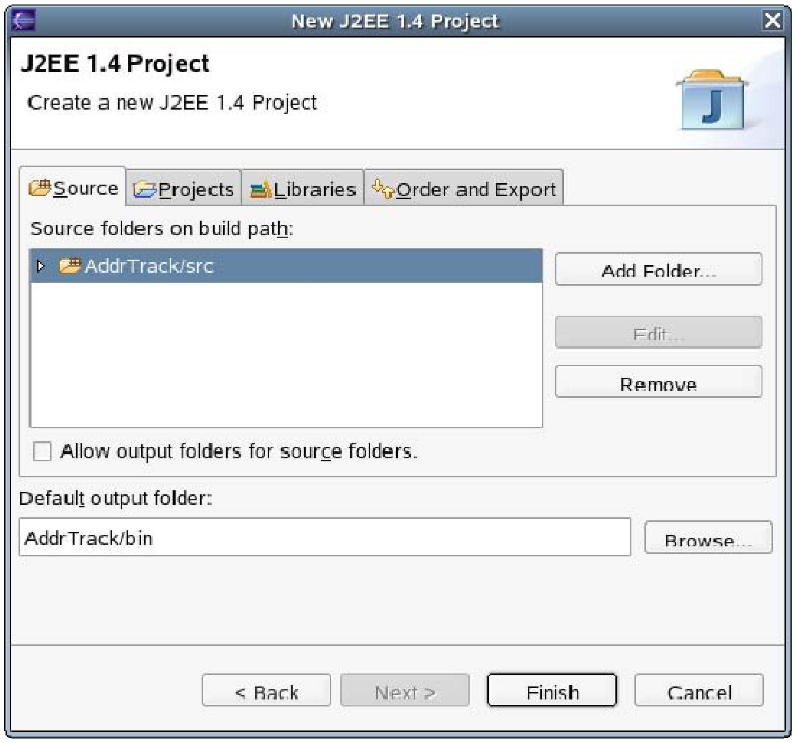


Figure 2.4: J2EE project selection.

Click Next, then enter the project name AddrTrack in the Project name box. Click Next again, and add a source folder, src, for the project. Also set the Default output folder to AddrTrack/bin, as Figure 2.5 illustrates.


New J2EE 1.4 Project

J2EE 1.4 Project

Create a new J2EE 1.4 Project

Source | Projects | Libraries | Order and Export

Source folders on build path:

AddrTrack/src

Allow output folders for source folders.

Default output folder:

AddrTrack/bin

Figure 2.5: Project source and output folders.

Now that the J2EE project has been created, you can create the first component for the application—the entity bean. To do so, right-click the project in the Package Explorer, and select New, Other, then select Entity Bean from the EJB Components category under JBoss-IDE (see Figure 2.6).

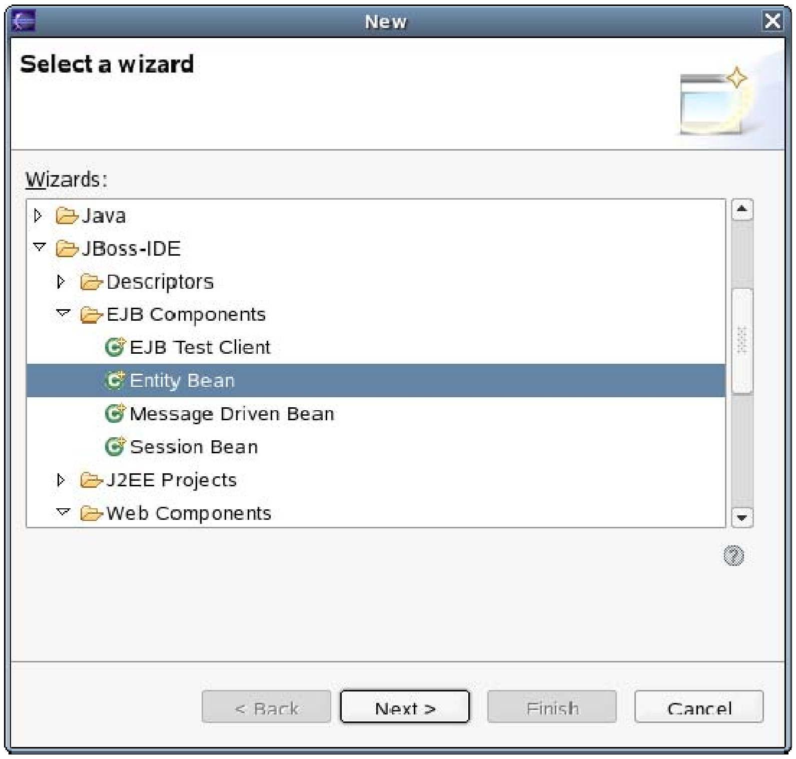
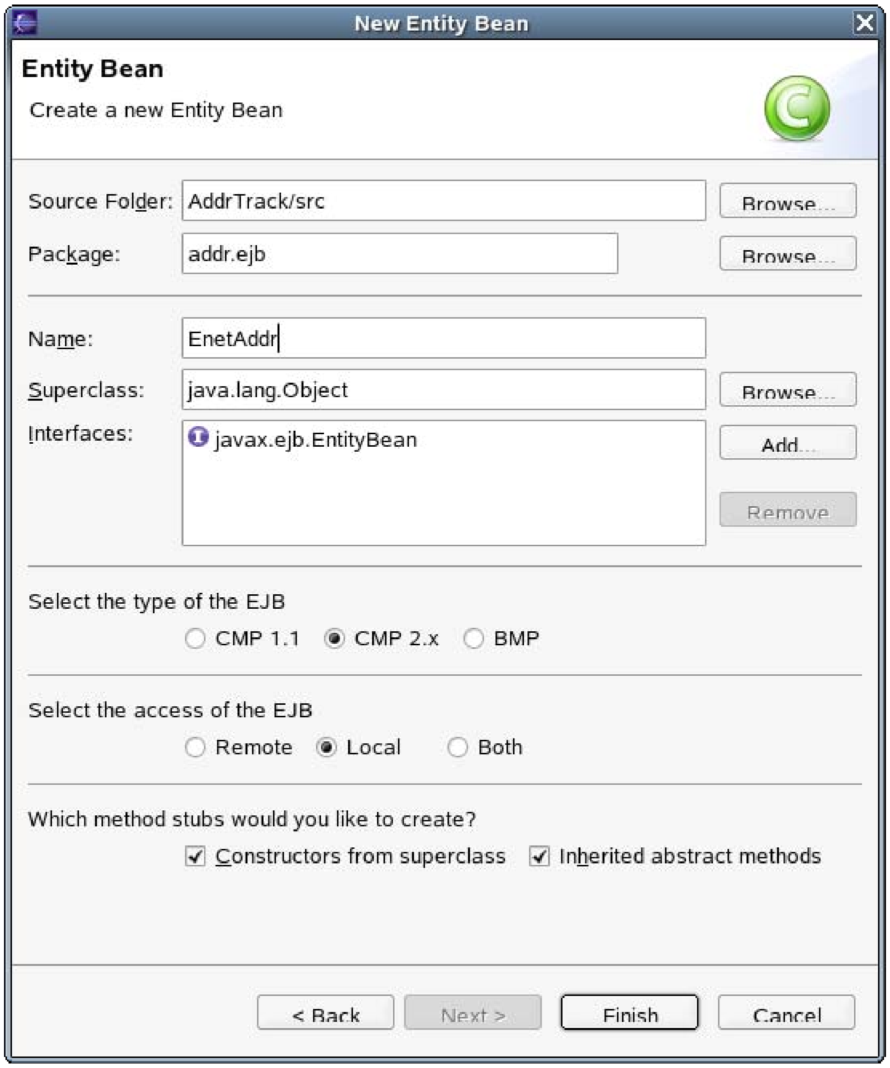


