# CHAPTER 6

## *Organizing Your Development Project*

*All right, guys! It's time to clean up this town!*

—Homer Simpson

In this book we describe how to build applications that are defined by the J2EE specification. When you build an application, you create one or more projects that correspond to J2EE modules. You also use these same projects to organize your development work; that is, you use these projects

- to manage the source code and files that make up the application,
- to divide the work between the teams, and
- to set up an automated process that builds the application, runs tests, and creates project reports.

This chapter starts with a basic description of the types of applications and projects that are supported in WTP. We will show you how to create different kinds of projects to build applications.

In the second part of the chapter, we will describe some of the advanced project features that are available with WTP. There is very little available in terms of standards to guide you in the organization of project artifacts and source code for Web projects. Project best practices achieve a balance between the concerns that drive a particular development project:

- How many teams and developers are there?
- What are the subsystems?
- What components are tested, and how are they tested?
- Who builds the code?

- How is it integrated?
- How is it released?

Naturally, each concern is a different dimension of the project. We will use advanced WTP features to create project templates and apply best practices that are helpful to organize your development work. We use the generic term *Web project* to describe the WTP project types that are provided for J2EE development.

## Web Project Types and J2EE Applications

A project is used to develop modules such as J2EE Web applications and EJBs. Typically, each module is a project, but this is not a strict requirement (see Figure 6.1).

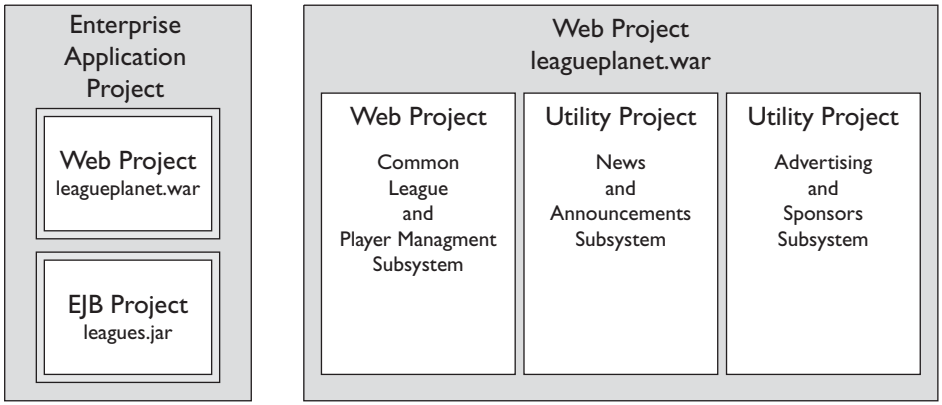


**Figure 6.1** J2EE Applications and Web Projects

For example, in a complete J2EE enterprise application, one project might consist of a Web application module for the presentation logic while another would be used to develop the EJB module for the business components. In this case, the complete application consists of three projects for the modules: one for the enterprise application, one for the Web application, and one for the EJBs. It is also possible to split the development of a single module into multiple projects. For example, a basic module like a Web application might be built from utility modules built in other projects. You will learn how to organize your projects and modules using similar patterns later in this chapter.

## Web Projects

Projects organize your source code and modules. WTP provides Web projects that are sophisticated Eclipse projects that know about J2EE artifacts. In addition to having basic Java project capabilities, a Web project can be used to organize J2EE artifacts into buildable, reusable units (see Figure 6.2).

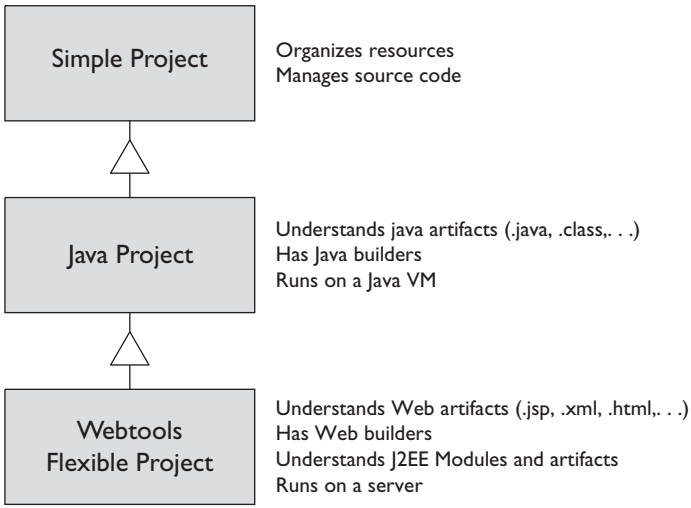


**Figure 6.2** Web Projects

An Eclipse *simple project* (or general project) provides the basic infrastructure to organize and build resources. The structure of a general project is very open; resources such as files and directories can be organized in any arbitrary form that makes sense for a particular purpose.

A JDT *Java project* contains Java elements such as packages, types, methods, fields, and property files for creating Java programs. A Java project knows how to build and run Java programs. Each Java project has a Java builder that can incrementally compile Java source files as they are edited.

You can change the properties of a Java project, such as the Java build path. The build path is the classpath that is used for building the project. There are alternative ways of structuring the sources in a Java project; examples include using a single source folder that is the project root or multiple source folders for organizing complex Java projects.

A WTP Web project has more than just Java code. It contains sources that are used to build Web applications, EJBs, and enterprise applications. A Web application can be as simple as a bunch of HTML files, or it can have servlets,

JSPs, tag libraries, and Web services. These artifacts make the Web application. A Web project knows how to build, publish, and run J2EE modules and artifacts on application servers.

Web projects have builders, validators, and code generators. Builders produce standard publishable modules from complex development layouts. Validators help identify and catch coding errors at development time. J2EE validators are very valuable, because the sooner you find a problem the easier it is to fix. In J2EE, there are many deployment descriptors that have references to Java code and each other. These are interrelated in complex ways. Failure to catch a problem at development time could lead to a runtime error that might be very difficult to diagnose and fix. Generators create components from annotations in source code (for example, using XDoclet or JSR 175).

### J2EE Modules

The output of the development activities are discrete J2EE components (EJBs, servlets, application clients), which are packaged with component-level deployment descriptors and assembled into J2EE modules. Web application modules, EJB modules, enterprise application modules, and Java 2 Connector Architecture (J2CA) resource modules are typical J2EE modules. A module contains code, resources, and deployment descriptors. A J2EE module forms a stand-alone unit, which can be deployed and run on a J2EE application server. Figure 6.3 provides an overview of the J2EE structure associated with common J2EE modules, such as Web, EJB, and EAR, as described by the specification.

### Creating Applications

WTP provides projects and wizards to help you get started quickly with different types of Web and J2EE applications. You can use these wizards to create most standard Web and J2EE artifacts. Additional tools will help you create, build, validate, and run your applications on servers.

To get started, we will review the steps involved in creating different types of applications. The simple steps provided in this section will help you acquire the skills you will need to work with the examples in this book. More specifically, you will learn how to create these types of projects:

- Dynamic Web project, where the output artifact is a WAR file
- EJB project, where the output artifact is an EJB JAR file
- EJB client project, where the output artifact is a JAR file that contains client-side classes for accessing an EJB module

- Enterprise application project, where the output artifact is an EAR file containing Web, EJB, and other modules

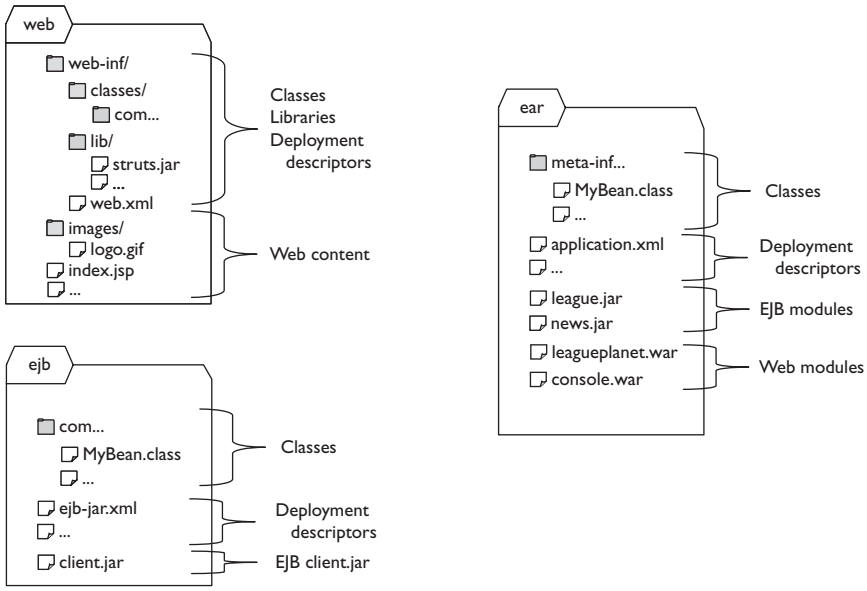


**Figure 6.3** J2EE Modules

### *Creating Web Applications*

To build a Web application you need a project that contains a Web module. There are two types of Web projects: static and dynamic.

Static Web projects contain resources that provide static content. You can use a static Web project to develop Web applications that contain many of the standard Web resources, such as HTML, images, CSS, and XML, and test them using a Web browser. These projects can be deployed to a conventional Web server, such as the Apache HTTP Server, that has no J2EE capabilities.

Dynamic Web projects are for J2EE Web applications that contain servlets, JSPs, and filters, in addition to static content. A dynamic Web project can be used as a stand-alone Web application, or it can be combined with other modules to create a J2EE enterprise application.

The J2EE specification defines a standard for Web application directory structure. It specifies the location of static Web files, JSPs, Java class files, Java libraries, deployment descriptors, and supporting metadata. The default dynamic Web project layout resembles the structure of a J2EE Web application

module. In the workbench, you can use the **New Web Project** wizard to create a new Web project. WTP has support for other types of project layouts and can automatically build a J2EE Web application archive (WAR) structure defined by the standard.

When you want to create a dynamic Web project, you will typically do the following:

1. Invoke the **Dynamic Web Project** wizard.
2. Provide parameters such as project name and locations for Web artifacts.
3. Choose a target runtime.
4. Choose project facets.

You can try these steps by repeating the following:

1. Switch to the **J2EE** perspective. In the **Project Explorer** view, right click, and invoke the **New ▸ Dynamic Web Project** menu item (see Figure 6.4).

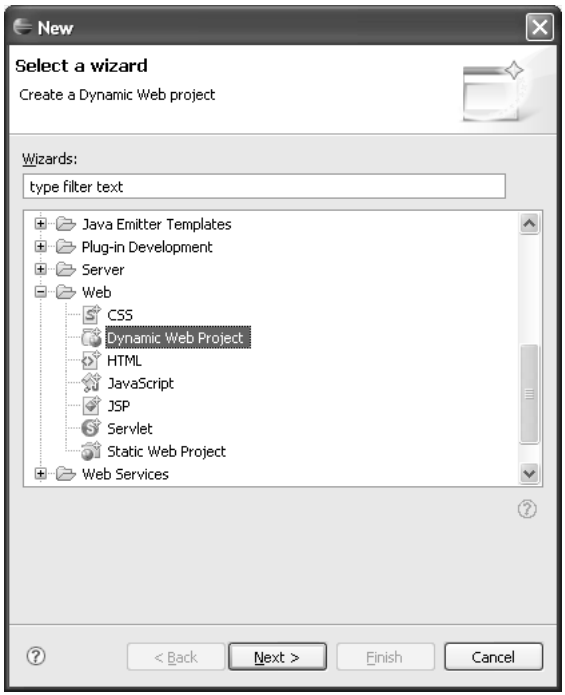


**Figure 6.4** Select Wizard

Click **Next**. The **New Dynamic Web Project** wizard opens (see Figure 6.5).

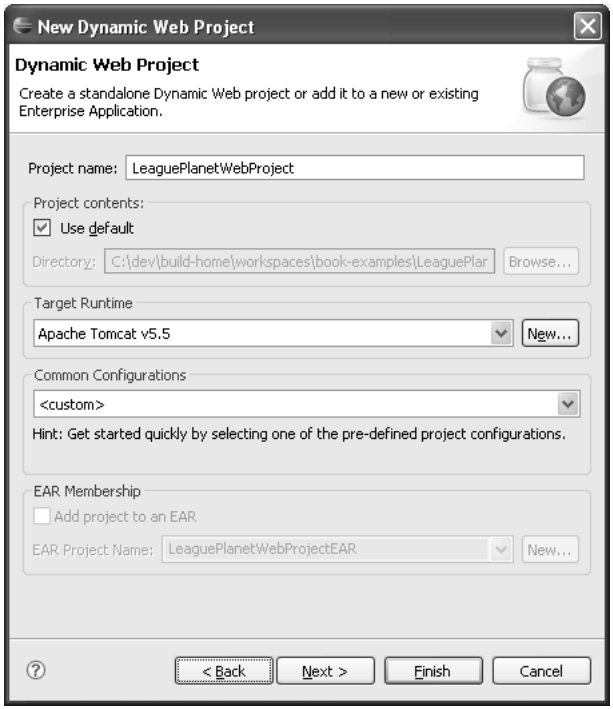


**Figure 6.5** New Dynamic Web Project

2. Enter `LeaguePlanetWebProject` for the project name. A dynamic Web project contains J2EE components such as JSPs and servlets. It is necessary for J2EE APIs to be a part of the project classpath. This is done for you automatically when you associate a J2EE server runtime with the project. The runtime provides a set of libraries that will also contain JARs such as the `servlet.jar`. If you switch the runtime at a later time, the classpath is also updated. If your prefer not to use a runtime to provide these libraries, you can create a folder that contains the J2EE libraries and point to it as your runtime library. However, this method will require you to obtain appropriate libraries for the J2EE APIs from

   ```
   http://java.sun.com
   ```

   Assuming you have defined a server runtime such as Tomcat, select it as the target runtime. We will revisit servers and runtimes in other chapters.

   Configurations allow you to choose a set of project facets for common styles of Web projects. For example, if you choose the WebDoclet configuration, WTP will set up the project to enable XDoclet.

Click the **Next** button. The **Project Facets** selection page is displayed (see Figure 6.6).

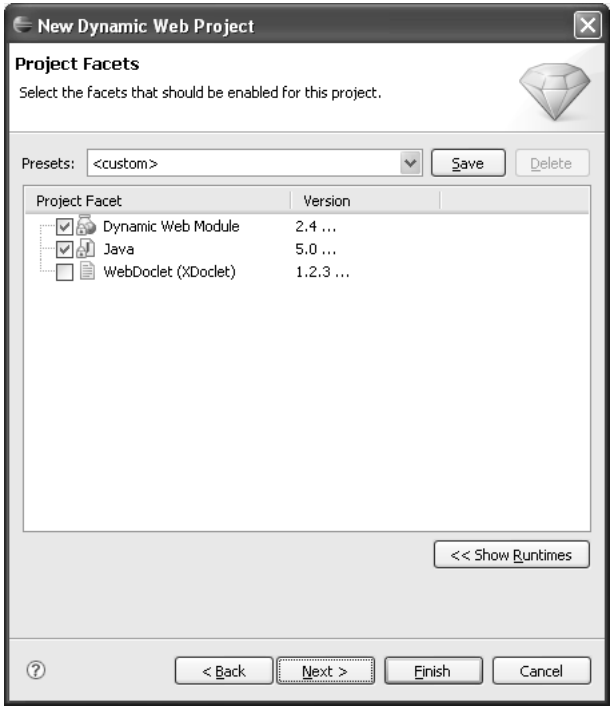


**Figure 6.6** Select Project Facets

3. A project facet describes some runtime aspect of the Web module. For Tomcat 5.0, you can specify the J2EE version, the Java version, and, optionally, the XDoclet version. Each server defines a set of supported facets and their allowed values. WTP configures the Web module and sets up the classpath for the project so that it matches the specified facets. Accept the defaults here and click the **Next** button. The **Web Module** page is displayed (see Figure 6.7).
4. The **Web Module** page lets you specify its context root name and the directories for its Web and Java resources. The context root is the name that appears in the URL for the Web application. Specify `LeaguePlanetWebProject` as the context root and accept the defaults for the directory names. Click **Finish**. WTP creates the project and populates it with configuration files such as the J2EE Web deployment descriptor, `web.xml` (see Figure 6.8).


**Figure 6.7** Web Module

You have now created a dynamic Web project named `LeaguePlanetWebProject` and targeted it to Tomcat.

The Dynamic Web Project wizard creates folders and files under the project (see Figure 6.9). Open the project you have just created and browse its contents. For example, the `WebContent` folder contains a special folder named `WEB-INF`, which holds items that are defined by the J2EE specification and are not accessible by a Web browser. The `WEB-INF/classes` folder is where compiled Java code goes. It also contains a special file, `web.xml`, which is the J2EE Web deployment descriptor.

The `WebContent` folder contains Web resources such as JSP and HTML files, and other types of supporting resources (see Figure 6.9). The contents of `WebContent` will be accessible from the Web application context root.

The following default elements are created with a dynamic Web project:

- `WebContent/WEB-INF/web.xml`: This is the Web deployment descriptor.
- `src`: This is the Java source code for classes, beans, and servlets. The publisher will copy the compiled class files into the `WEB-INF/classes` folder of the final application.

