SAMPLE CHAPTER

IntelliJ IDEA IN ACTION

Duane K. Fields
Stephen Saunders
Eugene Belyaev

MANNING

IntelliJ IDEA in Action

by Duane K. Fields
Stephen Saunders
Eugene Belayev
with
Arron Bates

Sample Chapter 4

brief contents

4 Managing projects

In this chapter...

- Creating and managing projects in IDEA
- Using modules to create efficient projects
- Using libraries to manage third-party components

Work in IDEA begins with the concept of a project. A project encapsulates all your source code, library files, and build instructions into a single organizational unit. Since version 4.0, IDEA's modules and libraries let you segregate larger, complex projects in more manageable structures that can share common code. Modularized projects are also a great benefit when you're building enterprise applications composed of several different components with complex interdependencies. Even if you've used an older version of IDEA extensively, we recommend that you not skip this chapter due to the fundamental changes in project management.

4.1 Understanding IDEA's project strategy

When you work on source code in IDEA, you do so in the context of a *project*. Because everything in IDEA revolves around the project, it's important to have a firm understanding of how and why IDEA handles projects the way it does.

4.1.1 Examining the IDEA project hierarchy

If you've never used an integrated development environment (IDE) like IDEA before, you may not immediately understand why you have to define a project before diving into your work. Remember, however, that IDEA isn't a simple text editor; it's a Java development environment. As such, you can't just start typing in your source code willy-nilly. First you have to create a project.

What is a project?

A project in IDEA is an organizational unit that represents a complete software solution. Your finished product may be decomposed into a series of discrete, isolated modules, but it's a project definition that brings them together, relates them with dependencies, and ties them into a greater whole.

Projects don't themselves contain development artifacts such as source code, build scripts, or documentation. They're the highest level of organization in the IDE, and they define project-wide settings as well as collections of what IDEA refers to as *modules* and *libraries*.

What is a module?

A *module* in IDEA is a discrete unit of functionality that can be run, tested, and debugged independently. Modules contain the development artifacts for their specific task; this includes such things as source code, build scripts, unit tests, documentation, and deployment descriptors. IDEA supports different types of modules, from plain Java applications to web apps, EJB modules, and so on. For many projects, a single module will suffice.

What is a library?

A *library* is an archive of compiled code that your modules depend on. Such an archive is typically represented as a JAR file or an expanded JAR in a directory. Libraries may optionally contain references to source files and API documentation: Including these references doesn't alter the usage of the library in any way, but it does add valuable information to the editor during class navigation and inspection. Examples of libraries include a DBMS vendor's private JDBC driver, or an open-source XML parser.

4.1.2 Selecting different types of modules

IDEA provides four distinct types of modules, which fall into two categories: basic Java modules and enterprise Java (or J2EE) modules. This chapter covers the first category; we'll reserve discussion of J2EE modules until chapter 11. As a head start, here's the purpose of each of the available module types shipped with IDEA:

IDEA 5

With version 5.0, this list has been expanded. Now there are six types of modules, roughly categorized into basic/standard Java (J2SE) modules, J2EE modules, and J2ME modules.

- *Java modules* are the simplest module type and represent a basic Java application project, whether it's a command-line tool, a Swing application, or a JAR library. When configuring this type of module, you can specify a set of Java source paths that will be compiled to a single class folder. We'll discuss using and configuring this type of module in detail in this chapter. The basic capabilities of this module are carried over into the web module.
- The *web module* is an extension of the Java module that adds support for web applications. In addition to providing the ability to create and build Java sources, it lets you edit your web application's deployment descriptor, build and deploy it to your application server, and configure other web application capabilities. You create a web module for each web application in your project. Web modules will be discussed in detail in chapter 11.
- An EJB *module* lets you design and package a collection of Enterprise JavaBeans. EJB modules will be discussed more fully in chapter 11.
- A J2EE *application module* is different than the other module types discussed so far. The J2EE module type is primarily concerned with packaging J2EE applications for deployment as enterprise archive (EAR) files. As such, it references Web and EJB modules that it packages for deployment.

IDEA 5

IDEA version 5.0 comes with two new module types: J2ME modules and IntelliJ Plugin modules. A J2ME module is a module suited for working on micro applications (such as for mobile technologies), and an IntelliJ Plugin module provides you with a correctly configured module for developing your own IDEA extensions.