Figure 2.6: JBoss-IDE entity bean wizard.

As Figure 2.7 shows, enter the bean name and package name in the New Entity Bean dialog box, then click Finish to create the bean.



Entity Bean

Create a new Entity Bean

Source Folder:	AddrTrack/src	Browse...
Package:	addr.ejb	Browse...
Name:	EnetAddr	
Superclass:	java.lang.Object	Browse...
Interfaces:	javax.ejb.EntityBean	Add... / Remove

Select the type of the EJB

CMP 1.1 [✓ CMP 2.x] BMP

Select the access of the EJB

Remote [✓ Local] Both

Which method stubs would you like to create?

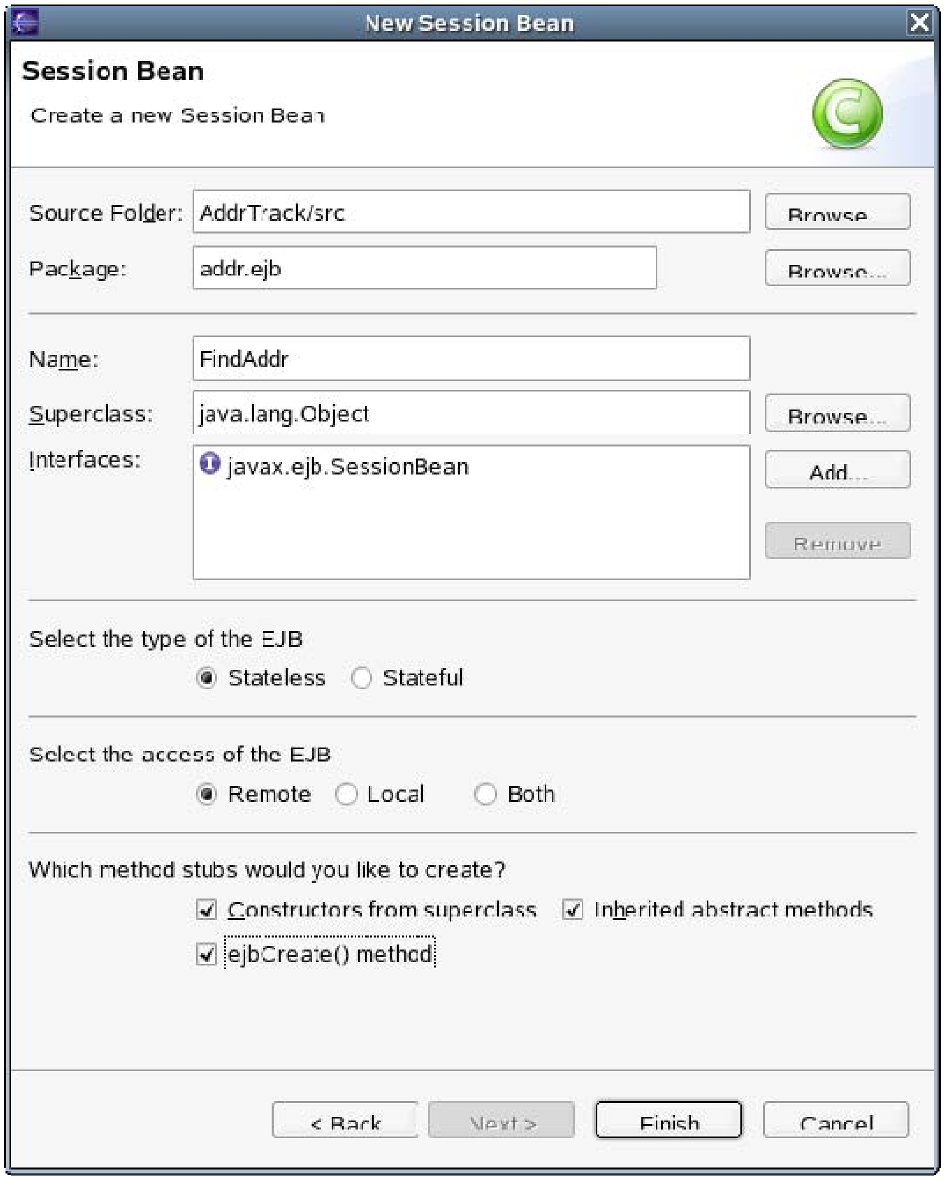
☑ Constructors from superclass ☑ Inherited abstract methods

Figure 2.7: The New Entity Bean dialog box.

One of the main reasons for using an entity bean is data persistence. You can choose between Container Managed Persistence (CMP) or Bean Managed Persistence (BMP). As Figure 2.7 shows, you will use CMP 2.x for this example application.