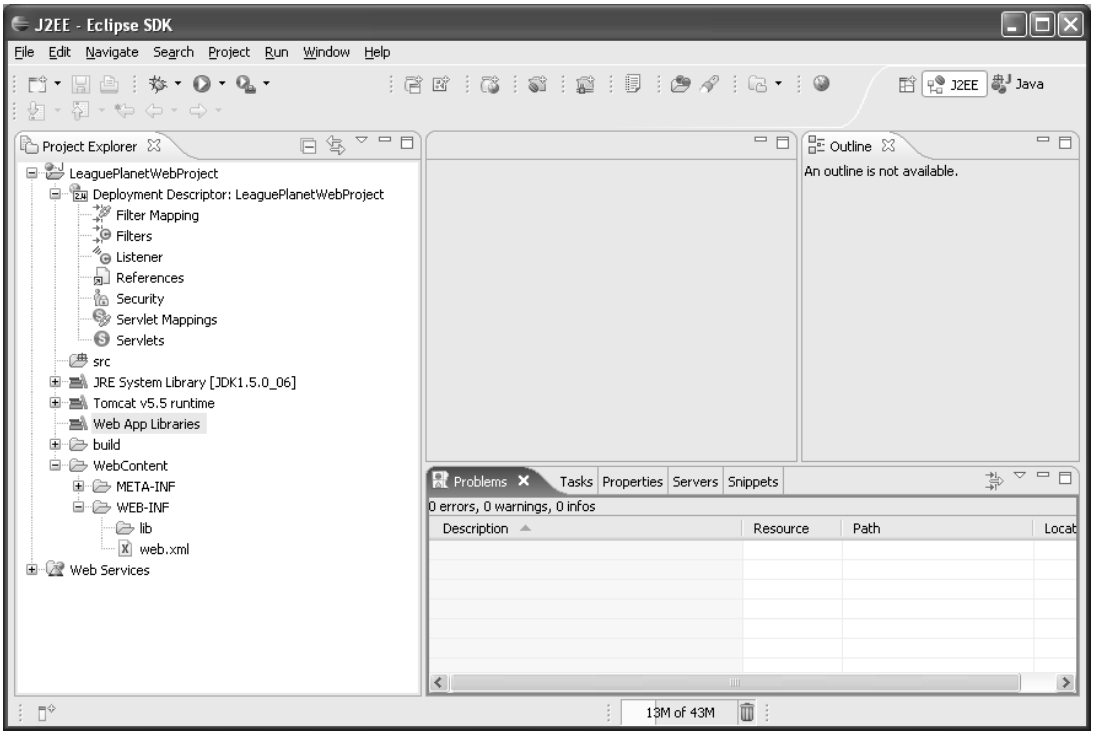


**Figure 6.8** Dynamic Web Project—LeaguePlanetWebProject

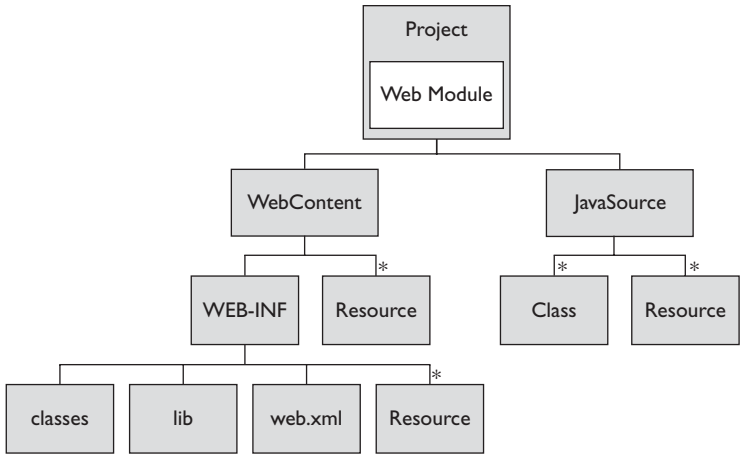


**Figure 6.9** Elements of a Dynamic Web Project

- `WebContent:` This is the Web application root. All Web artifacts placed in this folder will be available to the client. The publisher will copy the complete contents of this folder into the root of the final WAR file. It is possible to choose a different name for the `WebContent` folder or rename it.
- `WebContent/WEB-INF/classes:` Sometimes code and libraries will be delivered to you in the form of class files (in comparison to those that are provided to you as JAR files, which you would put into the `WEB-IF/lib` folder). To add them to the classpath of the final Web application, you can place them in this folder.
- `WebContent/WEB-INF/lib:` We will place all libraries that are provided to use in the form of JAR files here. They will be added to the build path of the project. The publisher will copy them into the WAR file, and they will be available to the class loader of the Web application.

A dynamic Web project can publish its contents as a Java Web application archive (WAR) file (see Figure 6.10). Publishers assemble the artifacts in a Web project, such as Java sources; Web content, such as JSPs, HTML, and images; and metadata, such as Web deployment descriptors, in a form that can run on a J2EE application server.

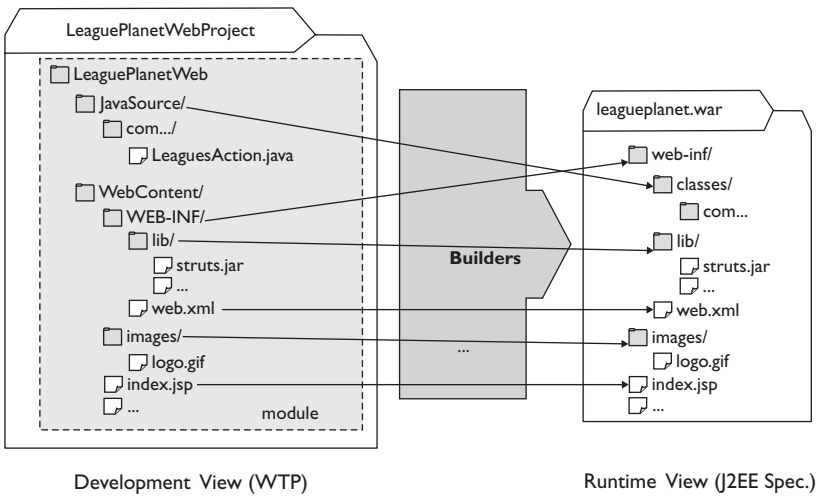


**Figure 6.10** Publisher

WTP wizards simplify the tasks involved in creating J2EE modules. We have just shown how to create a Web module. WTP online documentation at

```
www.eclipse.org/webtools
```

provides detailed information about these wizards and the project structure. The process of creating an EJB application is equally simple. The next section describes how to create an EJB project that contains an EJB module.

### *Creating EJB Applications*

An EJB project contains an EJB module. This project can be used to assemble one or more enterprise beans in a single deployable unit. EJBs are deployed in a standard Java archive (JAR) file. An EJB project can be used to build stand-alone components, or it can be combined with other modules in a J2EE enterprise application (EAR).

Recall the structure of an EJB module (see Figure 6.3 earlier). EJB modules have a simple structure that contains EJB classes and deployment descriptors. In the workbench, we can use the **New EJB Project** wizard to create a new EJB project with an EJB module in it.

**Getting an EJB Container**

EJB projects require a server runtime environment that supports EJBs. You will need an application server such as Geronimo, JBoss, or JOnAS to develop EJBs with WTP. You should obtain the application server first, and use the WTP preferences to define a new server runtime environment.

You can obtain Geronimo from

```
http://geronimo.apache.org
```

or you can download and install it via WTP (see the Installing Third-Party Content section in Chapter 4). JBoss can be obtained from

```
http://www.jboss.org
```

and JOnAS can be obtained from

```
http://jonas.objectweb.org
```

You will not be able to use Apache Tomcat for EJB development. Tomcat only supports J2EE Web modules, not EJBs or enterprise applications.

When you want to create an EJB project, you will typically do the following:

1. Switch to the **J2EE** perspective. In the **Project Explorer** view, right click, and invoke the **New ▸ EJB Project** menu item (see Figure 6.11).

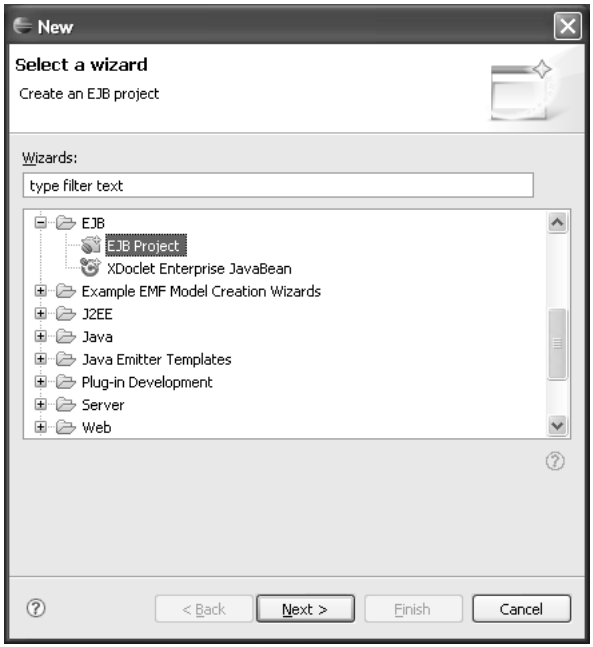


**Figure 6.11** Select Wizard

Click **Next**. The **New EJB Project** wizard opens (see Figure 6.12). Enter `LeaguePlanetEJB` for the project name and select a target runtime that supports EJBs such as JBoss. We will discuss EJBs in more detail later in Chapter 8.

Configurations allow you to choose a set of project facets for common styles of EJB projects. For example, if you choose the `EJB Project with XDoclet` configuration, WTP will set up the project to enable XDoclet. Click the **Next** button to proceed to the **Project Facets** selections page.

2. Project facets describe aspects of J2EE modules (see Figure 6.13). For an EJB module, you can specify the J2EE version, the Java version, and, optionally, the XDoclet version. Each server defines a set of supported facets and their allowed values. For example, you will not be able to set an

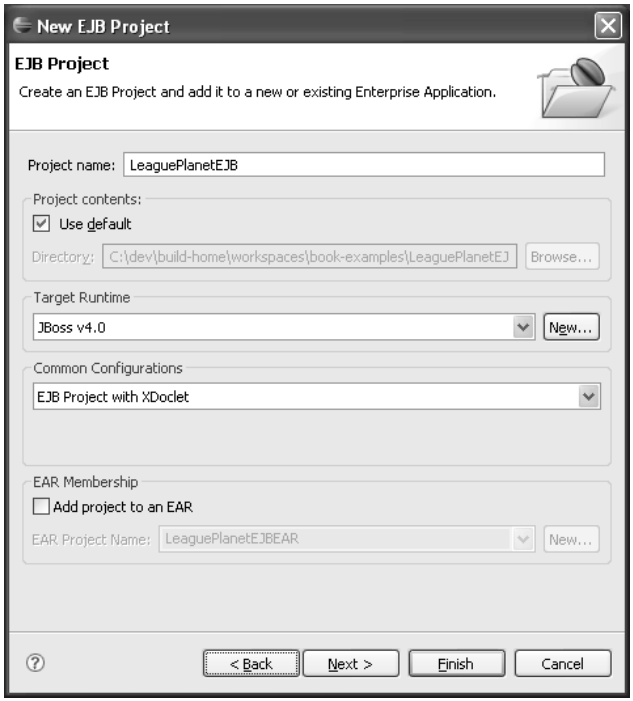


**Figure 6.12** New EJB Project

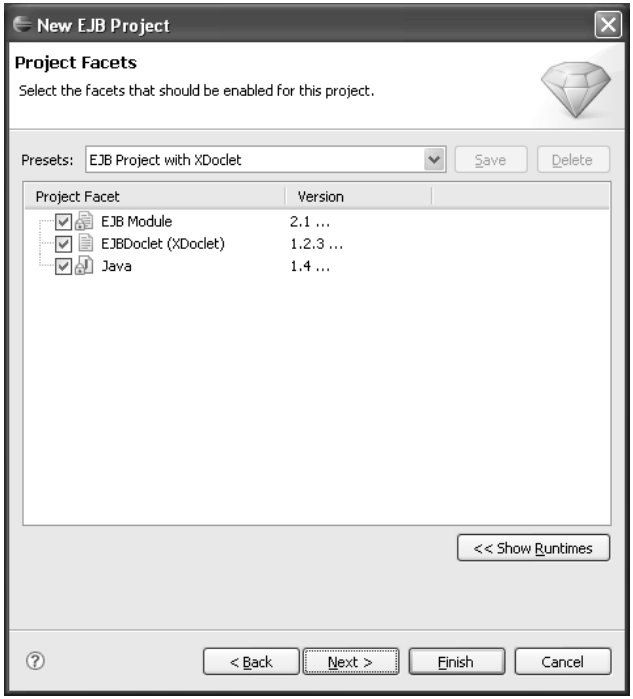


**Figure 6.13** EJB Project Facets

EJB facet using a Tomcat server because it does not have an EJB container. WTP configures the EJB module and sets up the classpath for the project so that it matches the specified facets. Here, you will use XDoclet to develop EJBs. Add the XDoclet facet by checking it. Accept the defaults for the EJB and Java facets and click the **Next** button to proceed to the EJB module settings.

3. The **EJB Module** page (see Figure 6.14) lets you specify the directory for Java resources. Optionally, you can create a Java utility module that will contain EJB classes and interfaces, which will be required by EJB clients. Click **Finish**.

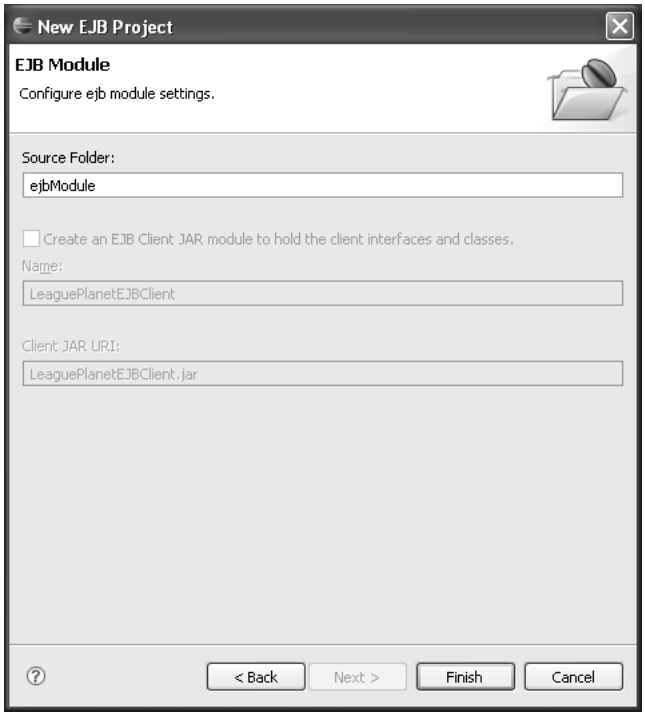


**Figure 6.14** EJB Module

4. WTP creates the EJB project and populates it with configuration files such as the EJB deployment descriptor, `ejb-jar.xml` (see Figure 6.15).

You may notice some errors in the new EJB project. For example, if your EJB project does not contain any EJB components, this is considered an error according to the J2EE specification. If you chose the XDoclet facet and an XDoclet runtime is

not yet configured, this will show up in the problem markers. These errors are normal and will be removed when you fix the preferences and add EJBs to the project.

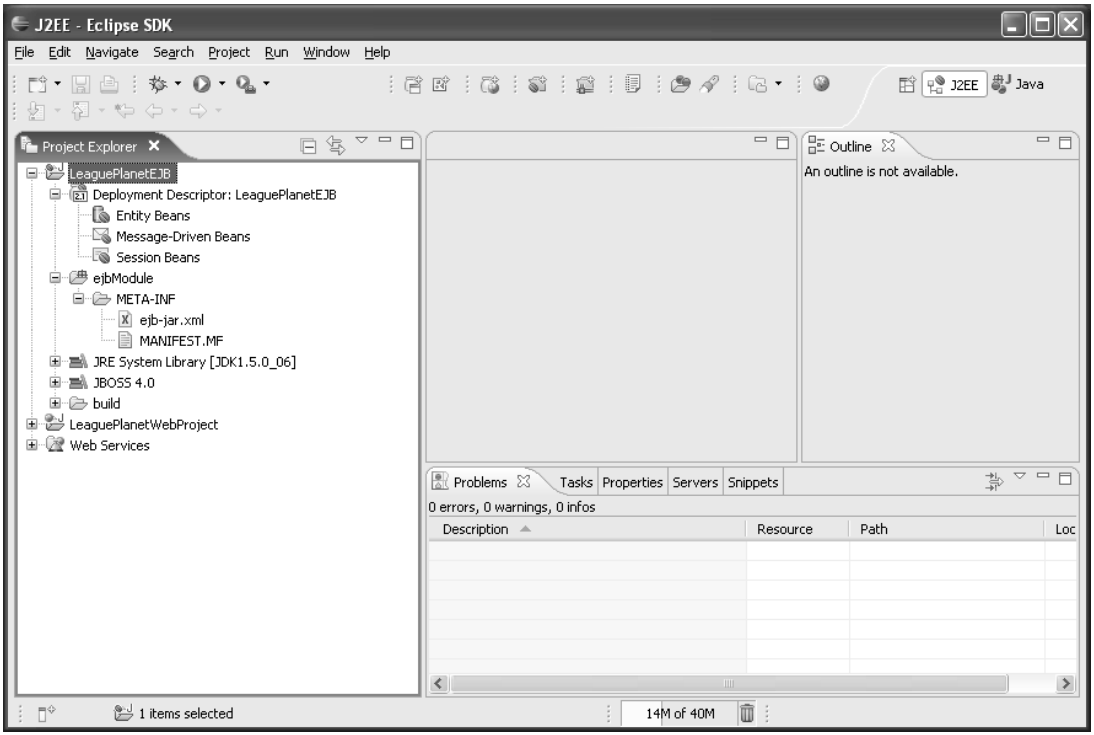


**Figure 6.15** Project Explorer—EJB Project

The `ejbModule` folder contains Java and EJB resources such as the deployment descriptor (see Figure 6.16).