4.1.3 Selecting a project structure

Because it's the module that defines a set of source files, a typical project must be composed of at least one module (you can create a project with no modules, but it's useless until the first module is added). For many projects, a single module is all you need. For more complex projects, especially J2EE projects or software suites composed of several discrete applications, a multi-module structure is more convenient. Separate modules let you build and test each piece separately while maintaining a common configuration. You obtain three primary benefits from breaking your project into modules:

- Reusability and sharing of modules between projects
- Improved project structure
- Module specific features

One benefit of the IDEA's modular projects is that a module can be shared among several projects if the need arises. Take, for example, a collection of utility classes that you'd like to share among several different types of projects. By putting them into their own module, you can easily add it to your other projects while maintaining the ability to develop it independently.

Modules can be built, tested, and versioned independently, so they're a great way to reduce the complexity of large projects. You can choose to compile and test a single module for example, without waiting on the rest of the application to be built. In addition, you can take a single module from many in a complex project and place it into a second project by itself, allowing you to remove the overhead and distraction of the larger project.

Modules in IDEA come in several different flavors, each designed with a particular type of application in mind. These application-specific modules extend the capabilities of IDEA to support new types of applications and to assist in their development and deployment. Web modules offer one-click deployment, whereas the J2EE module packages your application into an EAR file. No doubt future releases and third-party extensions to IDEA will add new types of modules.

Imagine you're building a Microsoft Office–style suite of office applications consisting of spreadsheet, word processor, and presentation designer applications. One natural way to model this project is to create a separate module for each application in the suite along with a module representing a set of utility functions common to all three applications.

4.2 Working with projects

Project creation is simplified with the assistance of the **New Project Wizard**, a process you've already run through with the creation of your "Hello World" and ACME projects. Things have a habit of changing over time, however, and it's a rare project that doesn't need some sort of reconfiguration during its lifetime. Ongoing project maintenance is handled through IDEA's **IDE Settings** window, which lets you change almost every aspect of the project you're working in.

4.2.1 Creating a new project

You create a new project by selecting the **File | New Project** command to launch the project wizard. The project wizard takes you through the steps required to set up a basic project and, if you desire, set up the project's initial module. When you first launch IDEA, it automatically directs you to the **New Project Wizard**.

IDEA 5

In version 5.0, the number of steps (as well as the order of the steps) of the New Project creation wizard may not match those described here for version 4.5.

Specifying the name and location of the project

In the first panel of the **New Project Wizard** (see figure 4.1), you're asked to specify the name of the project and the folder where the project file will be created. The name of the project file will be the name of the project plus the `.ipr` extension, so if you have specific requirements concerning spaces (or lack there of) in your filenames, take appropriate action. There is no requirement that your project file be located anywhere in particular, but you must consider several factors when choosing a location:

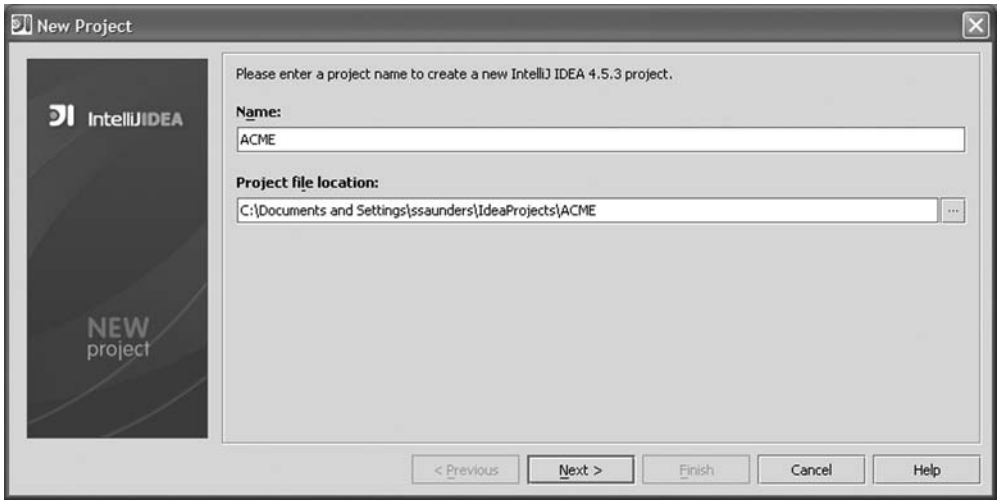

Figure 4.1 Step 1 of the New Project Wizard involves choosing a project name and a location to store the IPR file.