Creating the Session Bean

The session bean will contain the program logic of the application. It interacts with the UI to take requests and send responses. It also interacts with the entity bean to retrieve data and record new data. To create the session bean, right-click the project in the Package Explorer, and select New, Other as you did before with the entity bean. This time, pick Session Bean from the list, then click Next to proceed to the New Session Bean dialog box, which Figure 2.8 shows.



Source Folder: AddrTrack/src
Package: addr.ejb
Name: FindAddr
Superclass: java.lang.Object
Interfaces: javax.ejb.SessionBean

Select the type of the EJB
[✓ Stateless] Stateful

Select the access of the EJB
[✓ Remote] Local Both

Which method stubs would you like to create?
[✓ Constructors from superclass] [✓ Inherited abstract methods]
[✓ ejbCreate() method]

Figure 2.8: The New Session Bean dialog box.

The session bean will go in the same package as the entity bean, so enter addr.ejb (or select it from Browse) in the Package text box. The session bean name will be FindAddr; enter that in the Name text box. You will want a stateless bean with remote access, as those are the default selections. You also want to create a stub for the ejbCreate() method, so select that option near the bottom of the dialog box. Click Finish, and the bean will be created.

The main intention for your session bean is to respond to a request from the UI for the Ethernet hardware address that corresponds to an IP address. To accomplish this goal, you will add a method to the session bean called hardAddr. To do so, right-click the FindAddr bean class in Package Explorer, and select J2EE, Add Business Method as Figure 2.9 shows.

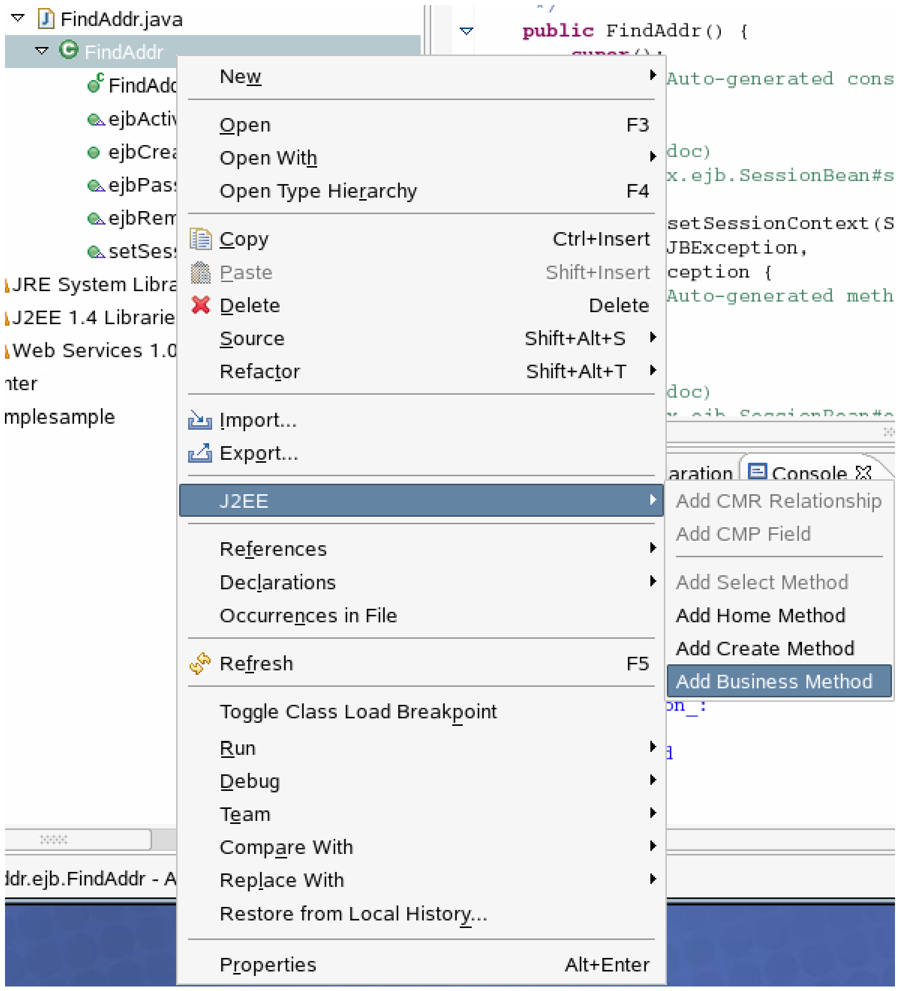


Figure 2.9: Adding a business method.

Enter the method name, hardAddr, in the name box, and enter the return type, which is String. The method will take a parameter that is the IP address to look up, and add a parameter called ipAddr of type String in the parameters section. Click Finish to create the method stub.

Next, you need to write the code for the application logic that finds the Ethernet hardware address for the IP address passed as a parameter to the method. There are a number of ways to do so. For this example, you will simply read from the file /proc/net/arp. You will also check the entity bean to determine whether a value has been recorded previously for this address. An obvious extension to the application would be to store a history of IP addresses for hardware addresses in the entity bean. This addition would be a straightforward exercise to augment the entity bean and add logic to the session bean.