Similar to Web application modules, an EJB project has a publisher for EJB applications (see Figure 6.17). This publisher creates a deployable EJB module from the contents of the project with all the classes and deployment descriptors.

### EJB Client Projects

There is another EJB related project type called the **EJB Client Project**. These projects are used to share common classes between EJB modules and their clients such as a Web application. Typical classes that are found in these modules are the EJB interface types and models. EJB project wizards can create an EJB client project. This option can be selected only when the EJB module is added to an EAR module. It is also possible to add the client project to an existing EJB module by using the context menu in the **Project Explorer** view.

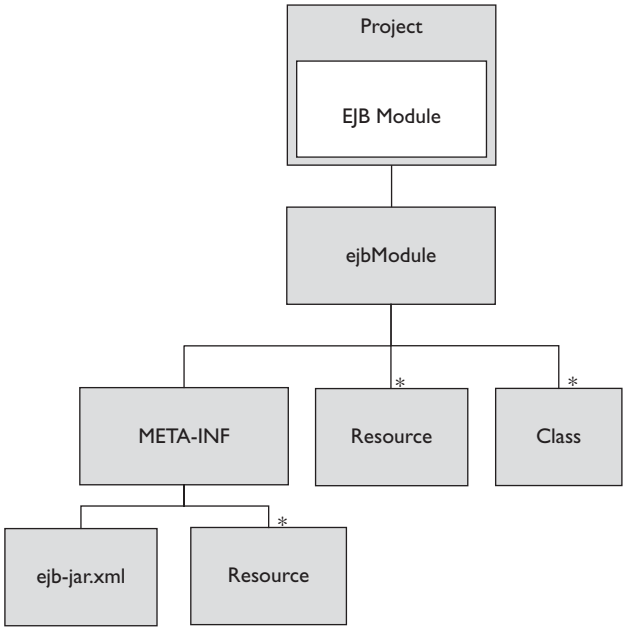


**Figure 6.16** Elements of an EJB Project

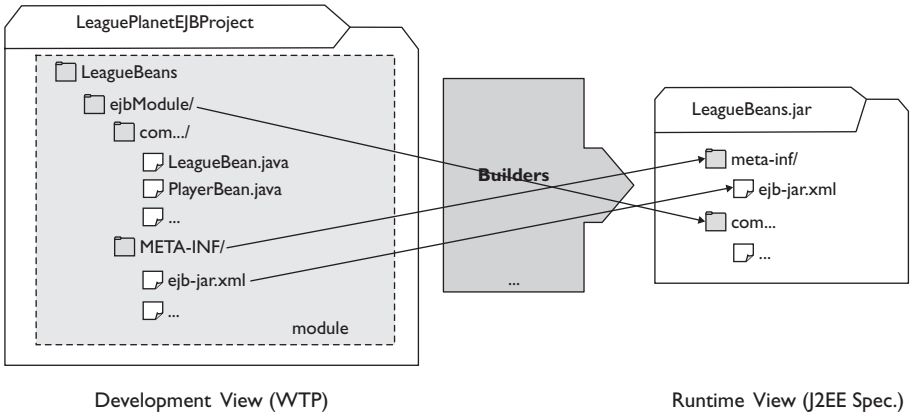


**Figure 6.17** EJB Publisher

This completes the process of creating an EJB project. The next section describes how to create an enterprise application project that can combine EJB and Web modules in a J2EE Enterprise Application (EAR) module.

## Creating Enterprise Applications

The most interesting J2EE enterprise applications have more than one module. They have several Web applications and EJB modules. The J2EE specification provides a basic application packaging structure called an *enterprise application*. Enterprise application archives are packaged as Java archives with the `.ear` suffix. Therefore, they are also known as *EARs*. An EAR can contain one or more

- EJB modules
- Web application modules
- J2CA resource adapter modules
- Application client modules

An enterprise application project contains the hierarchy of resources that are required to deploy these modules as a J2EE enterprise application.

An enterprise application module contains a set of references to the other J2EE modules that are combined to compose an EAR. In addition to the modules, an enterprise application module also includes a deployment descriptor, `application.xml`.

Publishers for enterprise application projects consume the output of the publishers from their component modules (see Figure 6.18). For example, the builder of an EAR that contains a Web application module and an EJB module waits until the builder for the Web and EJB projects creates the deployable structures for these modules, and then it assembles these artifacts in the EAR.

WTP has wizards and tools to create and edit EARs. They are described in the following use cases.

### Create a New Web or EJB Module in an EAR

When a new J2EE module project is created, such as a dynamic Web project or an EJB project, it can be associated with an enterprise application project (see Figure 6.19). The project wizards let you specify a new or existing enterprise application. You can also choose the project in which you would create the enterprise application module. Finally, the EAR is updated to include the new J2EE module in it.

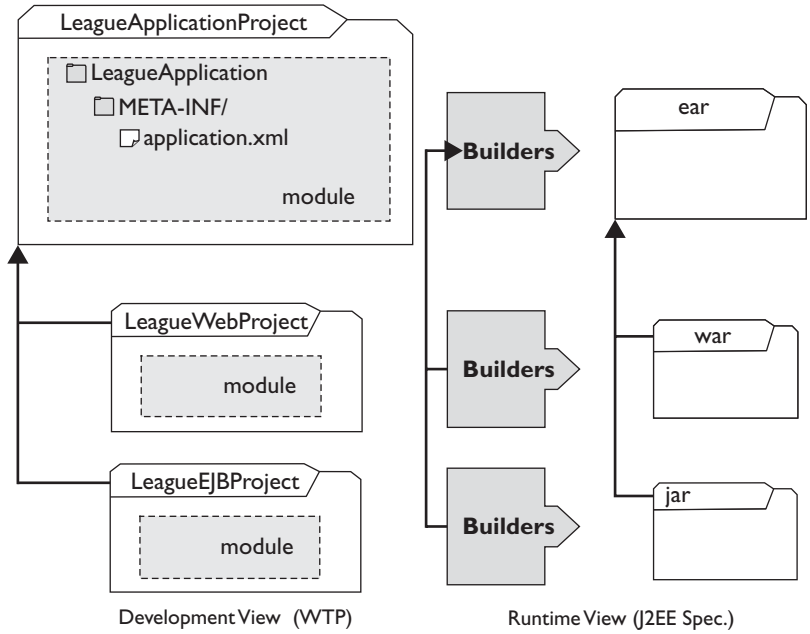


**Figure 6.18** EAR Publisher

### *Adding Existing Web and EJB Modules to an EAR*

In the second scenario there are existing J2EE modules, which are to be added to a new enterprise application. You create a new EAR project and add your existing modules to it. The Enterprise Application wizard creates a new project and allows you to choose the modules to be included in it.

When you want to create an EAR project, you will typically do the following:

1. Switch to the J2EE perspective. In the Project Explorer view, right click, and invoke the New ▸ Enterprise Application Project menu item (see Figure 6.20).
2. Click Next. The New Enterprise Application Project wizard opens (see Figure 6.21).
3. Enter `LeaguePlanetEar` for the Project name. Click the Next button to proceed to the Project Facets selection page.

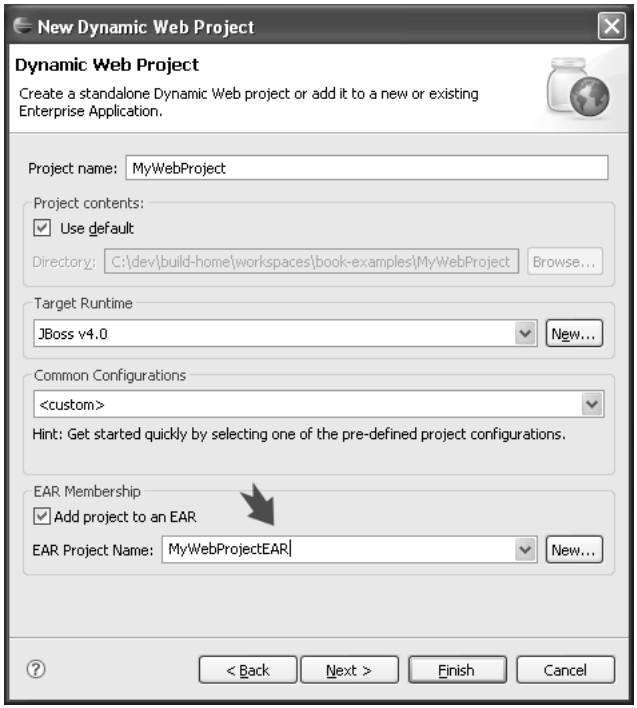


**Figure 6.19** Adding a Module to an EAR

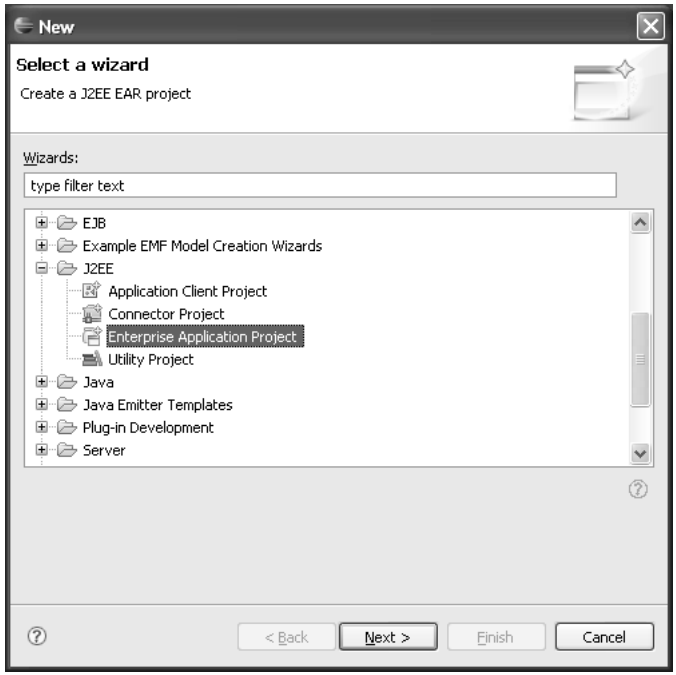


**Figure 6.20** Select Wizard


**Figure 6.21** New Ear Project

4. Project facets describe aspects of enterprise applications (see Figure 6.22). For the EAR module, there is only the EAR facet. Each server defines a set of supported facets and their allowed values. For example, you will not be able to set an EAR facet using a Tomcat server because it does not support EARs. Click the **Next** button to proceed to the EAR module settings.
5. The **J2EE Module** page (see Figure 6.23) lets you select the modules that will be included in the application. Select the `LeaguePlanetEJB` and `LeaguePlanetWebProject` modules. Note that you can also make the wizard generate new empty modules by clicking the **New Modules** button. Click **Finish**.
6. WTP creates the EAR project and its deployment descriptor, `application.xml` (see Figure 6.24).

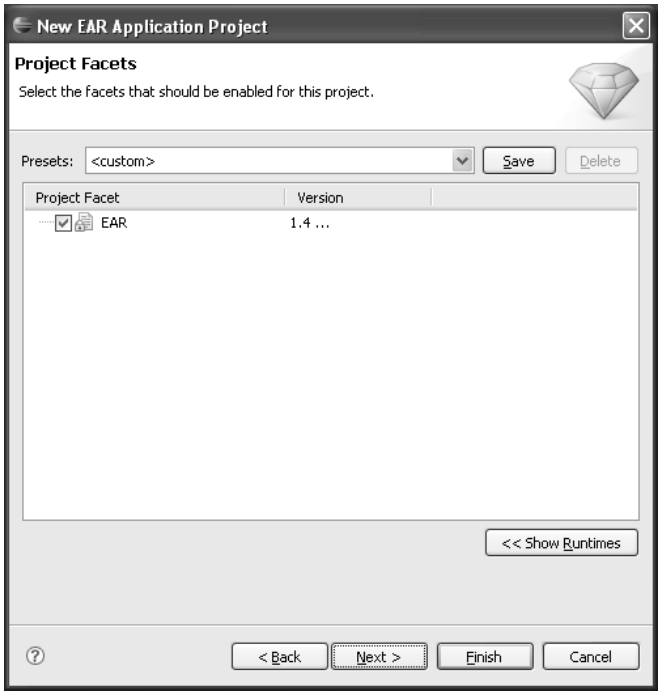


**Figure 6.22** EAR Project Facets

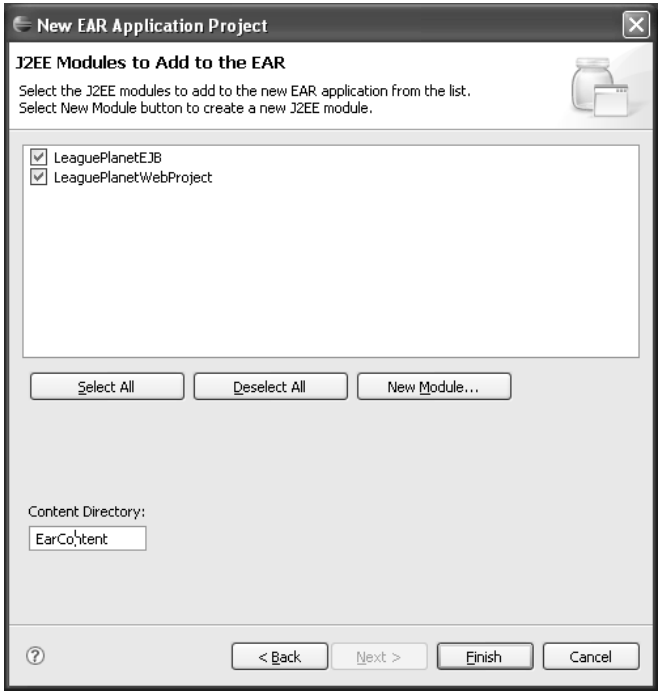


**Figure 6.23** J2EE Modules

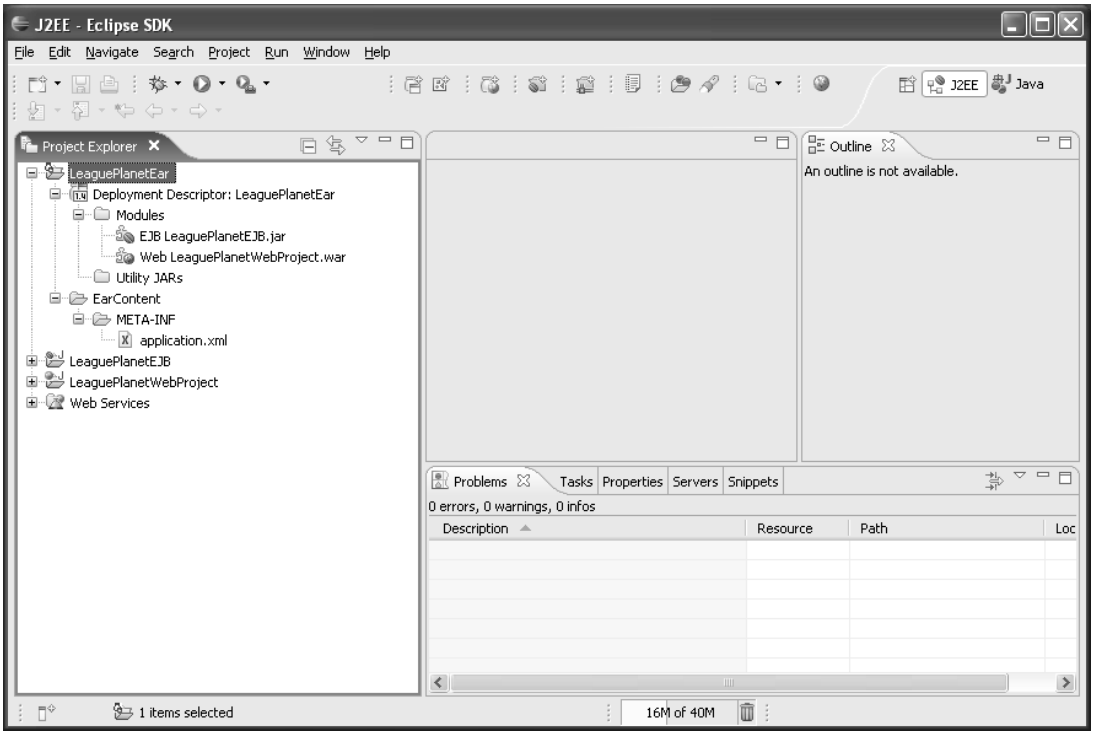


**Figure 6.24** Project Explorer—EAR Project

### *Editing EARs*

In the final scenario, you modify the modules in an EAR. You can add new modules to an EAR or remove existing ones by using the **J2EE Module Dependencies** property page.