- You'll have fewer problems sharing or relocating your project if you keep all of your project's file and components at the same level or below the project file itself.
- If you'll be maintaining your project file in a source control system, it's convenient to place the project file in the root of your source control project.
- Keeping all of your project files in the same folder makes it easier to find and access them; likewise for your module files. However, doing so may be less convenient with regard to your source code control system.

If the folder you specify for the project doesn't exist, the wizard asks you if you wish to create it. Remember that you're specifying the folder where the project file will be created, not the name of the project file itself (this is created automatically and is always the same as your project name). When you're ready, click **Next** to continue.

Selecting a project JDK

In the second step of the **New Project Wizard** (see figure 4.2), you're asked to specify which JDK to use for the project. (Refer to chapter 1 for instructions on configuring JDKs for use within projects.) Make your selection, and click **Next**.

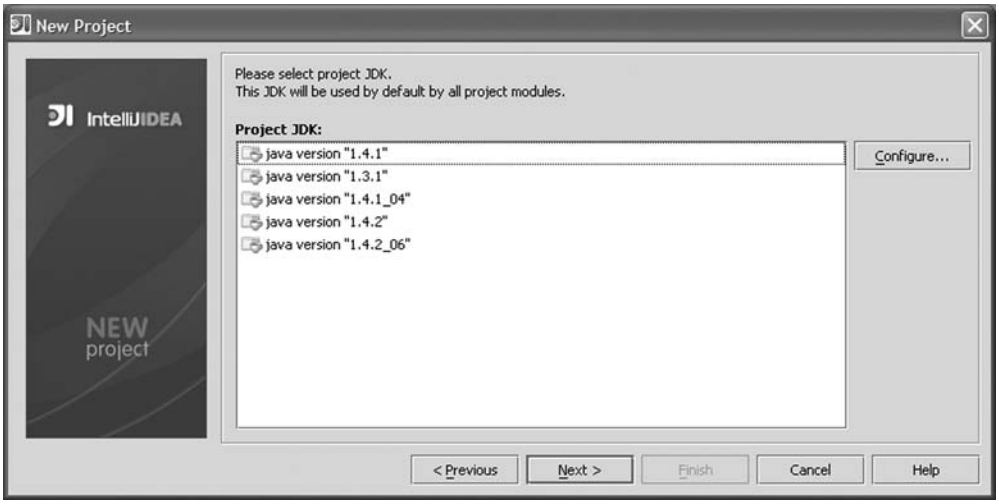

Figure 4.2 Step 2 of the New Project Wizard involves selecting a target JDK for the project.

Your choice of JDK determines which compiler and API library are used to build your project, unless a module specifically overrides this setting.

Selecting a single or multi-module project

At this point, the **New Project Wizard** has everything it needs to create a new project. However, without any modules, a project is just an empty shell and not very useful. This step lets you create your initial module (through the wizard) by selecting the first option **Create single-module project** and clicking **Next** (see figure 4.3).

If you want to create a multi-module project, or if you want an empty project to which you'll add modules later, select the second option **Create/configure multi-module project**; doing so changes the **Next** button to a **Finish** button. Click **Finish** to exit the wizard. The project is created, and you're taken to the **Add Module Wizard** to create or import modules. Creating and importing modules into your project is covered in the next section of this chapter.

4.2.2 Managing project settings

Along with its global settings and preferences, IDEA maintains a collection of settings specific to each project you define. These settings define not only the project's contents and behavior but also related information such as your source code control settings and compiler behavior. You can access these settings by