XDoclet

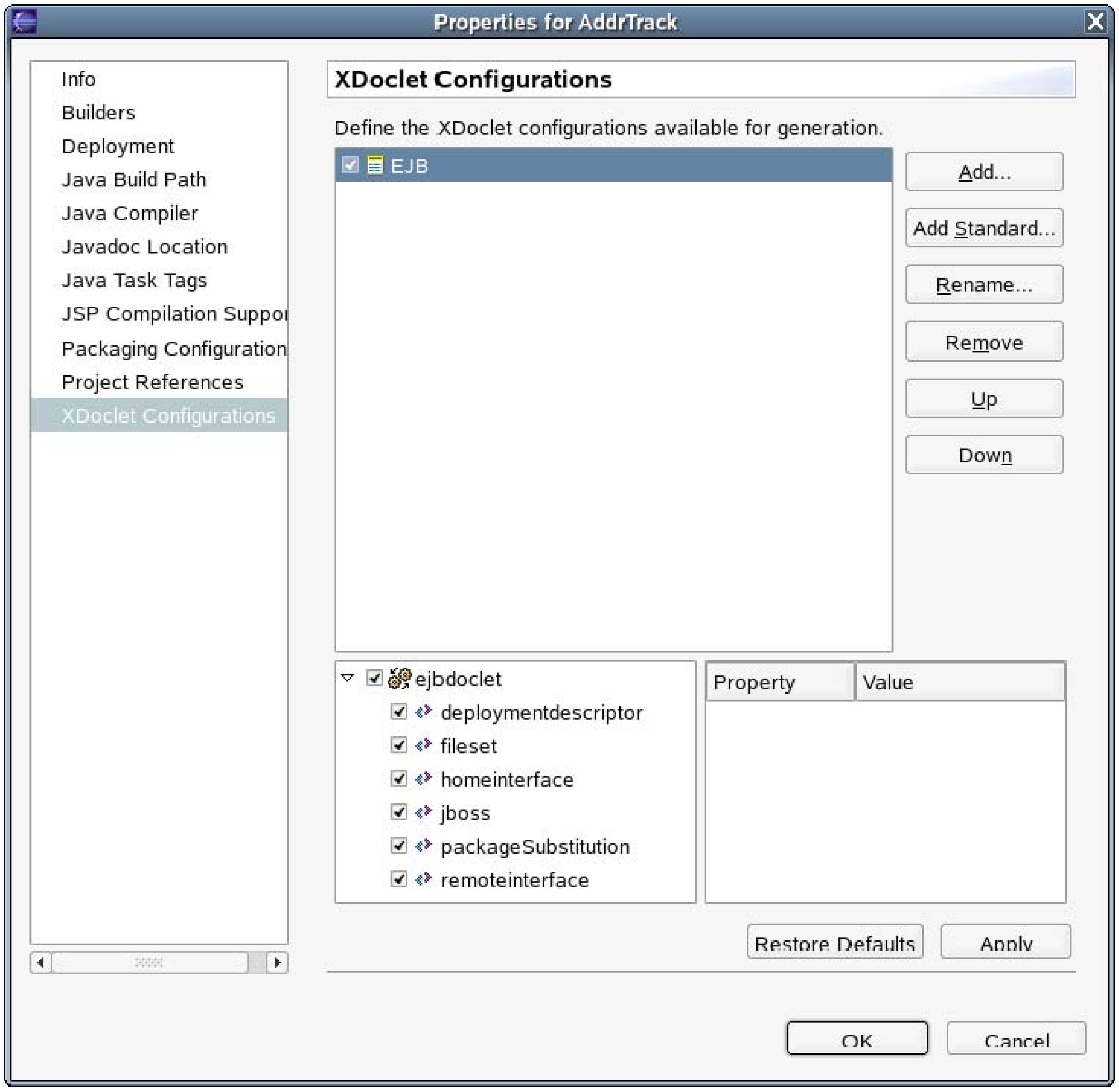
Now that you have created two EJB components; an entity bean and session bean, you need code files for the interfaces and XML files for descriptors. Instead of meeting this need by hand, you can use the XDoclet capability built-in to the JBoss Eclipse IDE plug-in. XDoclet is an Open Source project for code generation in Java development. JavaDoc tags are used in your source to define attributes for your component (for example, bean), and the XDoclet engine parses the code file with the tags and generates appropriate code and descriptors. When using XDoclet in JBoss Eclipse IDE, you need to define your XDoclet configuration and then run the XDoclet engine.

The XDoclet configuration page is on the property pages for the J2EE project—right-click the project in Package Explorer, and select Properties. Select XDoclet Configurations from the category list on the left side of the Properties dialog box (see Figure 2.10). Add a new configuration for your beans, and give the configuration a name; EJB, for example, as you are working on JavaBean components. You can either build a custom configuration or select Standard EJB from the list of built-in configurations. Table 2.1 shows the items and settings that are of interest for the beans in this example. All of these items are included in Standard EJB, so you can save some time by using that configuration.

Item	**Property**	**Value**
ejbdoclet	destDir	src
	ejbSpec	2.0
fileset	Dir	src
	includes	**/*.java
jboss	destDir	src/META-INF
	Version	4.0
	datasource	java:DefaultDS
	datasourceMapping	Hypersonic SQL
deploymentdescriptor	destDir	src/META-INF
packageSubstitution	packages	ejb
	substituteWith	interfaces
remoteinterface	No settings	
homeinterface	nosettings	

Table 2.1: XDoclet configuration, properties, and values.

The XDoclet Configuration property page will look something like Figure 2.10 if you set up a custom configuration.



Properties for AddrTrack

Info
Builders
Deployment
Java Build Path
Java Compiler
Javadoc Location
Java Task Tags
JSP Compilation Suppor
Packaging Configuration
Project References
XDoclet Configurations

XDoclet Configurations

Define the XDoclet configurations available for generation.

☑ EJB

☑ ejbdoclet
☑ deploymentdescriptor
☑ fileset
☑ homeinterface
☑ jboss
☑ packageSubstitution
☑ remoteinterface

Property	Value

Figure 2.10: An example XDoclet configuration.

The JavaDoc tags for EJB are already in the code files that the bean wizards create. Look near the top of FindAddr.java, for example, and you sill see the @ejb.bean tag. Listing 2.1 shows the tag with edited fields. The business method stub for hardAddr is added to the session bean file FindAddr.java. To complete the method, you need to add code to find the hardware address. Listing 2.1 shows the code that implements the approach of reading the /proc/net/arp file.

```
/**
 * @ejb.bean name="FindAddr"
 *           display-name="Find Address"
 *           description="Finds corresponding Ethernet hardware
 *                        address for given IP address"
 *           jndi-name="ejb/FindAddr"
 *           type="Stateless"
 *           view-type="remote"
 */

private EnetAddrLocalHome home;
```

```
/**
 * Business method
 * @ejb.interface-method  view-type = "remote"
 */
public String hardaddr(String ipAddr) {
      if (ipAddr == null) {
            throw new EJBException("Argument should not be blank");
      }

// first look in the /proc/net/arp file
      String result = "Not Found";
      try {
           BufferedReader arpin = new BufferedReader(new
                                         FileReader("/proc/net/arp"));

           String inline;

           while ((inline = arpin.readLine()) != null) {
            if (inline.startsWith(ipAddr)) {
                  result = inline.split("     ")[3];
            }
          }

          arpin.close();
          } catch (IOException e) {
            System.err.println
              ("File read error on /proc/net/arp " + e);
          }

// check EnetAddr entity if we didn't find it in /proc/net/arp
        if (result == "Not Found") {
                  try {
                        Context context = new InitialContext();
                        home = (EnetAddrLocalHome)
                        context.lookup("java:/comp/env/ejb/EnetAddr");

                  } catch (NamingException e) {
                        throw new EJBException(e);
                  }

                  try {
                        EnetAddrLocal enetAddr;

                        enetAddr = home.findByPrimaryKey(ipAddr);
                        if (enetAddr.getHardAddr()!=null)
                              result = enetAddr.getHardAddr();
                  } catch (FinderException e) {
                        throw new EJBException(e);
                  }
        }
      return result;
}
```

Listing 2.1: The XDoclet tag and code for the FindAddr session bean.