When you want to modify an EAR project, you will typically do the following: In the **Project Explorer**, highlight the enterprise application `LeaguePlanetEar`, right click, and select **Properties**. As Figure 6.25 shows, you can then choose the modules to be included in the EAR.

EAR modules have a simple structure. When modules are added or removed from an EAR, WTP automatically updates the module and the contents of the EAR deployment descriptor, `application.xml`, which is stored in the `META-INF` directory.

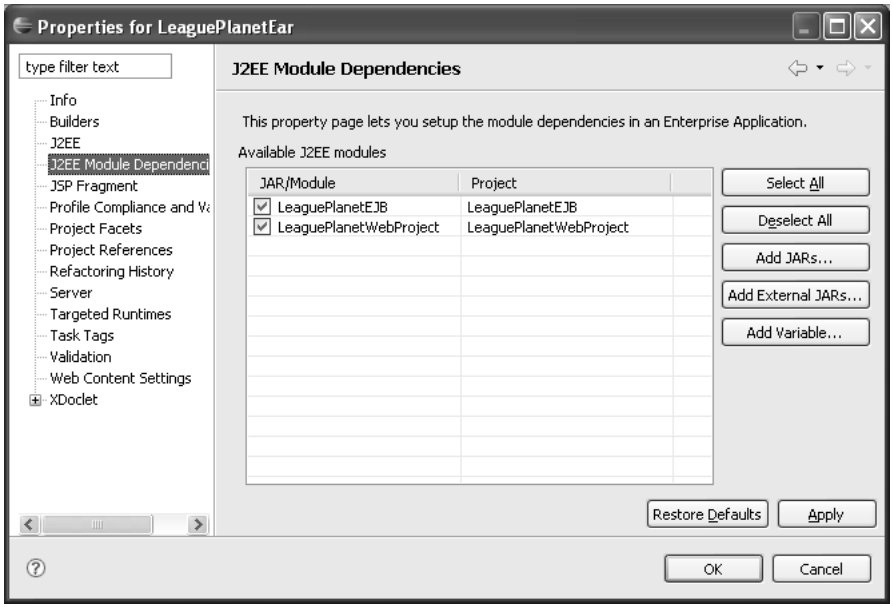


**Figure 6.25** J2EE Module Dependencies

## Advanced Web Projects

The default project types and layouts cover many of the common application and development needs. Sometimes you need to do more with a Web project; you can use it to improve your development process, organize your code, and share your work with other team members.

Here are some development considerations that can determine the organization of a project:

- *Project Deliverables:* These are the concrete outputs of the development activities. For example, in a J2EE development project, deliverables are the standard modules such as Web application archives (WARs), EJB component archives (JARs), Enterprise application archives (EARs), and so forth. Architecture also influences the design of deliverables. You may use a single EAR project if all the containers run on the same application server. However, it will be better to divide the projects if the Web and EJB containers are on different servers.

  Some projects are simple Web applications while others involve multiple modules and components. An application may group many Web applications and EJBs together. The J2EE specification describes a structure for these deliverables.

- *Team Organization:* Team organization determines who will do what in the project. A team can be one person or it can have groups of developers. The structure of the project is a significant factor in determining the productivity of the team and the management of the overall software engineering process.
- *Change Control, Configuration and Release Management:* Software can be viewed in terms of components that are assembled and configured to form an application. It is important to track the changes to these components using a version control system. The organization of these components determines the units that are used to control the changes in the scope of the project. The configuration and version of components that make an application are very important to the release process.
- *Testing:* Test plans, test cases, and execution of the tests must be regular and continuous parts of the development process. Test objectives and responsibilities are determined based on the modules. Unit and integration tests are part of the development for each module.

When the WTP project was started, the development team had long discussions on how to extend the basic Java projects to handle different styles of custom projects. A key requirement for Web projects was to enable the separation of the two fundamental view points to help manage resources in a project, for example, the *developer view* and the *runtime view*.

The runtime view is defined by the J2EE specification. The developer's view is most often modeled using the J2EE specification. Mimicking the structures defined in the specification creates valid J2EE applications, but this is not always suitable for all development projects.

In WTP, the developer's view of a project is captured by a model that maps the contents of the project to the runtime view. Each WTP Web project has a *structural model* that is used to describe how developers lay out the resources. Publishers and WTP tools use the structural model to create J2EE artifacts. This mapping gives you flexibility to create projects in ways that you could not do before. For that reason, WTP developers sometimes also refer to these projects as *flexible projects*. We'll use the term *Web project* in this book.

Technically speaking, an Eclipse project that has the `Module Core Nature` is a Web project. This nature indicates that these projects have a structural model for the modules and will support WTP tools. We will start with a short description of this advanced project capability, and then give examples demonstrating its use. Power users can employ these capabilities to create many different layouts for their projects.

### Modeling the Developer View

The structural model of a Web project tells publishers how to compose a runtime artifact (see Figure 6.26).

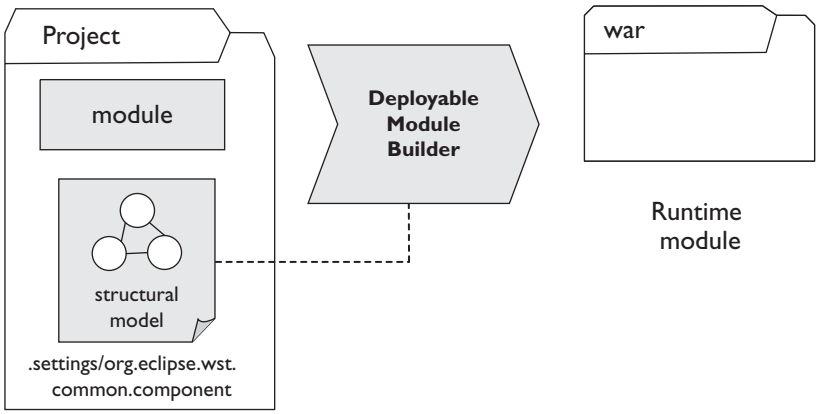


**Figure 6.26** Structural Model

This model is defined in an XML component file stored with the other project settings. The project settings and component files are normally invisible in the Project Explorer view. However, they are visible in the Eclipse Navigator view that is included in the Resource perspective. The structural model is stored in a file named

```
org.eclipse.wst.common.component
```

inside the `.settings` folder of any Web project (see Figure 6.27).

The model file listed in Example 6.1 is for a typical dynamic Web application module. The module is named `LeaguePlanetWebProject`. The model specifies how resources in the development view map to resources in the runtime view. Here, you map the complete contents of the `WebContent` folder to the module root. The `source-path` is relative to the project root and the `deploy-path` is relative to the module root at the destination. You can have as many resource mappings as you like for each module. The module also has type-specific properties such as `context root`, which defines the context root of the Web application module. The `java-output-path` property tells the publisher where to find the compiled classes.

**Example 6.1** Web Module Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<project-modules id="moduleCoreId" project-version="1.5.0">
    <wb-module deploy-name="LeaguePlanetWebProject">
```

```
        <wb-resource source-path="/WebContent" deploy-path="/"/>
        <wb-resource source-path="/src" deploy-path="/WEB-INF/classes"/>
        <property name="context-root" value="LeaguePlanetWebProject"/>
        <property name="java-output-path" value="build/classes"/>
    </wb-module>
</project-modules>
```

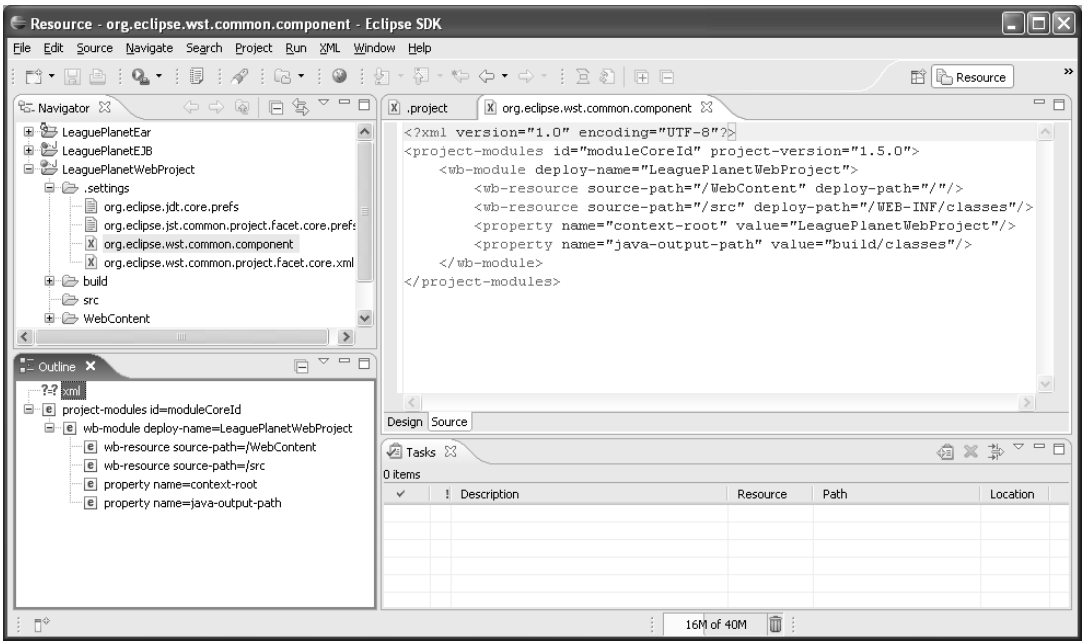


**Figure 6.27** Structural Model Definition

Another example is the model of an enterprise application (see Example 6.2). Here the interesting parts are the dependent modules. In this example, the EAR uses an EJB module and a Web module. A dependent module is referenced using a handle, which is a module URL. A module URL starts with the prefix `module:`, and is followed by a workspace-relative path to determine the project and the name of the module within that project.

**Example 6.2** EAR Module Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<project-modules id="moduleCoreId" project-version="1.5.0">
  <wb-module deploy-name="LeaguePlanetEar">
    <wb-resource source-path="/EarContent" deploy-path="/" />
    <dependent-module deploy-path="/"
      handle="module:/resource/LeaguePlanetEJB/LeaguePlanetEJB">
      <dependent-object>EjbModule_1147426182270</dependent-object>
```

```
      <dependency-type>uses</dependency-type>
    </dependent-module>
    <dependent-module deploy-path="/"
     handle="module:/resource/LeaguePlanetWebProject/LeaguePlanetWebProject">
      <dependent-object>WebModule_1147426182290</dependent-object>
      <dependency-type>uses</dependency-type>
    </dependent-module>
  </wb-module>
</project-modules>
```

The structural model is a mapping for the organization of files that are distributed over a set of Web projects. A publisher uses this model and can construct a deployable, runtime Web artifact as described in the J2EE specification.

When you create projects and modules using a project creation wizard, the model is automatically added to a project. Wizards create a model based on a default template. However, you can easily modify the default mapping as shown in the next sections. Some of the common types of artifacts used in model definitions are resources, modules, and dependent modules.

#### *Resource*

A *resource* is an abstraction of project artifacts such as files, folders, and libraries. An Eclipse project maintains its resources, ensuring that each resource is loaded only once within the workspace. Resources are referenced with resource URIs, which are relative to the projects that contain the resource. WTP has additional URI converters that can resolve URIs to their underlying physical resource, such as the module URI we discussed earlier.

#### *Module*

A *module* represents a deployable artifact, such as a WAR, EJB JAR, or EAR. A WTP project can be associated with only one module, but it can refer to others. This makes it possible to distribute the code for a module over a set of projects.

A J2EE module has a standard layout and is targeted to some J2EE runtime container. J2EE projects generate archives as JARs or as exploded archives. These archives must contain compulsory files, such as deployment descriptors, and must conform to the J2EE specification. There are five core types of J2EE modules and a general-purpose utility module:

- Enterprise application (EAR)
- Enterprise application client (JAR)
- Enterprise JavaBean (JAR)
- Web application (WAR)

- Resource adapter for J2CA (RAR)
- Utility modules (JAR)

#### Dependent Module

As its name suggests, a *dependent module* is used to define dependencies between modules. It can also help define a module with its code split into several projects. For example, we can maintain the Web applications that are in an enterprise application as dependent modules. Another common pattern is to maintain basic utility JAR modules, which contain the extracted contents of the archive, as separate projects. The benefit of using extracted modules is that all the artifacts can be modified, and Web projects assemble them into a deployable form.

## Example Projects

It is time to discover how you can create some interesting projects. These best practices provide different styles of projects for Web and J2EE development. You can extend and customize these examples to fit your needs. The examples we'll discuss are a basic enterprise application, dividing a Web module into multiple projects, and using Maven for Web application development.

### Basic Enterprise Application

Using the J2EE application deployment specification as a template, you will create an enterprise application with multiple modules. This is recommended if you do not have a compelling reason to do it another way. These projects map to the J2EE specification in a straightforward way and can be created using wizard defaults. Adherence to standards reduces the behavioral discrepancies between the runtime and the development environments.

In this example, each architectural application layer will correspond to a project. For example, the presentation layers will correspond to a dynamic Web project with a Web application module and the business logic layer to an EJB project with an EJB module. The enterprise application project will be used to assemble the modules as a single coherent unit.

To create this structure, you will use a J2EE Enterprise Application Project (see Figure 6.28). The EAR project has two modules: `LeaguePlanetWebProject`, a Web application module; and `LeaguePlanetEJBProject`, an EJB module. The Web application module is going to be a dynamic Web project with the same name. The EJB module is divided into an EJB project and the EJB client project. The EJB client JAR is a Java utility project named `LeaguePlanetEJBClientProject`.

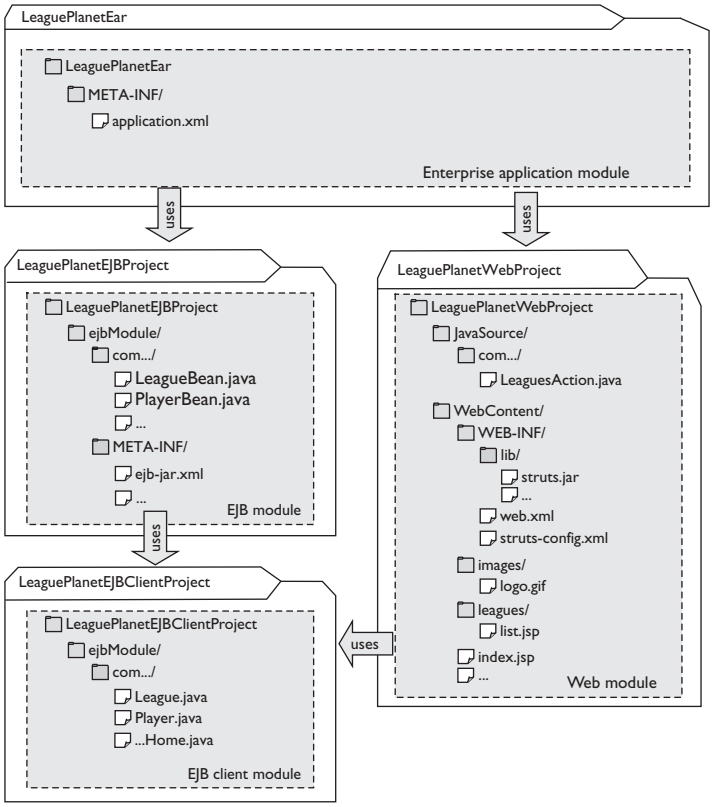


**Figure 6.28** Module Dependencies for League Planet Application

To demonstrate the use of Web application libraries, the Web application will use the Struts MVC framework. In order to use Struts, all Struts and supporting libraries, that is, `struts*.jar`, `commons*.jar`, `jakarta*.jar`, `log4j.jar`, and `antlr.jar`, are kept in the `WEB-INF/lib` directory. The Struts configuration file, `struts-config.xml`, is in the `WEB-INF` directory. The business model for League Planet is provided by the EJBs. The Web application delegates the business behavior to this layer.

### Clean Workspace

In the first part of this chapter, we described how you can create different types of projects. In this example we will use the same names. If you have tried the earlier examples and are using the same workspace, you should delete those projects before starting this one. If you would like to keep the old work, remember to back up.

To create an EAR project with this structure, do the following:

1. Start as we described earlier in this chapter to create a new **Enterprise Application Project**. Name it `LeaguePlanetEar`. Select the default facets, continue to the **J2EE Modules** page, and click **Finish** to create an empty EAR. In the next steps you will create the Web and EJB projects.
2. Repeat the steps we described earlier in this chapter to create a new **Dynamic Web Project**. Name it `LeaguePlanetWebProject`. Choose the `LeaguePlanetEar` as the EAR for the Web project (see Figure 6.29). Continue to the other pages to select the default facets, and click **Finish** to create the Web project. The EAR project will be automatically updated to reflect the addition of the new Web module.