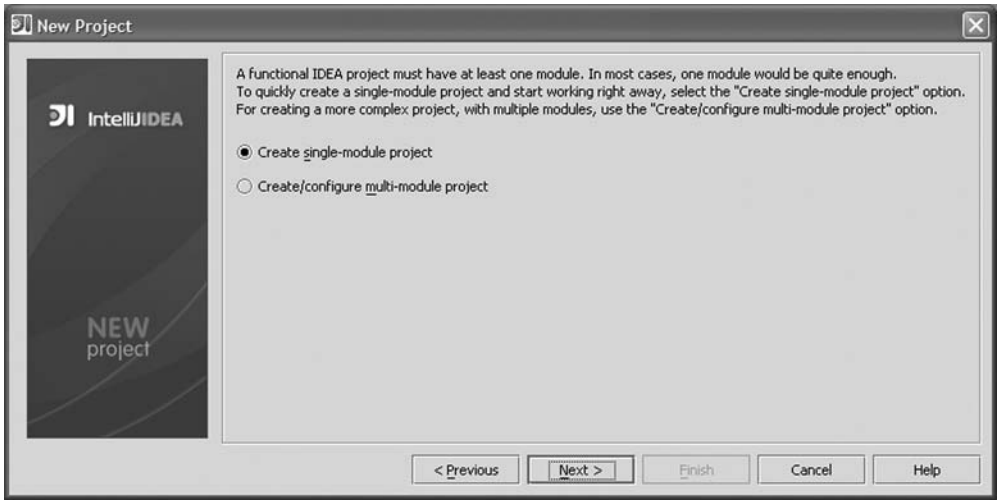

Figure 4.3 Step 3 of the New Project Wizard lets you choose between a single-module and a multi-module project.

selecting **File | Settings**, by using the shortcut **Ctrl+Alt+S**, or by clicking the **Settings** icon in the main toolbar (the wrench and machine nut). Doing so brings up the **Settings** control panel, shown in figure 4.4. The project-specific categories for the currently active project are shown at the top of the panel; global IDE settings are listed in the lower half. We'll cover the details of customizing IDEA through the settings panels in chapter 12.

The project-specific options that you can specify include defining the list of modules involved in the project, the compiler to use for building the project (and some of its options), the version control system configuration for the project, the code style to which the project adheres, and a few options controlling the behavior of the IDEA GUI Designer.

Configuring paths

The **Paths** settings panel is the main control panel for configuring your project. However, as you can see in figure 4.5, this control panel is sparse when no modules are present (you'll see another screenshot of this window shortly with modules in it, for comparison). This is because the modules, not the project, manage the development artifacts like source folders, dependent libraries, and so forth. The module list lets you add and remove modules from the project; it's discussed in the next section.

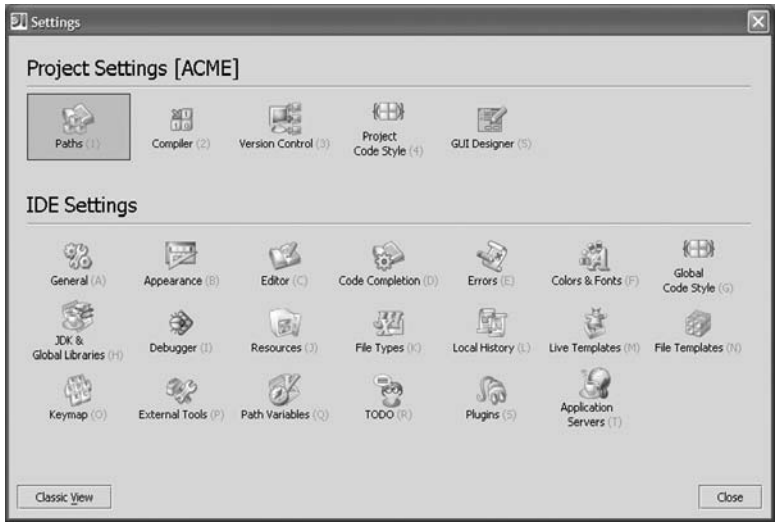

Figure 4.4 The Settings window lets you control project-specific settings as well as general IDE settings. The two categories are separated for convenience, as shown here.

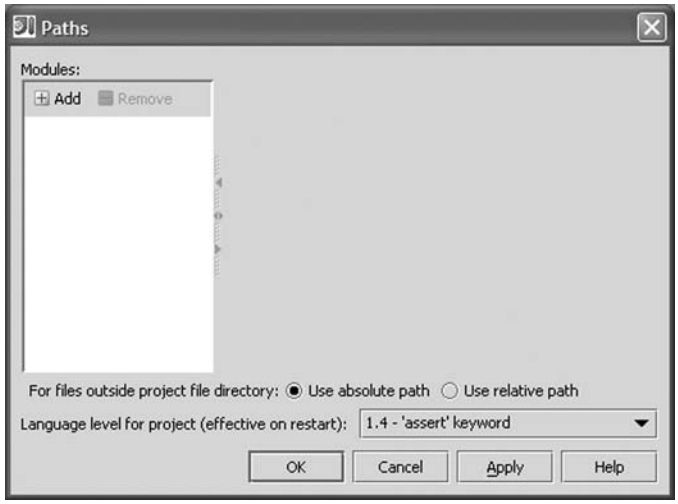

Figure 4.5 The Paths window lets you control a few project-specific settings, as shown here, but it's principally used to configure the paths of the project's included modules (for a project with modules defined, see figure 4.13).