The entity bean skeleton file EnetAddr.java has XDoclet tags for its bean and persistence properties. The tags need to be edited and augmented and code added to identify the persisted data. Edit the XDoclet tags in EnetAddr.java as Listing 2.2 shows.

```
/**
 * @ejb.bean name="EnetAddr"
 *           display-name="Ethernet Addresses"
 *           description="Records Ethernet hardware addresses"
 *           jndi-name="ejb/EnetAddr"
 *           type="CMP"
 *           cmp-version="2.x"
 *           view-type="local"
 *           primkey-field = "ipAddr"
 * @ejb.persistence table-name = "enetAddress"
 */

      /**
       * @ejb.pk-field
       * @ejb.persistent-field
       * @ejb.persistence
       *     column-name="ipAddr"
       *     sql-type="VARCHAR"
       */
      public abstract String getIpAddr();

      /**
       * @ejb.persistent-field
       * @ejb.persistence
       *     column-name="hardAddr"
       *     sql-type="VARCHAR"
       * @ejb.interface-method view-type = “local”
       */
      public abstract String getHardAddr();
      public abstract void setHardAddr(String str);
```

Listing 2.2 XDoclet for entity bean and fields.

Next, you can run the XDoclet engine. To do so, right-click the project, and select Run XDoclet (see Figure 2.11).

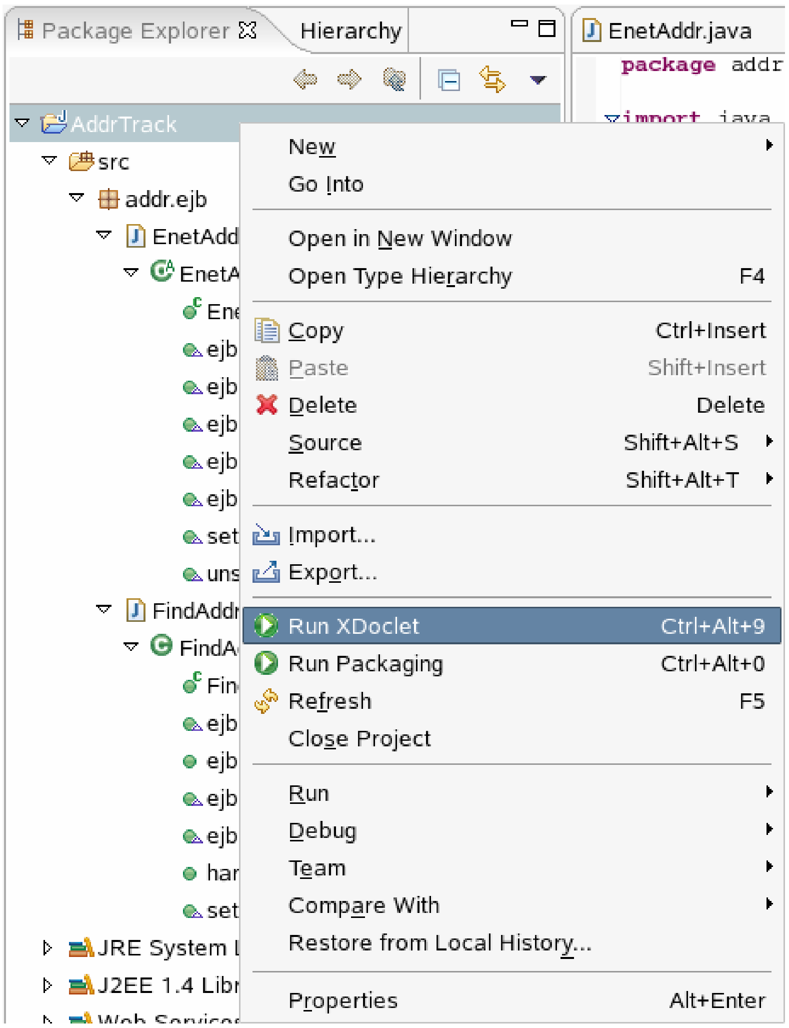


Figure 2.11: Running the XDoclet Engine.