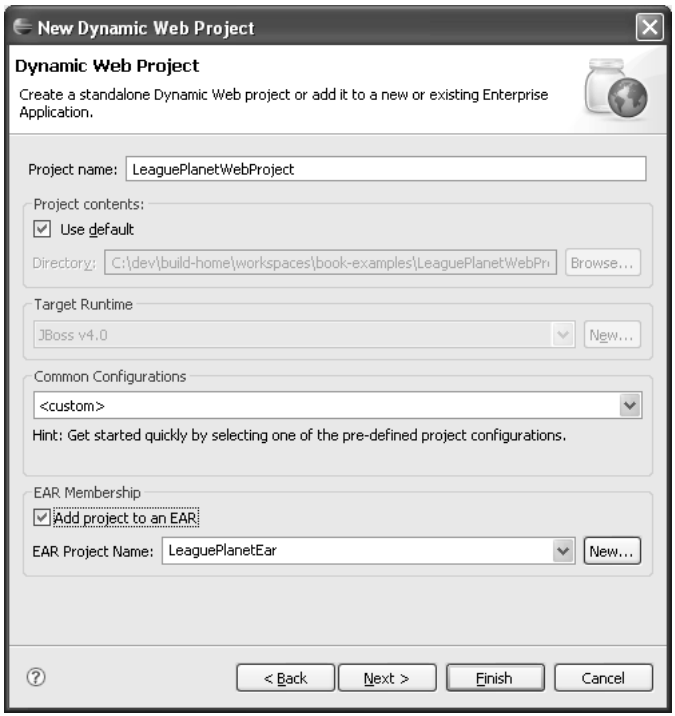


**Figure 6.29** Web Project Added to an EAR

3. To do this step, you must have the Struts framework installed someplace on your machine. You can obtain Struts from

   `http://struts.apache.org`

Import all the Struts libraries and their supporting libraries into

```
WebContent/WEB-INF/lib
```

Refer to the Struts documentation for the exact list of libraries. Once the JARs are copied into the `lib` folder, they will be automatically added to the build path under the `Web App Libraries` category (see Figure 6.30).

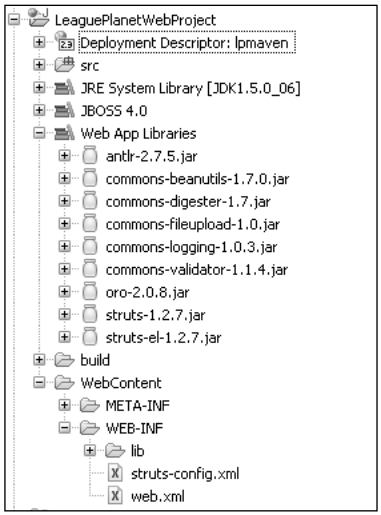


**Figure 6.30** Web App Library

4. Repeat the steps we described earlier in this chapter to create a new EJB project. Name it `LeaguePlanetEJBProject`. Choose the `LeaguePlanetEar` as the EAR for the EJB project (see Figure 6.31). You can choose one of the default facet configurations for development, such as the **EJB Project with XDoclet**. You do not need to change the default choices. If you do choose one, you should make sure that your workspace is set up to use it (that is, the XDoclet settings are valid). Click **Next** to go to the other pages to select the default facets. Click **Next** to go to the **EJB Module** page.
5. The Web application will be a client of the EJB module. Create an EJB client module named `LeaguePlanetEJBClientProject` (see Figure 6.32). Click **Finish** to create the EJB and EJB client projects. The EAR project will be automatically updated to reflect the addition of the two new modules.

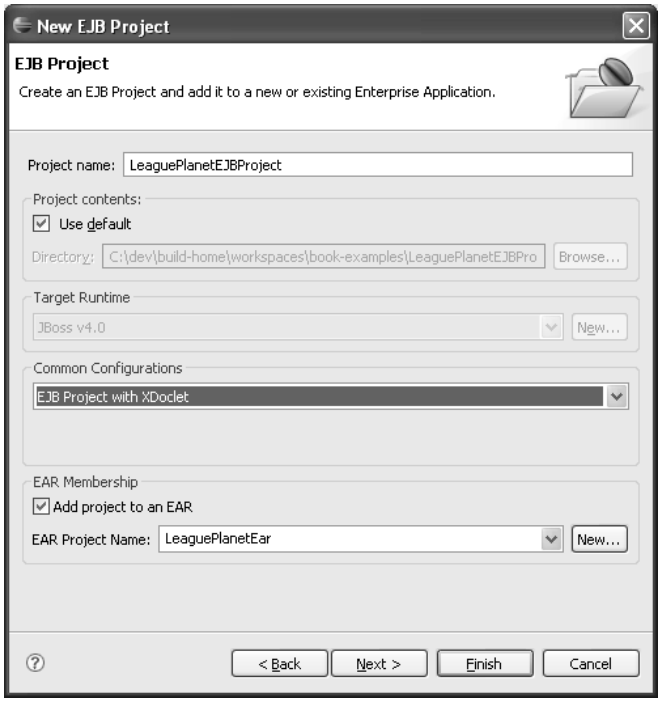


**Figure 6.31** EJB Project Added to an EAR


**Figure 6.32** EJB Client Module

6. WTP updates the EAR project and the deployment descriptor, `application.xml` (see Figure 6.33).

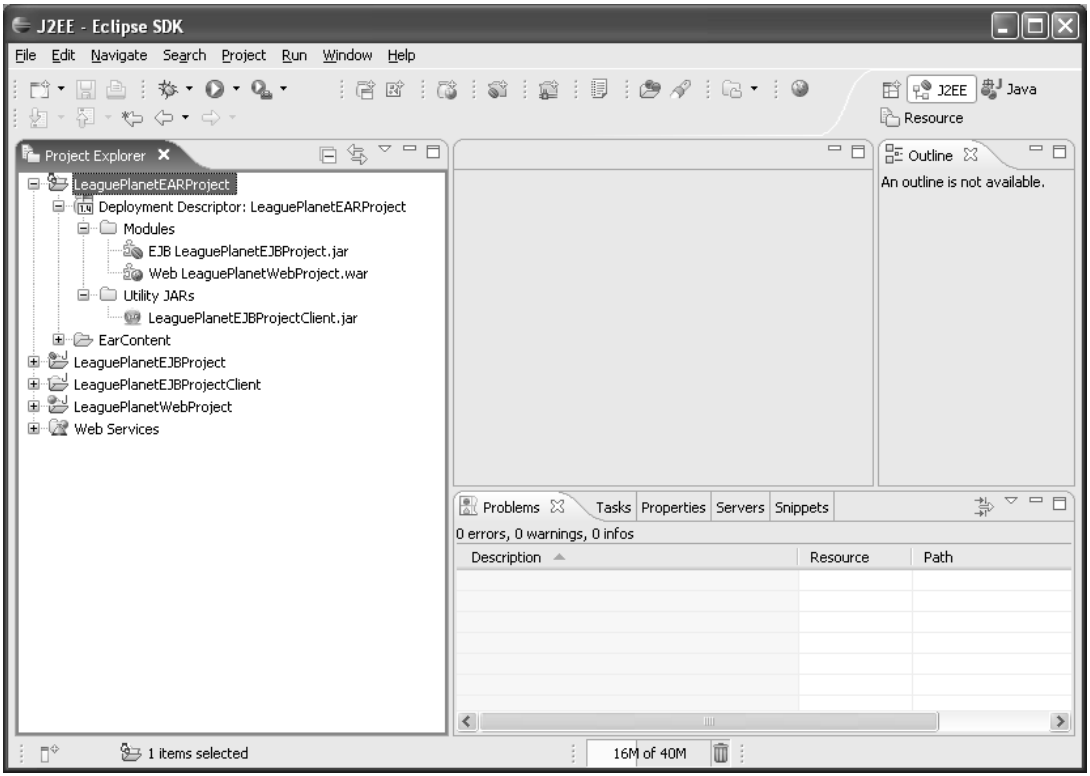


**Figure 6.33** Project Explorer—EAR Project

To create these projects, you used the same wizards described earlier in this chapter.

### Web Application Module Uses EJB Client

You need to make sure that the dependency between the Web application module and the EJB client is set. The Web application is a client of the EJB module. You need to describe this dependency. Remember that you created an EJB client module named `LeaguePlanetEJBClientProject`. You will add this module to the J2EE dependencies in the Web project. Select the Web project in the **Project Explorer,** right click and invoke the **Properties** menu item. Select the **J2EE Dependencies** page. In this tab, select `LeaguePlanetEJBClient` from the list (see Figure 6.34).

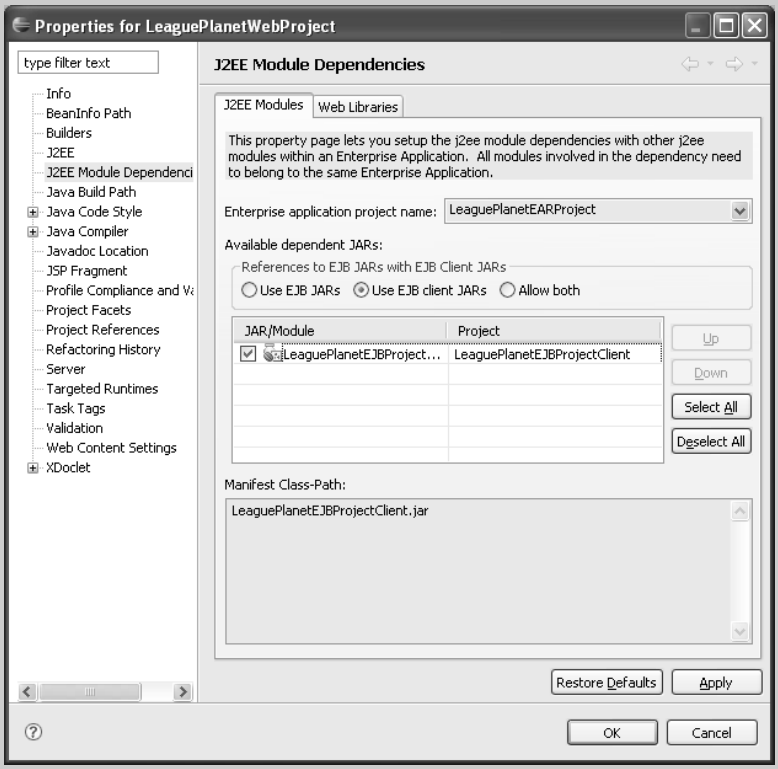


**Figure 6.34** Dependency to EJB Client Module

Later, you can extend this model by adding more Web projects—an administration site, for example. The business model can be extended with more EJBs.

### *Dividing a Web Module into Multiple Projects*

Size, structure, and the geographical and sociological aspects of a development team are significant factors in determining the project layout. When these are important to a project, they can determine the structure. The key constraints for this template are the manageability and divisibility of work. Manageability relates to aspects such as ownership of code, development responsibilities and tasks, configuration and version control, integration, and release management. Divisibility relates to dividing the work between members of the development team.

In this example, you will extend the project structure described in the previous example. `LeaguePlanetWebProject` is a large Web application module. It will

contain many large, loosely coupled subsystems. League management, player management, sponsorship, and advertising are some of these subsystems that will be developed by different teams. You will divide and manage subsystems as separate projects. Each subsystem can be released on different schedules. You will therefore start by dividing the Web module into two projects (see Figure 6.35). You can increase the number of subsystems following the same pattern later on. The dynamic Web project in the previous example contains the Web application module and will have common Web components such as menus, navigation bars, and so forth. There is a new subsystem for league management. This is a Java utility project on steroids. The league management module has its own presentation layer with JSPs and Struts configuration files in addition to its own Java classes.

To create this structure, you will need to create a new basic **Java Utility Project** named `LeaguePlanetManagementWebProject`. Java utility projects can be used to refactor reusable components of applications into separate projects. J2EE module dependencies will help assemble these components automatically.

To create the **Java Utility Project** and divide the module, the following steps must be performed:

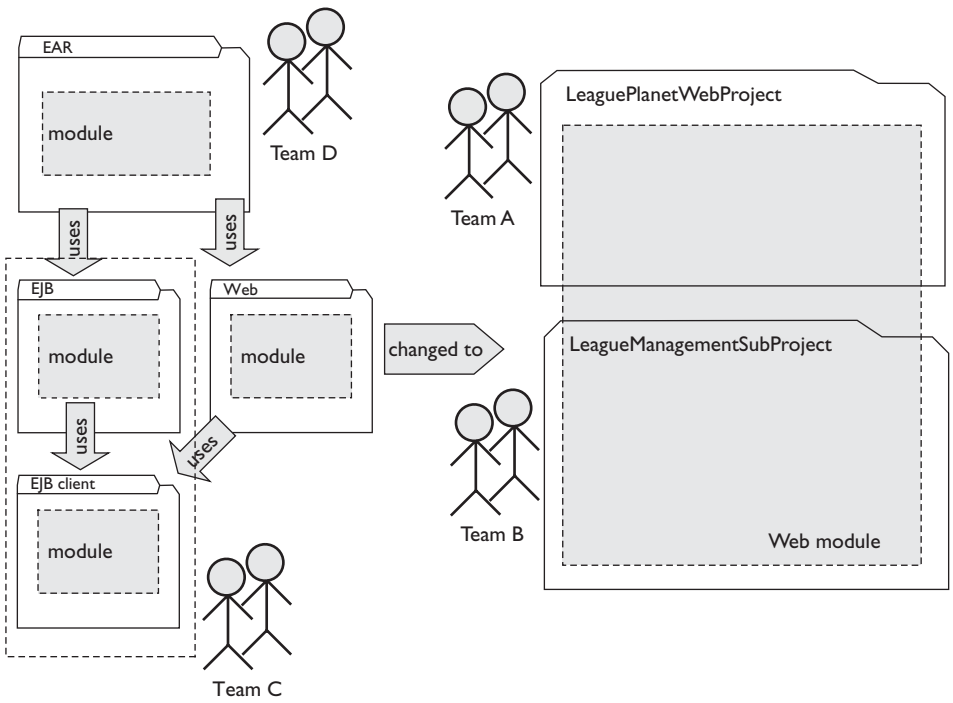


**Figure 6.35** Dividing a Web Module into Multiple Projects

1. Create a new **Java Utility Project** using the wizard.
2. Add the Web application libraries to its build path.
3. Add the utility project to the list of J2EE dependencies for the Web project.
4. Create a new `WebContent` folder in the utility project and add this to the structural model.

Do the following:

1. In the **Project Explorer** view, right click and invoke the **New ▸ Other ▸ J2EE** menu item (see Figure 6.36). Select **Utility Project**.

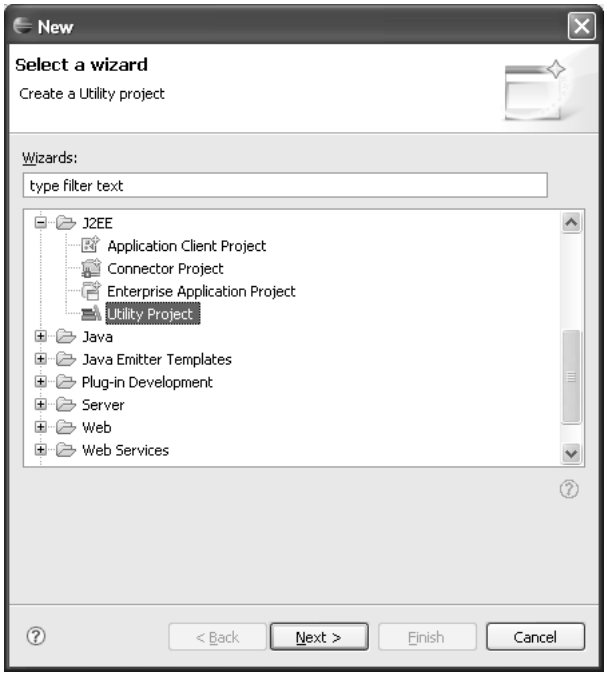


**Figure 6.36** Select Wizard

Click **Next**. The **New Java Utility Project** wizard opens (see Figure 6.37).

2. Enter `LeaguePlanetManagementWebProject` for the project name. Use the same target runtime for all your projects. Use the default configuration. Click the **Next** button. The **Project Facets** selection page is displayed (see Figure 6.38).

   Accept the defaults here and click **Finish**. WTP creates the empty utility project.