The first option allows you to select between absolute and relative paths. If a module, library, or other referenced file is located outside the project file's directory, IDEA uses either the absolute path or the relative path (relative to the project file) to reference them, depending on which option you've selected in the project's path options. For files within the project directory, relative paths are always used to make project files as portable as possible between machines. This setting is also available on a module-by-module basis, should that become necessary.

The other setting lets you to configure the language level for the project. The default language level is 1.3, but you can use 1.4 or 1.5. Language level 1.4 enables the `assert` keyword, introduced in JDK 1.4. By default, Java compilers disable this keyword, because it wasn't a reserved keyword prior to JDK 1.4 and may cause conflicts with older source code. Similarly, language level 1.5 enables some of the new JDK 1.5 features, such as the `enum` keyword and autoboxing. If you alter the language level, you must restart IDEA in order for the option to take affect.

Configuring compiler settings

The **Compiler** settings panel lets you control build-related options such as whether to generate warnings, passing additional parameters to the compiler, and so forth. The details of the settings configured through this panel will be covered in chapter 5.

Configuring version control settings

The **Version Control** settings panel lets you integrate your version control system with your project. Because this is a project-level setting, you're free to use different source code control systems for different project or tweak the behavior of each from project to project. Note that IDEA also maintains a set of global settings for version control shared from project to project, such as the location of your source code repository. These settings are also accessed via this panel. Using and configuring your version control system with IDEA is covered in chapter 8.

Configuring project code style settings

The **Project Code Style** settings panel lets you override the code layout and formatting styles used for this project. IDEA maintains a detailed set of code formatting options that are shared between projects unless overridden through this panel. The **Code Style** settings let you specify everything from the size of your indents to the spacing around method calls and operator symbols. Code style settings are covered in chapter 12.

Configuring GUI Designer settings

The options in the **GUI Designer** settings panel pertain to IDEA's GUI Designer tool, which makes building user interfaces in Swing relatively painless. Using the GUI Designer and configuring its options is covered in chapter 10.

Configuring project template defaults

When you create a new project, the initial settings are based on project template settings maintained by IDEA. To edit these settings, select the **File | Template Project Settings** command to reveal a subset of the **Settings** panel (figure 4.6). IDEA lets you specify default settings for the compiler, version control, code style, and GUI designer options. Once set, all new projects begin with these default settings. Because the options in these panels are identical to those used when configuring an existing project, we'll defer discussing them in detail until we come to their usage throughout the book.

Saving your project settings

IDEA automatically saves project settings, so there is no need (or opportunity) to explicitly save your project. The newly defined settings are applied immediately, so you don't need to restart IDEA or close and reopen project to make them effective.

Reopening a project

When you start IDEA, it automatically reopens the last project you worked on, unless you've disabled the **Reopen last project on startup** option under IDEA's **General** settings. You can open an existing project by selecting the **File | Open Project** command and selecting the IPR file corresponding to your project. Or, if

Figure 4.6 The Template Project Settings window lets you specify defaults for most project-specific settings. These defaults are applied to all new projects you create.

you've used the project recently, it's listed in the **File | Reopen** submenu, which maintains a list of the most recently used projects.

Working with multiple projects

When you attempt to work on a project while one is already open, IDEA asks if you wish to open the project in a new frame. If you want to work on multiple projects simultaneously, click **Yes**. Otherwise, click **No** to close your existing project and open the new one.

When you open multiple projects, each is loaded in its own IDEA frame. The two projects are completely independent; other than letting you cut and paste between the application windows, they can't share data. To close a project, select the **File | Close Project** menu option.

WARNING Regardless of how many projects you have open at once, only a single instance of IDEA is running, so all open projects must share the same memory space. You may need to bump up the amount of memory allocated to IDEA if you plan to frequently have multiple projects open at once.