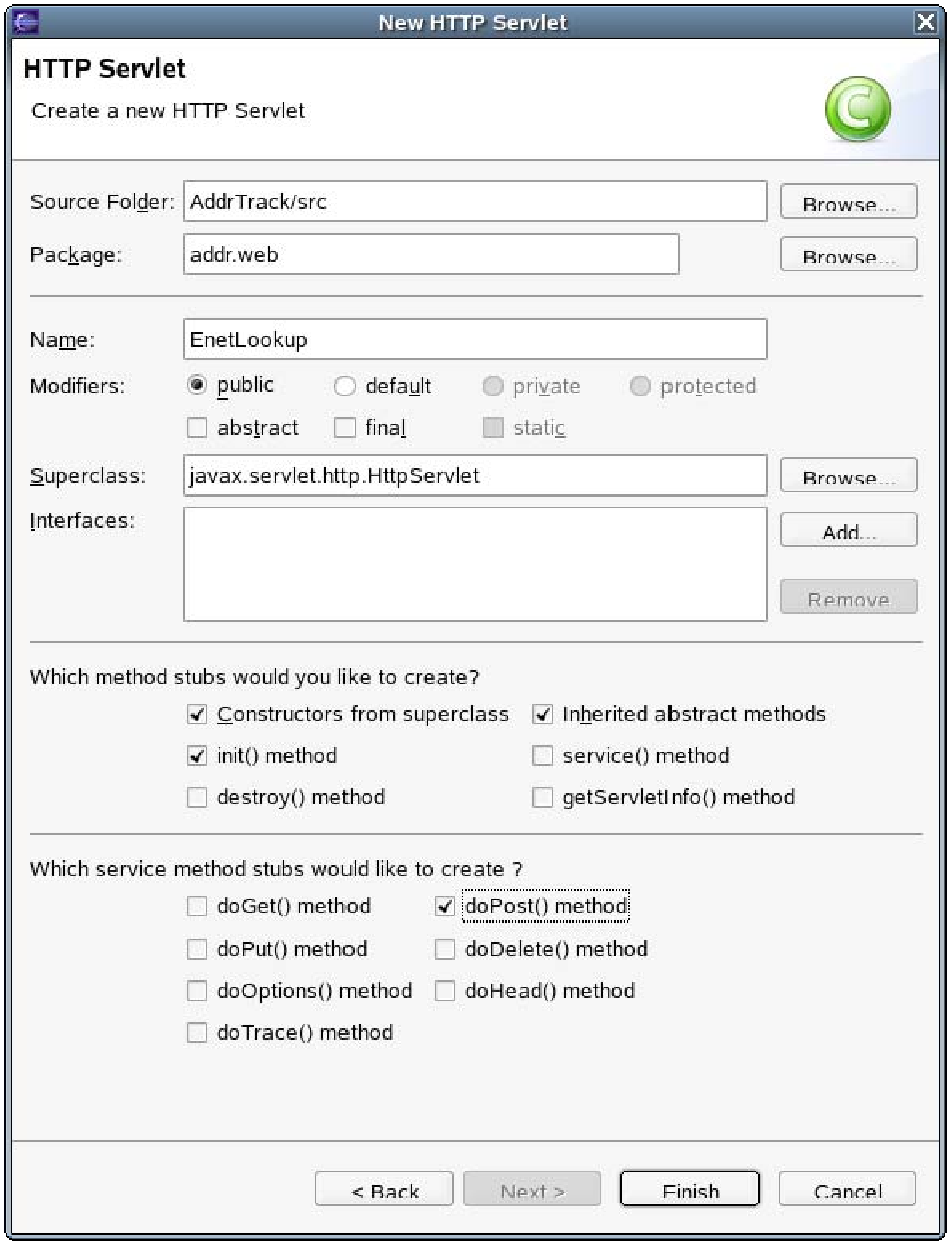
You will see a new package created, addr.interfaces, that contains the code files for the bean interfaces. You will also see a new folder, META-INF, that contains XML configuration files. The version of JBoss Eclipse IDE that this example is using, version 1.4.1, does not generate EJB 2.1 descriptors, so you will see EJB 2.0 DTD instead of EJB 2.1 XSD.

At the time of writing, JBoss Eclipse IDE 1.5M2 was released, which supports EJB 3.0 and Eclipse 3.1.

Thanks to XDoclet, you have your complete beans. The next step is to create the pieces of the UI.

Creating the UI

This example process will use a servlet for the Web-based UI. The servlet communicates with the session bean to get data and creates dynamic HTML to send results back to the user. To create the servlet, right-click the project in Package Explorer, select New, Other, and pick HTTP Servlet from the Web Components group under JBoss-IDE. Enter addr.web for the package and Lookup for the name. The New HTTP Servlet dialog box will look like the example that Figure 2.12 shows.



Source Folder: AddrTrack/src
Package: addr.web
Name: EnetLookup
Modifiers: [✓ public] default private protected
abstract final static
Superclass: javax.servlet.http.HttpServlet
Interfaces:

Which method stubs would you like to create?
[✓ Constructors from superclass] [✓ Inherited abstract methods]
[✓ init() method] service() method
destroy() method getServletInfo() method

Which service method stubs would like to create ?
doGet() method [✓ doPost() method]
doPut() method doDelete() method
doOptions() method doHead() method
doTrace() method

Figure 2.12: The New HTTP Servlet dialog box.

Click Finish, and the servlet .java code file will be created in the new addr.web package. Next, you need to add the code for the doPost method and the init method. The doPost method takes a request posted by the user, finds and connects to the session bean FindAddr, and calls the hardAddr method of FindAddr, passing the IP address string as a parameter. The servlet then writes an HTML response to the user, incorporating the answer returned from the session bean. Listing 2.3 shows the doPost code that accomplishes these tasks.

```
protected void doPost(
    HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

response.setContentType("text/html");
PrintWriter htout = response.getWriter();

htout.println("<html><head><title>");
htout.println("Ethernet Hardware Address Lookup");
htout.println("</title></head>");
htout.println("<body>");
htout.println("<h1>");
htout.println("Ethernet Hardware Address Lookup");
htout.println("</h1>");

try {
      FindAddr bean = home.create();

      String ipAddr = request.getParameter("ipAddr");
      String enetAddr = bean.hardAddr(ipAddr);
      bean.remove();

      htout.println("<p>");
      htout.println("The Ethernet hardware address for ");
      htout.println(ipAddr);
      htout.println(" is ");
      htout.println(enetAddr);
      htout.println("</p>");
    } catch (Exception e) {
      htout.println(e.getMessage());
      e.printStackTrace(htout);
    } finally {
      htout.println
         (“<p><a href=\”/EnetAddr\”>Return to Lookup</a></p>”);
      htout.println("</body></html>");
      htout.close();
    }
}
```

Listing 2.3: The doPost method.

The code for the servlet init method simply finds and connects the FindAddr session bean. Listing 2.4 shows this method.

```
private FindAddrHome home;

public void init(ServletConfig config) throws ServletException {
    super.init(config);
    try {
          Context context = new InitialContext();
          Object ref = context.lookup("java:/comp/env/ejb/FindAddr");
          home = (FindAddrHome) PortableRemoteObject.narrow(
                                  ref, FindAddrHome.class);
        } catch (Exception e) {
          throw new ServletException("Could not find java:/comp/env/");
        }
}
```

Listing 2.4: The init method.