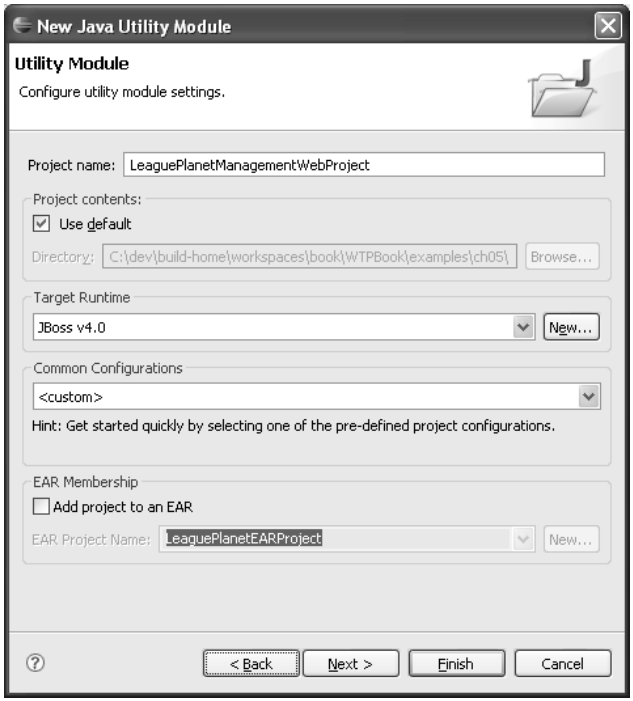


**Figure 6.37** New Java Utility Project

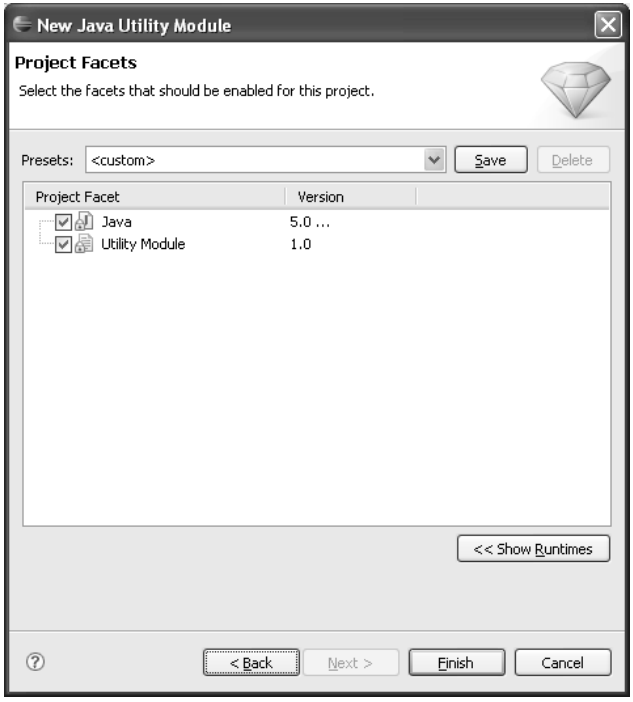


**Figure 6.38** Select Project Facets

3. You need to add this submodule to the J2EE dependencies of the Web project. To do this, select `LeaguePlanetWebProject` in the **Project Explorer**, right click, and invoke the **Properties** menu item. Select the **J2EE Dependencies** page. In this page, go the **Web Libraries** tab and add `LeaguePlanetManagementWebProject` from the list (see Figure 6.39).

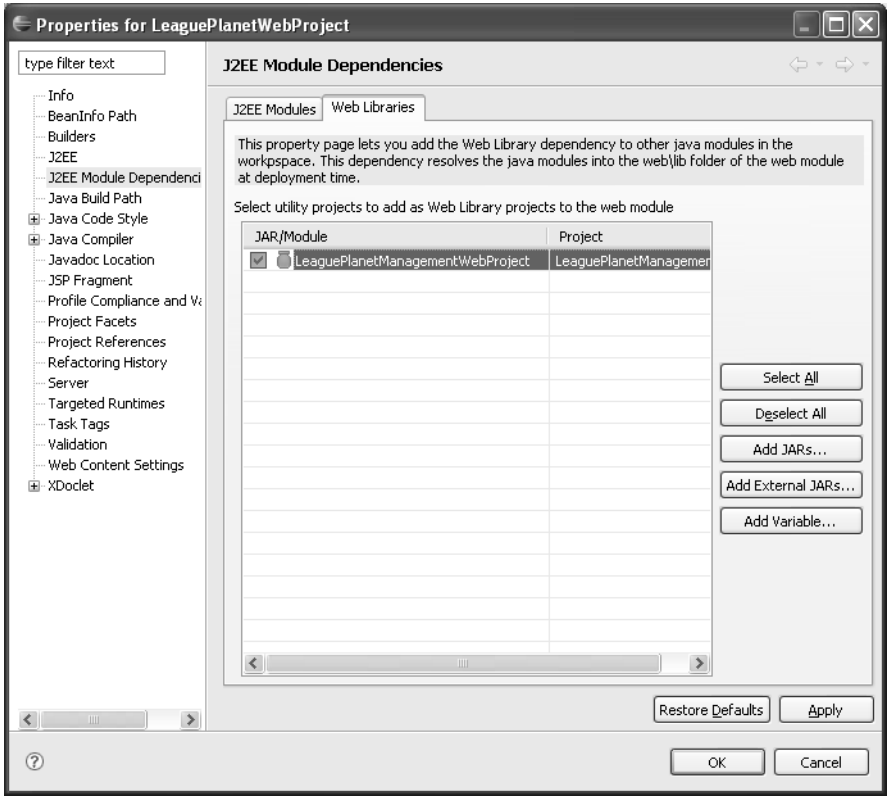


**Figure 6.39** Web Project Depends on Utility Project

### Managing the Web Application Classpath

When you add a dependency to a utility project, it is automatically added to the final WAR and to the **Web App Libraries** section of the build path of the Web project. However, the reverse is not true. The utility project has no knowledge of the Web application. If you have dependencies to external libraries, like Struts, in the original Web module, all JARs that are inside the `WEB-INF/lib` are available in the class loader of `LeaguePlanetWebProject`.

However, things can get a bit complicated if your new utility project needs classes from the Web application. For example, you may want to add new Struts actions to the utility project module or use Struts taglibs in the JSP files.

You can try to add `LeaguePlanetWebProject` to the build path of the utility project but this would create a circularity, so Eclipse will not allow it.

The best solution is to create other utility projects for common subsystems. These common utility projects can be added to the build path of the Web application as J2EE module dependencies and can also be included in the build path of the other utility projects as Java project dependencies. This approach avoids circularities.

Finally, some development teams prefer to maintain the binaries for external libraries, such as Struts or Hibernate, in a common folder but not in the Web project. For example, some use Maven repositories to maintain project dependencies to these JARs. You will learn about Maven in the next section. WTP allows you to maintain libraries externally and automatically assembles them into the final WAR file before publishing it to the server. If these libraries are added as J2EE dependencies, they are also automatically added to the build path. You can use the project **Properties** window and add them as an external JAR dependency on the **J2EE Module Dependencies** tab.

4. This is an optional step. The league management module is a part of the Web module, but it may need some external libraries to be on its build path. You can do this by adding the external JARs to the build path of the Java utility project. Select `LeaguePlanetManagementWebProject` in the **Project Explorer**, right click, and invoke the **Properties** menu item. Select the **Java Build Path** page. Click on the **Libraries** tab. In this tab, click **Add External JARs** (see Figure 6.40).

   The **JAR Selection** wizard will open (see Figure 6.41). This wizard allows you to browse your local file system for JARs.

   Select all the same external libraries, like Struts, that you have used for the Web project here, too. Click **Finish**. Apply and close the **Properties** window.

5. Next you will create a new `WebContent` folder in the league management project. In the **Project Explorer**, select `LeaguePlanetManagementWebProject`, right click, and invoke the **File ▸ New ▸ Folder** menu item. The **New Folder** wizard will open (see Figure 6.42).

6. Enter `WebContent` as the folder name. Repeat the same process to create a new `WEB-INF` folder inside the `WebContent` folder.

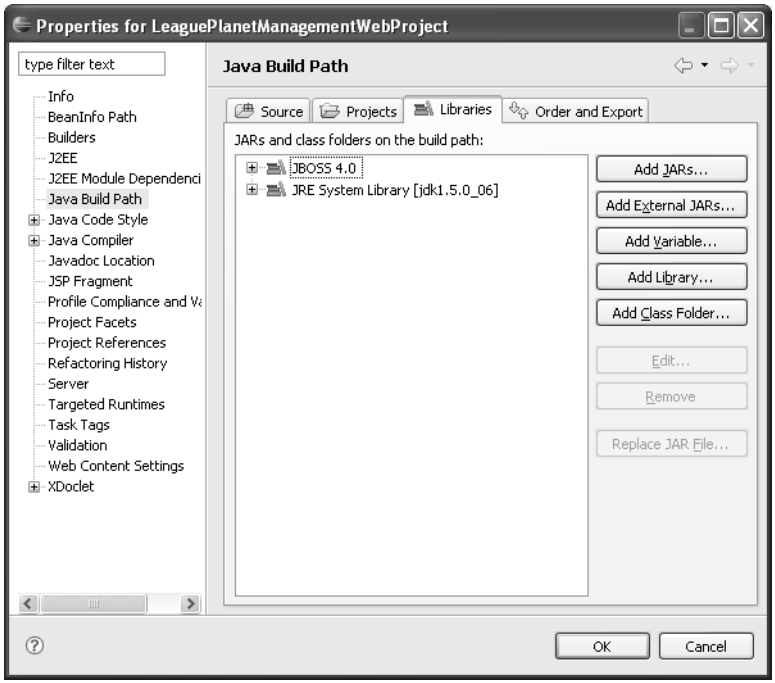


**Figure 6.40** Utility Project Java Build Path

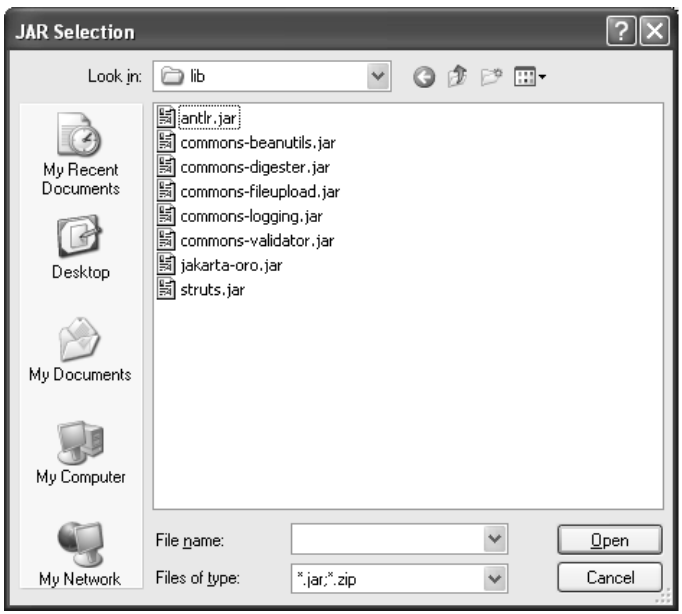


**Figure 6.41** Add External JARs Library

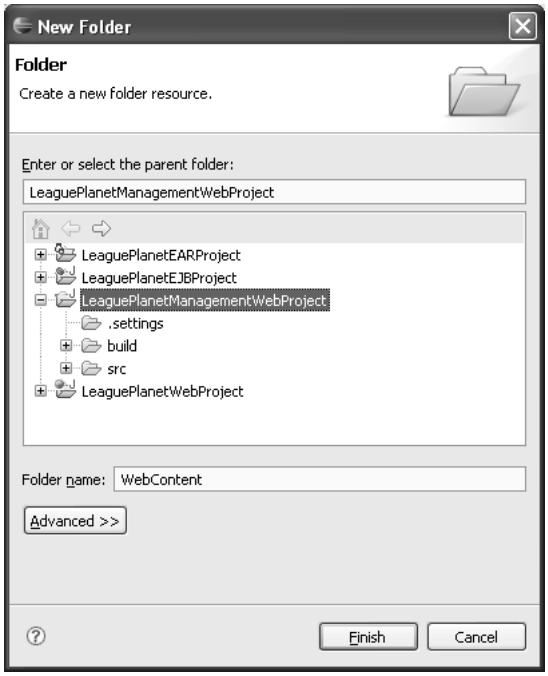


**Figure 6.42** WebContent Folder

7. Next you will link the new `WebContent` folder to the main Web project and add it to the structural model so that publishers will assemble the contents of the `WebContent` folder from the league management project into the overall project. In the Project Explorer, select `LeaguePlanetWebProject`, right click, and invoke File ▸ New ▸ Folder. The New Folder wizard will open (see Figure 6.43).
8. Enter `Management` as the folder name. Click on Link to folder in the file system. Click Browse to select the `WebContent` folder created in the previous step.

   You will need to specify that the `WebContent` folder in `LeagueManagementWebProject` gets copied into the deployable Web application module. Currently, there are no nice graphical tools to map these resources, so you will need to edit some files. You need to create the link to the `WebContent` folder before editing the module definition file. You already completed this step. Therefore, you can modify the XML component file to specify that this content folder is to be published with the Web module. This involves manually editing the

   ```
   org.eclipse.wst.common.component
   ```

   definition in the `.settings` folder. Edit the file as shown in Example 6.3.

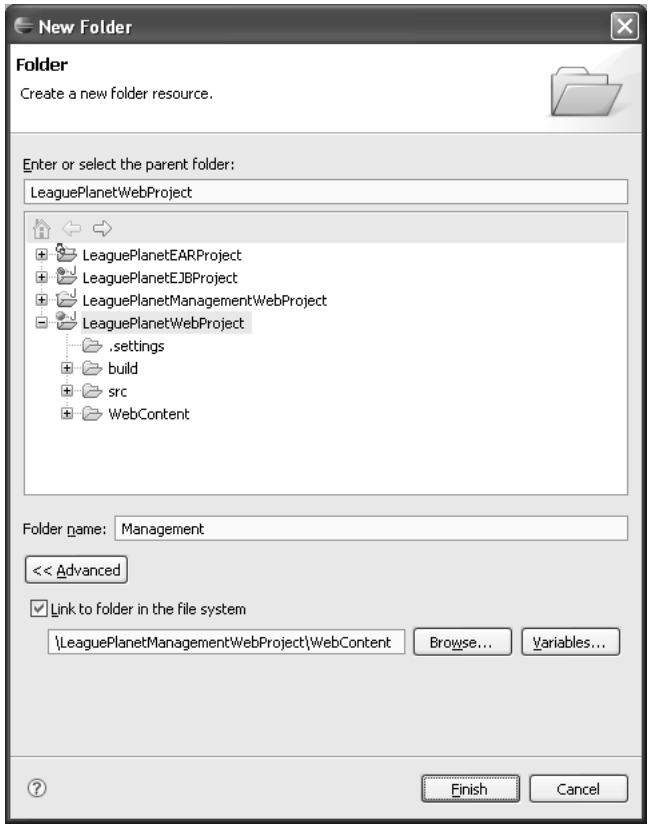


**Figure 6.43** Link to Management WebContent Folder

**Example 6.3** Modified Web Module Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<project-modules id="moduleCoreId" project-version="1.5.0">
  <wb-module deploy-name="LeaguePlanetWebProject">
    <wb-resource source-path="/WebContent" deploy-path="/" />
    <wb-resource source-path="/Management" deploy-path="/" />
    <wb-resource source-path="/src" deploy-path="/WEB-INF/classes" />
    <dependent-module deploy-path="/"
      handle="module:/resource/LeaguePlanetEJBProject/
      LeaguePlanetEJBProject">
      <dependency-type>uses</dependency-type>
    </dependent-module>
    <dependent-module deploy-path="/WEB-INF/lib"
      handle="module:/resource/LeaguePlanetManagementWebProject/
      LeaguePlanetManagementWebProject">
      <dependency-type>uses</dependency-type>
    </dependent-module>
    <property name="context-root" value="LeaguePlanetWebProject" />
    <property name="java-output-path" value="build/classes" />
  </wb-module>
</project-modules>
```

You have now split a Web module into multiple projects. The publisher will add the Java classes developed in the league management project as a JAR in the `WEB-INF/lib` folder to the original Web application module.

The publisher will also assemble any JSPs and additional Struts configuration files from the league management module, as well as all the Web content in this submodule. This content will be deployed with the Web application automatically. After the WAR is created, it will be assembled into the enterprise application as usual. When you are done, the workbench will have projects that look like Figure 6.44.

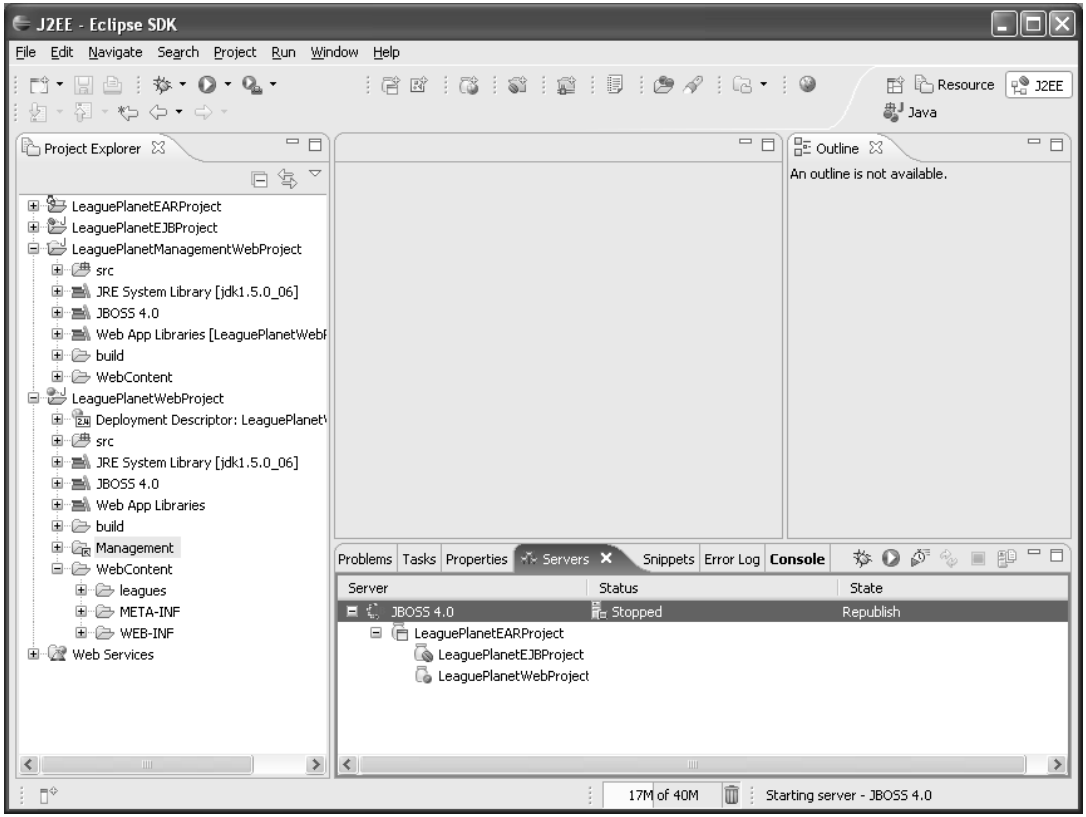


**Figure 6.44** Dependent Module in the Project Explorer

## *Using Maven for Web Application Development*

Maven is a software project management and comprehension tool. It started as a tool developed to build the Turbine project at `apache.org` and quickly spread to other Apache projects. Today, it is used as the main build tool for many of the

Java projects at Apache. For an in-depth description of how to use Maven on your project, refer to *Maven: A Developer's Notebook* [Massol2005] by Vincent Massol and Timothy O'Brien.

Maven is more than a Java build tool. It provides capabilities to make your life easy as a developer. Some of these capabilities are a well-defined project structure, a well-defined development process to follow, and a coherent body of documentation that keeps developers and users informed of what's happening in the project. This is essential in many team projects where there aren't enough people dedicated to the task of building, documenting, and propagating the information about the project. Maven captures the knowledge embedded in people's heads to do these tasks. For example, the development processes of Eclipse and Apache are evolutionary and resulted from the experiences gained from running many projects. This body of knowledge is typically captured in the tools that are used in building projects. Maven provides a standard environment that encourages the use of development and project best practices, and it disseminates this information to project stakeholders.

Following the success of Maven in Apache projects, many teams adopted Maven for their own use, including some J2EE projects. There is a set of J2EE-specific development best practices and processes captured in Maven. The use of Maven to develop a J2EE project enables the transfer of this knowledge. When a new J2EE project starts, it can immediately copy the build tasks and project know-how. The new project reuses the existing tools and conforms to the established practices. Maven does this by providing a framework and templates. For example, by having a common directory structure, developers are instantly familiar with a new project. To quote Aristotle, "We are what we repeatedly do. Excellence is not an act, but a habit."

There are other, less well-known approaches, such as JOFFAD, that also provide generic development frameworks to facilitate, speed up, and normalize J2EE projects. You can read about JOFFAD at