4.2.3 Working with project files

IDEA stores the configuration data for projects and their components in plaintext XML files, making it easy to manage, edit, and share project configuration data with others. IDEA creates three different types of files: the project file, the workspace file, and the module file.

Project files have an `.IPR` extension and contain information core to the project itself, such as the names and location of its component modules, compiler settings, Ant configurations, and so forth. You can click an `.IPR` file to launch the project in IDEA. By default, this file is created at the root of the project.

Along with each project file, IDEA creates an `.IWS` file to store your personal workspace settings. This file remembers the placement and positions of your windows, your VCS and History settings, your Run/Debug configuration targets, and other data pertaining to the development environment. This file is always created alongside your project file. If this file is deleted, it's regenerated automatically, unlike the project `.IPR` file.

Module files are created for each module you defined and have the `.IML` extension. The module file stores all the path and dependency information associated with the module. Its exact contents depend on what type of module it is. By default, module files are located in the module's content root folder.

4.3 Working with modules

Although the project may be the center of attention, the module does all the work. Without a module, your project has no source code, no output, nothing other than a collection of configuration preferences. Like projects, modules often require alterations after their initial creation. Their participation in a project, as well as their individual internal settings, is controlled through IDEA's Settings interface.

4.3.1 Managing project modules

Modules are managed from within the project's **Paths** settings panel. The list of modules associated with the project is shown along the left side of the panel. IDEA gives you options for adding, removing, and editing the list of modules.

Adding a new module to your project

When you create a new project, the project wizard gives you the opportunity to define your first module as part of the project setup. Alternatively, to add new modules after the initial project creation, select **File | New Module**. Either way, the process is the same, beginning with the **Add Module Wizard**, shown later in figure 4.9.

Importing a module into your project

In the first step of the **Add Module Wizard** you have two options: You can create a new module from scratch or import an existing one into your project by selecting the **Import existing module** radio button. Select the path to the module's `.IML` file by either typing it in or clicking the ellipsis button to select it via the file requestor.

When you import an existing module, you're really just adding a reference to it. The module isn't copied into the project's root folder or altered in any way. The module is always stored in a single location. This allows a single module to be shared by multiple projects.

Since version 5.0, IntelliJ IDEA provides the ability to automatically convert Eclipse and JBuilder projects into its own format.

Removing modules from the project

You can remove a module from a project by selecting it from the module list and then clicking the minus button in the toolbar. This only removes the module reference from the project and doesn't delete or affect the module file or its contents. You can add the module again by using the **Add Module Wizard** and selecting the **Import** option.

Managing dependencies between projects

When you're building a multi-module project, you may end up with modules that rely on the output of one or more of the other modules in the project. For example, a module representing the installation artifacts and scripts for a packaged product may depend on the module representing that product. In order for IDEA to build your project successfully, you must specify all such dependencies. Click the **Dependencies** tab at the top of the **Paths** settings window for the dependent module, as shown in figure 4.7.

All the other modules in the project are listed, along with a checkbox for each. Select the modules the currently selected module relies on to build or function properly. Any attempt to build the selected module automatically triggers builds of its dependencies. Likewise, a change to one of the module's dependencies triggers a rebuild of the dependent module.