Last, to finish the servlet code, you need to edit the XDoclet tags entered from the servlet wizard template. The tags are near the top of the EnetLookup.java code file for the servlet. Edit the tags as Listing 2.5 shows.

```
/**
 * Servlet Class
 *
* @web.servlet              name="EnetLookup"
 *                           display-name="Ethernet Address Lookup"
 *                           description="Servlet to lookup Ethernet
 *                                        hardware address"
 * @web.servlet-mapping      url-pattern="/EnetLookup"
 * @web.ejb-ref              name="ejb/FindAddr"
 *                           type="Session"
 *                           home="addr.interfaces.FindAddrHome"
 *                           remote="addr.interfaces.FindAddr"
 *                           description="Reference for FindAddr EJB"
 * @jboss.ejb-ref-jndi       ref-name="ejb/FindAddr"
 *                           jndi-name="ejb/FindAddr"
 */
```

Listing 2.5: XDoclet tags for servlet.

Once the XDoclet tags are set, you need an XDoclet configuration for servlet tags. There is a built-in configuration called Standard Web (as there is Standard EJB for JavaBeans) that you can use. You can also create a custom configuration that includes the relevant items and properties. Table 2.2 shows the properties and values needed.

Item	Property	Value
webdoclet	destDir	scr/WEB-INF
fileset	dir	src
	includes	**/*.java
jbosswebxml	destDir	src/WEB-INF
	Version	4.0
deploymentdescriptor	destDir	src/WEB-INF
	Servletspec	2.3

Table 2.2: XDoclet configuration for servlets.

Run the XDoclet engine to create the WEB-INF folder and the XML descriptor files. Doing so will complete the creation of the servlet part of the UI. The second part you need is the HTML page the application presents the user. This example will use a simple page with HTML (see Listing 2.6). First, create a folder called appstart by right-clicking the project, and selecting New, Folder from the menu. Enter appstart for the folder name. To create the HTML page, right-click the folder appstart, and select New, Other HTML Page in the Web Components category under JBoss-IDE. Enter index.html for the file name of the page.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
   <title>Ethernet Address Lookup</title>
</head>
<body bgcolor="#FFFFFF">
   <h1>Ethernet Address Lookup</h1>
      <form action="EnetLookup" method="POST" >
         <table cellspacing="2" cellpadding="2" border="0">
            <tr>
               <td>
                  IP Address :
               </td>
               <td>
                  <input type="text" name="ipAddr" value="0.0.0.0"/>
               </td>
            </tr>
            <tr>
               <td>
                  <input type="submit" name="Lookup" value="Lookup"/>
               </td>
               <td>
                  <input type="Reset"/>
               </td>
            </tr>
         </table>
      </form>
</body>
</html>
```

Listing 2.6: The Application Start Page—index.html.

That completes the UI and all the components of the application. Next, you need to set up the overall packaging for the complete application, which will end up in an EAR.

The Application Package

The tasks involved in configuring the application package are:

- Defining the component packages for EJB components and Web components
- Defining the total application package (Enterprise Archive)
- Defining the deployment descriptor for the application

The deployment descriptor is application.xml and goes in the META-INF folder. To create the file, right-click the META-INF folder, and select New, Other, and pick EAR 1.4 Deployment Descriptor from the Descriptors category under JBoss-IDE. The default name will be application.xml. Click Finish to create the file.

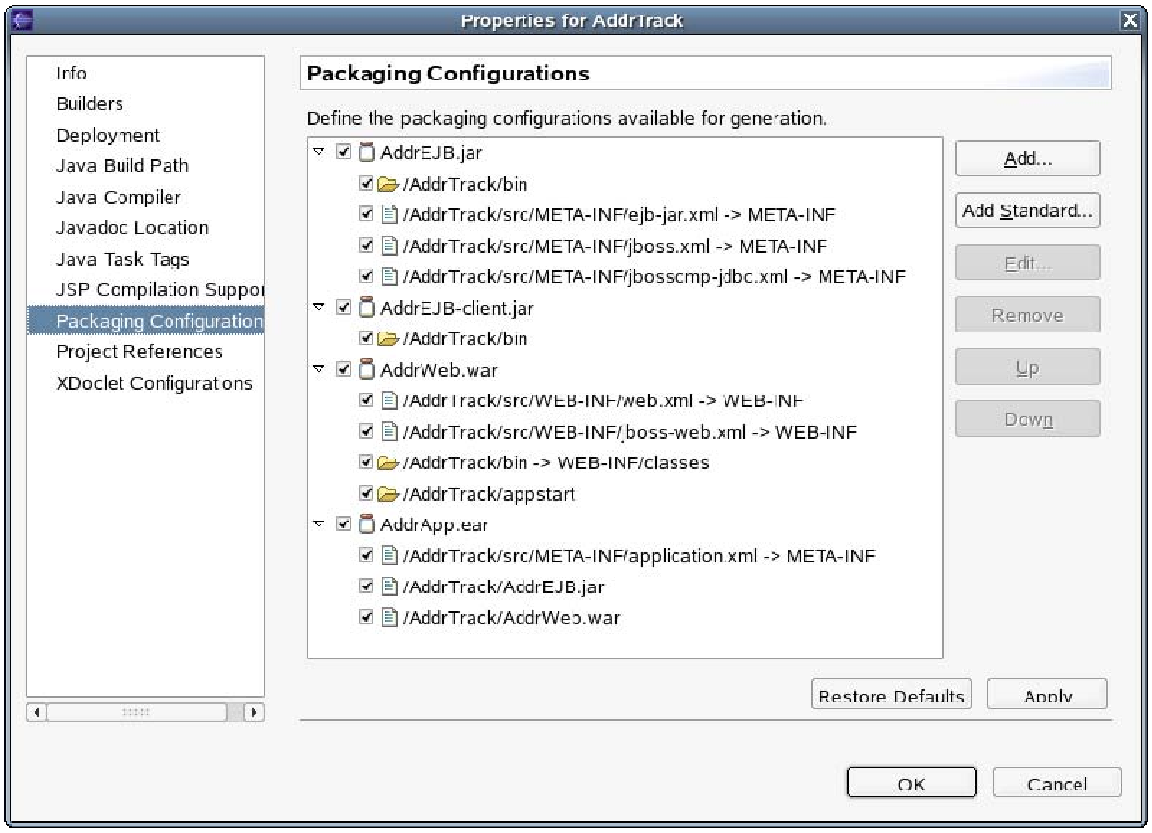
Next, the EAR Deployment Descriptor template needs to be edited. To do so, double-click application.xml to open it, and edit so that it appears as the sample in Listing 2.7 looks.