```
http://joffad.sourceforge.net/structure.html
```

In Example 6.4 you will use the advanced WTP Web project features to develop a Web application using Maven. Maven has a default, but customizable, process that gets a project started using these J2EE best practices quickly. Although both are named a project, a Maven project is conceptually very different from a WTP project.

Maven and Eclipse have overlapping functionality such as compiling, building, and testing. However, Eclipse is normally used for developer-centric coding, testing, and debugging activities, whereas Maven is used for team-centric build management, reporting, and deployment. The primary purpose of Maven is to create a documented, repeatable, and modeled build process that is inclusive of all these activities. It complements the development activities in Eclipse.

You will start by defining a new Web project and organizing the resources in this project according to the best practices suggested by Maven. See

```
http://maven.apache.org/reference/conventions.html
```

for a description of Maven conventions. Maven recommends a standard project directory structure, which is referenced in the Maven Project Object Model (POM). The directory structure of your project will follow Maven conventions (see Example 6.4).

**Manual Operation**

At the time of writing this book, neither WTP nor Maven had tools to create a Maven-style Web project. Therefore, you will manually prepare the project files to make WTP work with the resource structure of Maven-style projects.

**Example 6.4** Maven Project Layout

```
/LeaguePlanetWebProject
+- src/
|  +- main/
|  |  +- java/
|  |  |  +- ...[classes and packages]
|  |  +- resources/
|  |     +- ...
|  |  +- webapp/
|  |  |  +- web-inf/
|  |  |  |  +- classes/
|  |  |  |     |  +- ...[compiled classes]
|  |  |  |  +- lib/
|  |  |  |     |  +- ...[external libraries]
|  |  |  |  +- web.xml
|  |  |  |  +- ...
|  |  |  +- ...[other web files]
|  +- test/
|  |  +- java/
|  |  |  +- ...[test classes and packages]
|  |  +- resources/
|  |     +- ...
|  +- site/
|     +- xdoc/
|        +- ...
+- target/
|  +- ...
+- pom.xml
```

All sources are grouped under the `src` directory. `src/main/java` contains your primary Java classes and packages. `src/test/java` contains your classes

and packages for unit tests. `src/main/webapp`, similar to the WTP `WebContent` folder, contains your Web content, such as the JSP and HTML files, and their supporting resources. `src/site/xdoc` has sources for the project Web site.

To create the Maven project, do the following:

1. Repeat the steps described earlier in this chapter to create a new dynamic Web project named `LeaguePlanetWebProject`. Select a target runtime and default configuration for facets. Click the **Next** button to proceed to the Web module settings (see Figure 6.45).

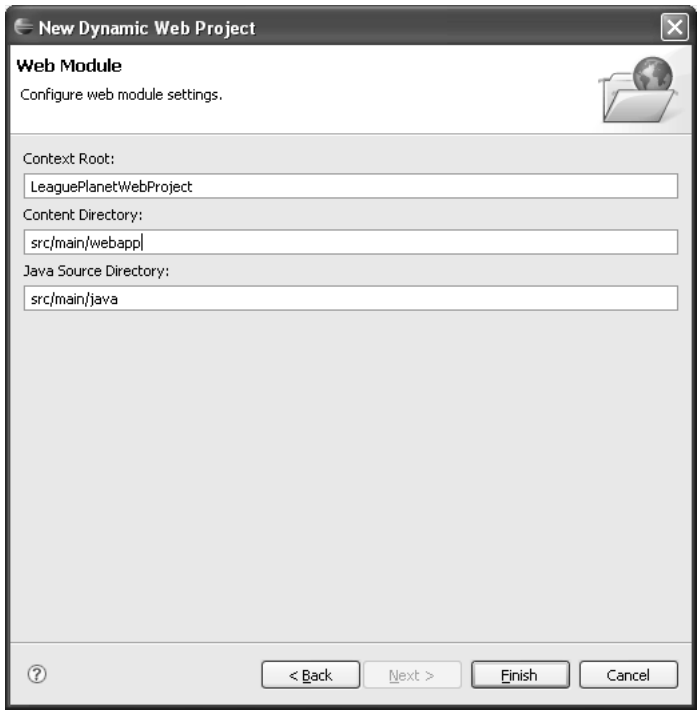


**Figure 6.45** Maven Web Module

2. The **Web Module** page lets you specify the directory for Java resources. This is where you will define locations for the Java sources and Web content. Enter `src/main/webapp` for **Content Directory** and `src/main/java` for **Java Source Directory**. Click **Finish**.
3. WTP creates the Web project, configuration files, deployment descriptor, and so forth.

Once the project is created, the structural model for the Web project is defined as Example 6.5.

**Example 6.5** Structural Model for Maven-Style Web Project

```
<?xml version="1.0" encoding="UTF-8"?>
<project-modules id="moduleCoreId" project-version="1.5.0">
  <wb-module deploy-name="LeaguePlanetWebProject">
    <wb-resource source-path="/src/main/webapp" deploy-path="/" />
    <wb-resource source-path="/src/main/java"
      deploy-path="/WEB-INF/classes" />
    <wb-resource source-path="/src/test/java"
      deploy-path="/WEB-INF/classes" />
    <property name="context-root" value="LeaguePlanetWebProject" />
    <property name="java-output-path" value="build/classes" />
  </wb-module>
</project-modules>
```

**Classpath Management with Maven and WTP**

WTP requires that the `webContent` folder contain the J2EE specification directories `WEB-INF`, `WEB-INF/classes` for the compiled Java classes, and `WEB-INF/lib` for the JARs. All JARs inside this folder are automatically added to the classpath of the project under the **Web App Libraries** category. WTP manages the build path of the project automatically based on the contents of the `WEB-INF` folder.

Maven does not know about your WTP project classpath. It uses dependencies to manage external libraries and code that your project needs. Dependencies are defined in the POM and used to automatically construct a classpath for the Java compiler. Selected libraries are also included in the `WEB-INF/lib` folder. Maven encourages the use of repositories to store and share external libraries, and does not keep them with the project. Instead, Maven retrieves them from a repository when needed. Repositories provide a very consistent and manageable method for maintaining libraries. There is a default Internet-based central Maven repository that keeps most popular Java libraries, served from `ibiblio.org` at

`http://www.ibiblio.org/maven/`

On the other hand, WTP requires that these libraries be kept inside the `WEB-INF/lib` folder. There is code duplication here. In Maven 1.0, dependencies and WTP can coexist in a number of ways. One such method is to use a mechanism to override dependencies per project. This allows you to maintain your external libraries inside the `WEB-INF/lib` folder and override the JAR dependencies. Maven will then retrieve these libraries from your project location instead of the repository. In Maven 2.0, dependencies are always retrieved from a repository.

Let's review what you accomplished so far. You have created a dynamic Web project using the project layout conventions suggested by Maven (see Figure 6.46).

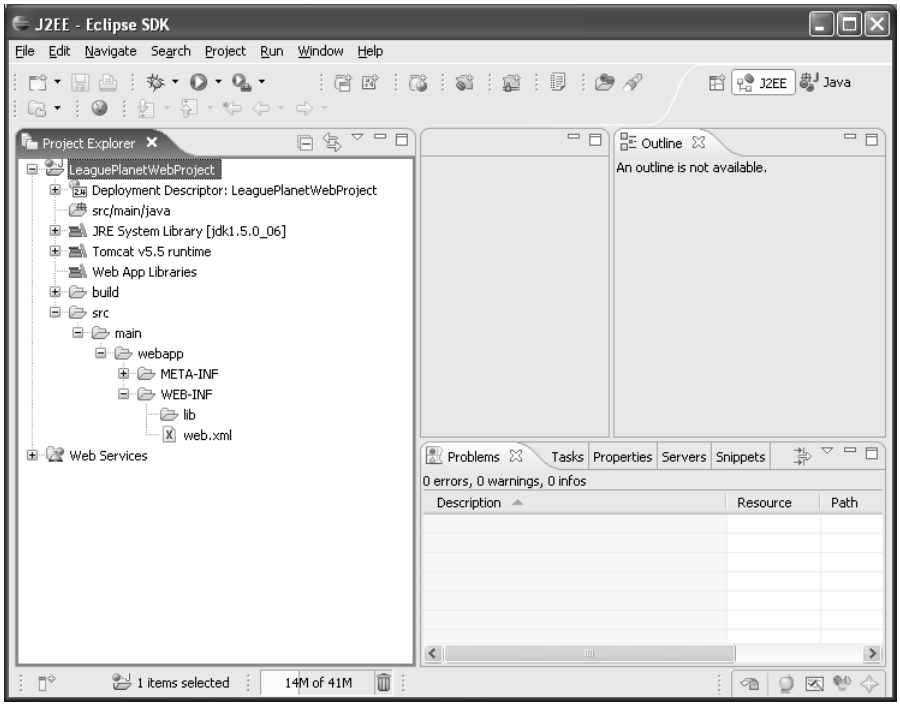


**Figure 6.46** Project Explorer—Maven Web Project

### *Mavenizing the Project*

The next step is defining the Maven POM that will automate builds, unit tests, documentation, project reporting, and so on.

The POM is defined by an XML file named `pom.xml` (see Example 6.6). This file tells Maven everything that it needs to know about your project. Maven has tools that can create skeleton POMs, but we will create the POM from scratch. The snippet shown in Example 6.6 is the start of a POM for your Web application.

**Example 6.6** Content of POM

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>leagueplanet</artifactId>
  <groupId>com.leagueplanet</groupId>
  <name>LeaguePlanet.com Web Project</name>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>
  <build>[...]</build>
  <dependencies>[...]</dependencies>
</project>
```

The project `artifactId` corresponds to the Web application module in your project. Dependencies will define external libraries needed by your Web application. You will use the Struts framework, so `struts*.jar` and `commons*.jar` libraries must be present in this list. The build section tells Maven how the Java sources and other resources are organized. Maven project definition allows you to define filters for including or excluding source files.

The build section is quite simple to set up, as shown in Example 6.7.

**Example 6.7** Maven Build Section

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  [...]
  <build>
    <finalName>${artifactId}-${version}</finalName>
  </build>
</project>
```

The build section can be used to customize your project. Since you used the default location, you do not have to modify anything here. The `finalName` element automatically constructs the name of the exported WAR from other information provided in the POM.

The dependency section is probably the longest (see Example 6.8).

**Example 6.8** Maven Dependencies Section

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  [...]
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>struts</groupId>
      <artifactId>struts</artifactId>
      <version>1.2.7</version>
    </dependency>
    <dependency>
      <groupId>struts</groupId>
      <artifactId>struts-el</artifactId>
      <version>1.2.7</version>
    </dependency>
    <dependency>
      <groupId>commons-validator</groupId>
      <artifactId>commons-validator</artifactId>
```

```
      <version>1.1.4</version>
    </dependency>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.0.3</version>
    </dependency>
    <dependency>
      <groupId>commons-fileupload</groupId>
      <artifactId>commons-fileupload</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>antlr</groupId>
      <artifactId>antlr</artifactId>
      <version>2.7.5</version>
    </dependency>
    <dependency>
      <groupId>commons-digester</groupId>
      <artifactId>commons-digester</artifactId>
      <version>1.7</version>
    </dependency>
    <dependency>
      <groupId>commons-beanutils</groupId>
      <artifactId>commons-beanutils</artifactId>
      <version>1.7.0</version>
    </dependency>
    <dependency>
      <groupId>oro</groupId>
      <artifactId>oro</artifactId>
      <version>2.0.8</version>
    </dependency>
    <dependency>
      <groupId>servletapi</groupId>
      <artifactId>servletapi</artifactId>
      <version>2.3</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```

Each entry corresponds to an external JAR that is needed by your project. The Struts framework requires a few of these dependencies to be set. Some of these JARs are needed to compile your code; others, such as JUnit, are for testing. The JARs have a scope tag that defines when they are used. For example, by default all Struts JARs will be included with the Web application module, but JUnit has the scope `test`, so it will not be included.

Remember that Maven gets the libraries defined in the dependencies from a repository. However, for WTP to function properly, you need to keep a copy of these libraries inside the `src/webapp/WEB_INF/lib` folder instead of the repository. Unfortunately, there is no tool to synchronize the dependencies and libraries.

You have defined the minimal Maven POM to build your Web application. Maven is typically run from the command line. Maven commands are also called *goals*. Goals are high-level tasks that can include other subtasks. Mevenide is an Eclipse plug-in for Maven that allows you to run Maven goals from the Eclipse IDE. Here you will use the command line. You can build a deployable Web module and a project site by running the maven `clean package site` goals. The `package` goal depends on other goals such as `compile` and `test`, so Maven will run them automatically. During the build, Maven creates a folder named `target` to store the generated files. The name and location of the generated files can be modified by additional settings. When you run Maven, you will get an output like that shown in Example 6.9.

**Example 6.9** Maven Console Output

```
C:\workspace\LeaguePlanetWebProject>mvn clean package site
[INFO] Scanning for projects...
[INFO] ------------------------------
[INFO] Building LeaguePlanet.com Web Project
[INFO]    task-segment: [clean, package]
[INFO] ------------------------------
[INFO] [clean:clean]
[INFO] Deleting directory
  C:\workspace\LeaguePlanetWebProject\target
[INFO] Deleting directory
  C:\workspace\LeaguePlanetWebProject\target\classes
[INFO] Deleting directory
  C:\workspace\LeaguePlanetWebProject\target\test-classes
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[WARNING] While downloading servletapi:servletapi:2.3
  This artifact has been relocated to javax.servlet:servlet-api:2.3.

[INFO] [compiler:compile]
Compiling 1 source file to
C:\workspace\LeaguePlanetWebProject\target\classes
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
Compiling 1 source file to
  C:\workspace\LeaguePlanetWebProject\target\test-classes
[INFO] [surefire:test]
[INFO] Setting reports dir:
  C:\workspace\LeaguePlanetWebProject\target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
[surefire] Running com.leagueplanet.tests.LeaguePlanetBVTTest
[surefire] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.01 sec
[INFO] [site:site]
[INFO] Generate "Continuous Integration" report.
[ERROR] VM #displayTree: error : too few arguments to macro. Wanted 2 got 0
[ERROR] VM #menuItem: error : too few arguments to macro. Wanted 1 got 0
[INFO] Generate "Dependencies" report.
```

```
[INFO] Generate "Issue Tracking" report.
[INFO] Generate "Project License" report.
[INFO] Generate "Mailing Lists" report.
[INFO] Generate "Source Repository" report.
[INFO] Generate "Project Team" report.
[INFO] Generate "Maven Surefire Report" report.
[INFO] Generate an index file for the English version.
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------
[INFO] Total time: 11 seconds
[INFO] Finished at: Sat May 13 15:48:09 EEST 2006
[INFO] Final Memory: 9M/17M
[INFO] ------------------------------------------------------------
```

That is all there is to building a WAR with Maven. You will see from the log that `package` is a composite goal. In addition to assembling a Web application module using the `war` goal, it runs the `java` goal to compile the classes and the `test` goal to compile and run the tests. Once the build is complete, you can browse the results of the build in the `target` folder (see Figure 6.47).

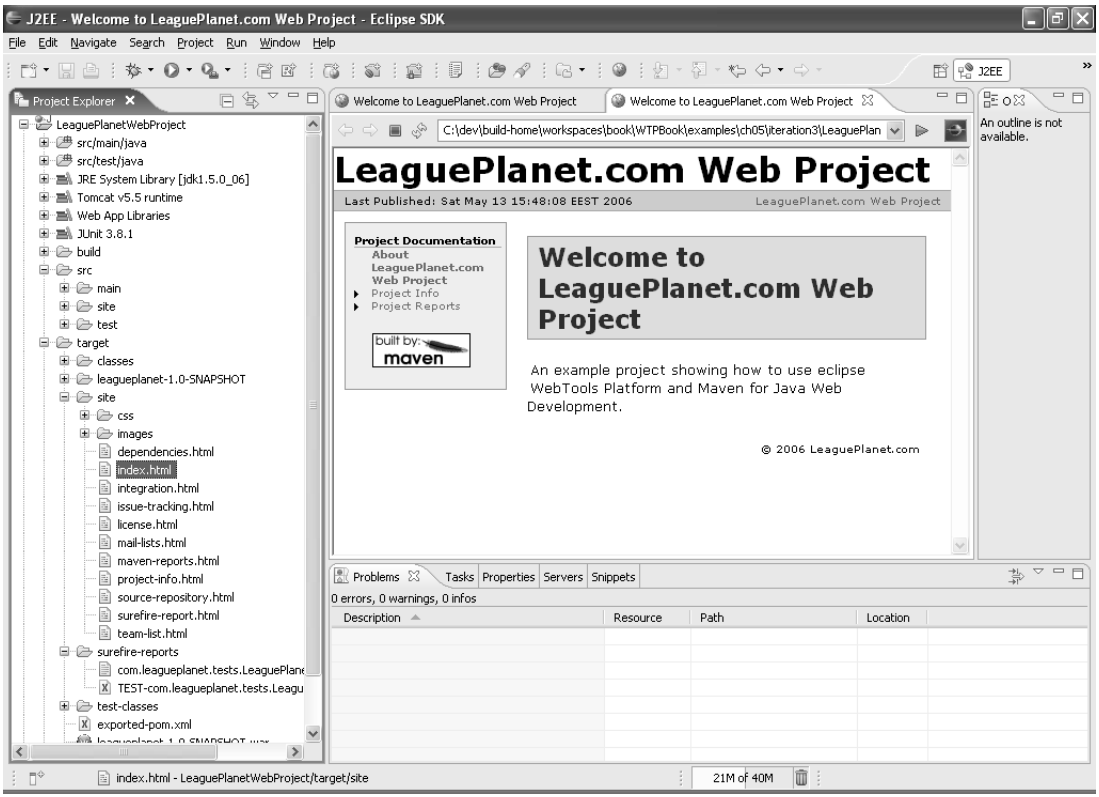


**Figure 6.47** Project Site

So far, you could have done most of this using WTP, without the hassle of setting up Maven in the project. Building Web application modules is something WTP does well, and it does it automatically with minimal effort. But you can get more out of Maven. The next section shows you how to automate testing and reporting on the League Planet project using Maven.

### *Getting More Out of Maven*

Now that you can build the Web application module using Maven, you can add tests and more project information to the POM to find out what more Maven can do.

#### Unit Tests with Maven

To run unit tests with Maven, you will create JUnit test cases and define required libraries, including JUnit in the project dependencies. Since you defined the JUnit dependencies in the previous section, you can start writing a test in the `src/tests/java` source folder. In the Project Explorer, select `LeaguePlanetWebProject`, right click, and invoke the File ▸ New ▸ Source Folder. The New Source Folder wizard will open (see Figure 6.48).

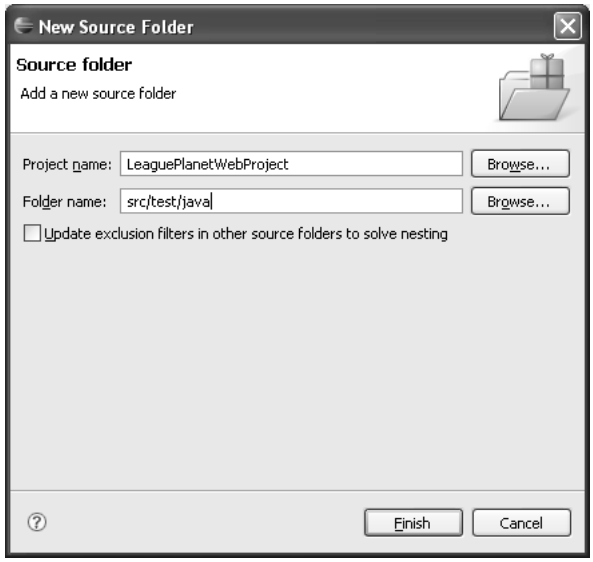


**Figure 6.48** Source Folder for Tests

Enter `src/test/java` as the folder name. Click Finish. A new source folder will be added to the project.

To create a new JUnit test case, invoke the JUnit test case wizard using File ▸ New ▸ JUnit Test Case, and then enter package and class names, for example,

`com.leagueplanet.tests` and `LeaguePlanetBVTTests`. Click **Finish**. The wizard will prompt you to add `junit.jar` to the project build path if it is not included there already. Accept it to add the JAR.

**JUnit JAR Is Defined Twice**

Remember that Maven does not know about your project classpath. Therefore it will not know about the JUnit JAR unless it is added to the POM dependencies. You add JUnit to the dependencies as shown in Example 6.10.

**Example 6.10** Maven JUnit Dependency

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>3.8.1</version>
  <scope>test</scope>
</dependency>
```

This is inconvenient, but it is something that you will have to live with if you want to use Maven.

A new test class will be created at the location shown in Example 6.11.

**Example 6.11** Maven Test Directory

```
/LeaguePlanetWebProject
+- src/
|  +- test/
|  |  +- java/
|  |  |  +- com
|  |  |  |  +- leagueplanet
|  |  |  |  |  +- tests
|  |  |  |  |  |  +- LeaguePlanetBVTTests.java
|  |  |  |  |  |  +- [...] other unit tests
|  [...]
```

Execute the Maven `package site` goals to run the tests. If you want to run the tests only, you just execute the `test` goal. Running the Maven `test` goal creates output as shown in Example 6.12.

**Example 6.12** Maven Test Output

```
[INFO] Scanning for projects...
[INFO] ------------------------------
[INFO] Building LeaguePlanet.com Web Project
[INFO]    task-segment: [test]
[INFO] ------------------------------
[INFO] [resources:resources]
```

```
[INFO] Using default encoding to copy filtered resources.
[WARNING] While downloading servletapi:servletapi:2.3
  This artifact has been relocated to javax.servlet:servlet-api:2.3.

[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test]
[INFO] Setting reports dir:
  C:\workspace\LeaguePlanetWebProject\target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
[surefire] Running com.leagueplanet.tests.LeaguePlanetBVTTest
[surefire] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.03 sec
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Sat May 13 15:58:43 EEST 2006
[INFO] Final Memory: 3M/6M
[INFO] ------------------------------------------------------------
```

You will find the Maven JUnit test reports under the `target/surefire-reports` folder. Of course, XML reports can be transformed into a more human-readable format, but you will see in the next section that Maven also does this for you (see Figure 6.49).

#### Project Information and Reports

The Maven project model can also contain information about the developers, configuration and version control systems, issue tracking, mailing lists, and other process-related project information. This information is used by Maven plug-ins to generate project information and reports. The listing shown in Example 6.13 provides the complete code for a typical Maven project model.

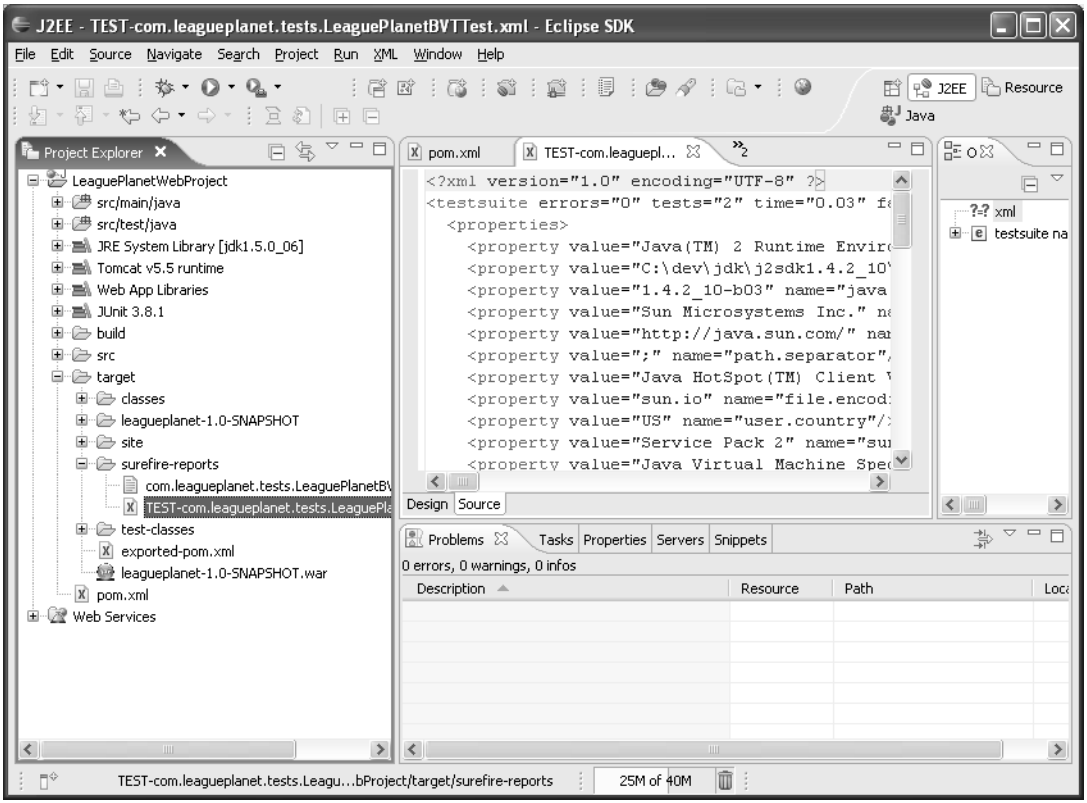


**Figure 6.49** Maven JUnit Test Reports

**Example 6.13** Listing of pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>leagueplanet</artifactId>
  <groupId>com.leagueplanet</groupId>
  <name>LeaguePlanet.com Web Project</name>
  <version>1.0-SNAPSHOT</version>

  <packaging>war</packaging>

  <organization>
    <name>LeaguePlanet.com</name>
    <url>http://www.leagueplanet.com/</url>
  </organization>
```

```
<description>
   An example project showing how to use eclipse WebTools Platform
   and Maven for Java Web Development.
 </description>

 <licenses>
   <license>
     <comments>Eclipse Public Licence (EPL)v1.0</comments>
     <url>http://www.eclipse.org/legal/epl-v10.html</url>
   </license>
 </licenses>

 <developers>
   <developer>
     <id>ndai</id>
     <name>Naci Dai</name>
     <email>naci.dai@eteration.com</email>
     <organization>Eteration</organization>
   </developer>
   <developer>
     <id>lmandel</id>
     <name>Lawrence Mandel</name>
     <email>lmandel@ca.ibm.com</email>
     <organization>IBM</organization>
   </developer>
   <developer>
     <id>ryman</id>
     <name>Arthur Ryman</name>
     <email>ryman@ca.ibm.com</email>
     <organization>IBM</organization>
   </developer>
 </developers>

 <build>
   <finalName>${artifactId}-${version}</finalName>
 </build>
 <dependencies>
   <dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>3.8.1</version>
     <scope>test</scope>
   </dependency>
   <dependency>
     <groupId>struts</groupId>
     <artifactId>struts</artifactId>
     <version>1.2.7</version>
   </dependency>
   <dependency>
     <groupId>struts</groupId>
     <artifactId>struts-el</artifactId>
     <version>1.2.7</version>
   </dependency>
   <dependency>
     <groupId>commons-validator</groupId>
```

```
      <artifactId>commons-validator</artifactId>
      <version>1.1.4</version>
    </dependency>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.0.3</version>
    </dependency>
    <dependency>
      <groupId>commons-fileupload</groupId>
      <artifactId>commons-fileupload</artifactId>
      <version>1.0</version>
    </dependency>

    <dependency>
      <groupId>antlr</groupId>
      <artifactId>antlr</artifactId>
      <version>2.7.5</version>
    </dependency>
    <dependency>
      <groupId>commons-digester</groupId>
      <artifactId>commons-digester</artifactId>
      <version>1.7</version>
    </dependency>
    <dependency>
      <groupId>commons-beanutils</groupId>
      <artifactId>commons-beanutils</artifactId>
      <version>1.7.0</version>
    </dependency>
    <dependency>
      <groupId>oro</groupId>
      <artifactId>oro</artifactId>
      <version>2.0.8</version>
    </dependency>
    <dependency>
      <groupId>servletapi</groupId>
      <artifactId>servletapi</artifactId>
      <version>2.3</version>
    </dependency>
  </dependencies>
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>
          maven-project-info-reports-plugin
        </artifactId>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-report-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
</project>
```

The project reports are generated using the Maven `site` goal. This goal builds a local copy of the project site for reports, documentation, and reference. The result is generated into the `target/site` directory in the project's base directory, which contains an entire Web site of documentation (see Figure 6.50).

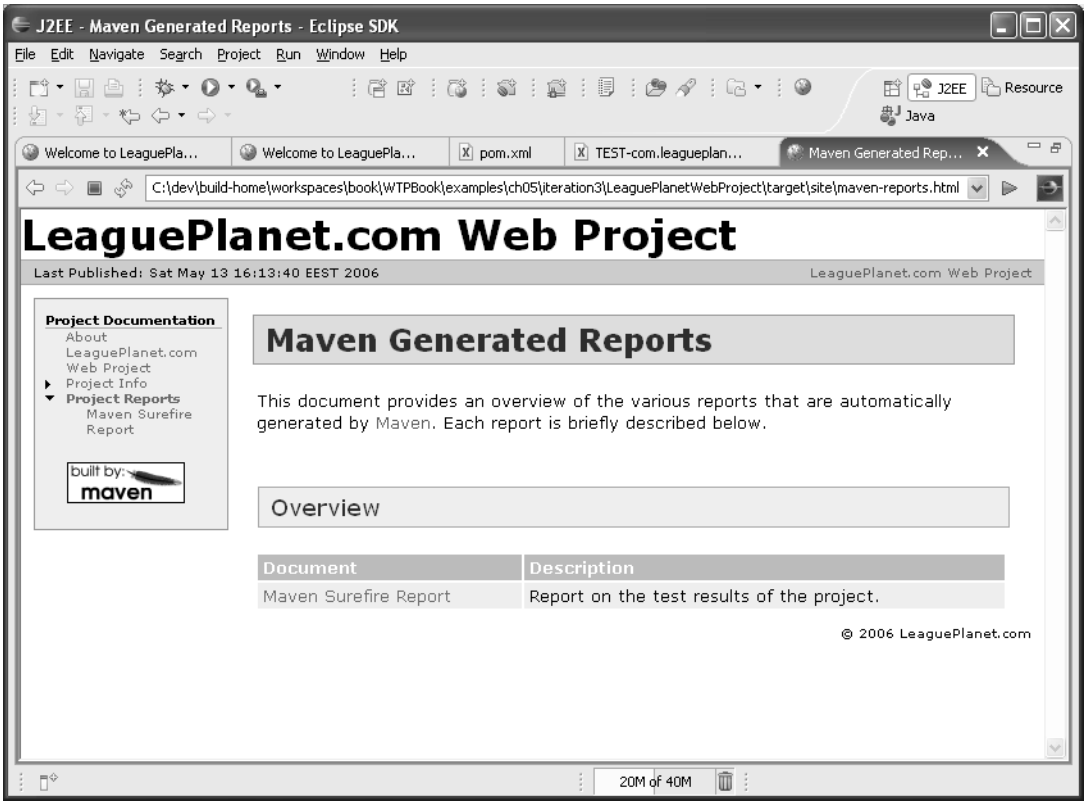


**Figure 6.50** Maven Project Reports

## Summary

We have described how modules and projects are managed in WTP. You now should have enough knowledge to start exploring these project styles and customizing them as you see fit.

Web projects are very flexible, but they can't model every style of project that you can imagine or that is in use somewhere. Our advice to you is to use one of the more popular templates, such as the default ones created by WTP, or widely published conventions such as Maven. You can build on top of existing know-how

and make use of the experience that is readily available. When you are organizing your development, the last thing you want to be is surprised, so do take advantage of well-established best practices.

You can now proceed to either Chapter 7, which covers the presentation layer, or Chapter 8, which covers business logic, depending on the type of development you want to start.