```
<?xml version="1.0" encoding="UTF-8"?>
<application version="1.4"
      xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
      http://java.sun.com/xml/ns/j2ee/application_1_4.xsd">

      <display-name>Ethernet Address Application</display-name>
      <module>
            <ejb>AddrEJB.jar</ejb>
      </module>
      <module>
            <web>
                  <web-uri>AddrWeb.war</web-uri>
                  <context-root>/EnetAddr</context-root>
            </web>
      </module>
</application>
```

Listing 2.7: The application.xml Deployment Descriptor.

Next, you need to define the sub-packages for the EJB components and the Web components. These will be Java Archive (JAR) files that contain the compiled code .class files and Web Archive (WAR) that contain Web pieces. The definitions for these packages are created in the Packaging Configurations page of the project properties. Open the project properties, and select the Packaging Configurations, as Figure 2.13 illustrates.

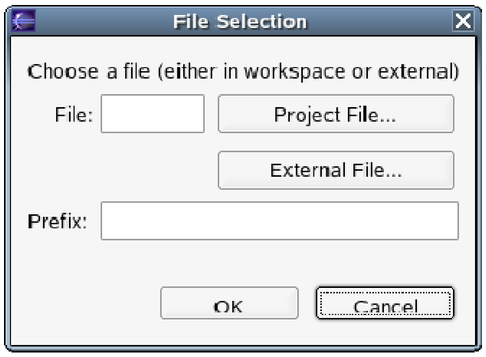


Define the packaging configurations available for generation.

```
AddrEJB.jar
    /AddrTrack/bin
    /AddrTrack/src/META-INF/ejb-jar.xml -> META-INF
    /AddrTrack/src/META-INF/jboss.xml -> META-INF
    /AddrTrack/src/META-INF/jbosscmp-jdbc.xml -> META-INF
AddrEJB-client.jar
    /AddrTrack/bin
AddrWeb.war
    /AddrTrack/src/WEB-INF/web.xml -> WEB-INF
    /AddrTrack/src/WEB-INF/jboss-web.xml -> WEB-INF
    /AddrTrack/bin -> WEB-INF/classes
    /AddrTrack/appstart
AddrApp.ear
    /AddrTrack/src/META-INF/application.xml -> META-INF
    /AddrTrack/AddrEJB.jar
    /AddrTrack/AddrWeb.war
```

Figure 2.13: Project packaging configurations.

Create the AddrEJB.jar package definition by selecting Add, and entering the name AddrEJB.jar. Add the four items by right-clicking and selecting Add Folder or Add File as appropriate. Use the entries in Table 2.3 to enter values in the File Selection and Folder Selection dialog boxes that Figure 2.14 shows.



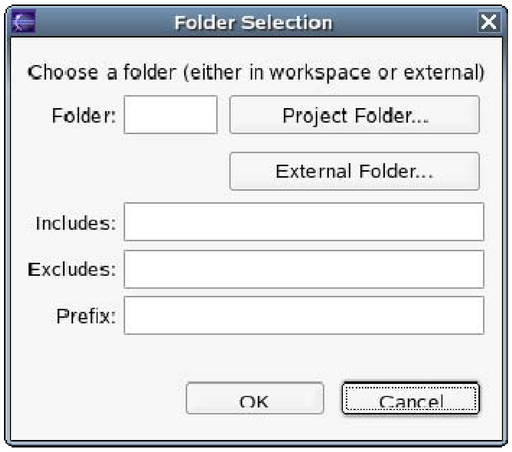


Figure 2.14: The File Selection and Folder Selection dialog boxes.

File/Folder Name	Prefix	Includes	Excludes
/AddrTrack/bin		addr/ejb/*.class,addr/interfaces/*.class	
/AddrTrack/src/META-INF/ejb-jar.xml	META-INF		
/AddrTrack/src/META-INF/jboss.xml	META-INF		

Table 2.3:File Selection and Folder Selection settings.

Next create the AddrEJB-client.jar package definition using the same method and the values in Table 2.4

File/Folder Name	Prefix	Includes	Excludes
/AddrTrack/bin		addr/interfaces/*.class	

Table 2.4: The AddrEJB-client.jar package definition values.

Then create the AddrWeb.war package definition using the same method and the values in Table 2.5

File/Folder Name	Prefix	Includes	Excludes
/AddrTrack/src/WEB-INF/web.xml	WEB-INF		
/AddrTrack/src/WEB-INF/jboss-web.xml	WEB-INF		
/AddrTrack/bin	WEB-INF/classes	addr/web/*.class	
/AddrTrack/appstart			

Table 2.5: The AddrWeb.war package definition values.

Finally, create the AddrApp.ear package definition using the values in Table 2.6.

File/Folder Name	Prefix	Includes	Excludes
/AddrTrack/src/META-INF/application.xml	META-INF		
/AddrTrack/AddrEJB.jar			
/AddrTrack/AddrWeb.war			

Table 2.6: The AddrApp.ear package definition values.

Once all the package definitions are created, build the packages by running the packaging engine. To do so, right-click the project, and select Run Packaging. The .jar, .war. and .ear files are created and appear in the Package Explorer. As mentioned earlier, the EAR file is the complete application. It can be dropped in the deploy folder of a JBoss application server, and the server will install and start the application.

Chapter 3 will provide more detail about deploying and debugging applications.

Summary

This chapter has explored how Open Source development tools such as Eclipse, JBoss Eclipse IDE, and XDoclet make it quick and efficient for developers to write J2EE applications. The application developer is able to focus on writing the application-specific code for the components, and the tools generate and manage the EJB “plumbing”—the interfaces, deployment descriptors, and configurations. The release of the EJB 3.0 specification brings significant changes to the EJB structure of J2EE applications. JBoss already provides support for the preliminary specification, and the process of using JBoss Eclipse IDE to write EJB 3.0 applications will be essentially the same.

The next chapter will look at the JBoss Application Server in detail. It will explore the internal design of JBoss along with administration and operation topics. It will then return to the example application to look at what is involved in deploying, running, and debugging a J2EE application in this environment.

[**Editor’s Note:** This content was excerpted from the free eBook *The Developer Shortcut Guide to SUSE Linux* (Realtimepublishers) written by John Featherly and available at http://cc.realtimepublishers.com/portal.aspx?pubid=339.]