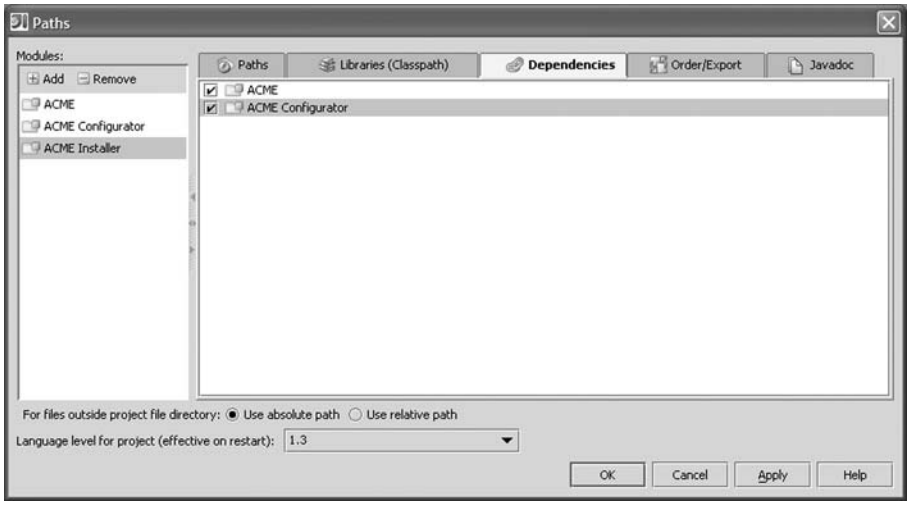

Figure 4.7 Dependencies between modules need to be specified so that IDEA can determine which modules require rebuilding when the build command is invoked.

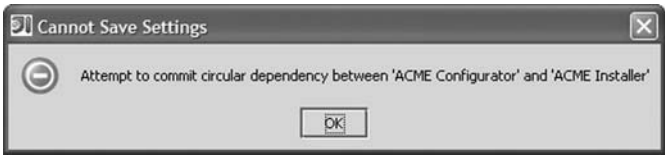

Figure 4.8 Circular dependencies make projects impossible to build, so IDEA prevents you from defining them.

Circular dependencies aren't allowed: Module A, for example, can't depend on Module B if Module B depends on A. Larger circular dependencies are likewise not permitted. If you create a circular dependency, IDEA warns you and keeps you from applying your changes, as shown in figure 4.8.

IDEA 5 Version 5.0 allows circular dependencies, but it warns the user that they aren't recommended. Version 5.0 is capable of analyzing such dependencies and can assist in correct organization of the build process for such projects.

4.3.2 Creating a Java module with the module wizard

The primary module used by IDEA for everything other than J2EE applications is the *Java module*. The Java module covers everything from a simple command-line utility, to Swing applications, to libraries with a public API but no direct user interface. Being a generalist, it doesn't provide much in the way of special application support the way the J2EE modules do. We expect that in the future, specialized extensions of this module type will be available to ease the creation of common application themes. Even so, the features of the Java module form the core of the other module types and are important to learn.

Specifying the type of module to create

If you choose to create a new module from scratch, select the module type from the list in the **Add Module Wizard** (figure 4.9), and click **Next** to continue. This chapter focuses on the Java module; the other module types are covered in chapter 11.

IDEA 5 The **Add Module Wizard** looks different in version 5.0; it supports two new types of modules.

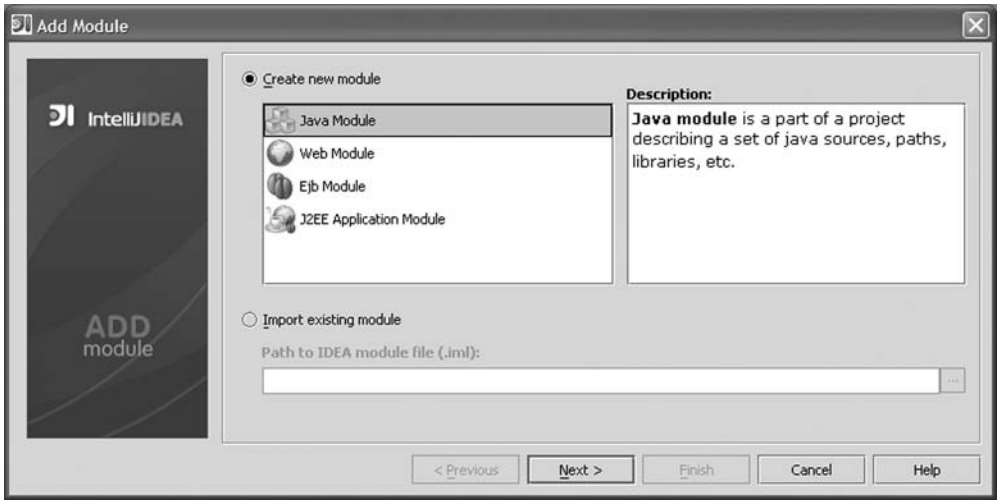

Figure 4.9 Step 1 of the Add Module Wizard prompts you to either create a new module from scratch or select an existing module on disk (represented by an IDEA module file, extension .iml).

Specifying the module file's name and location

The next step in the **Add Module Wizard** is much the same as the similar step in the **New Project Wizard**. As shown in figure 4.10, it lets you specify the name and

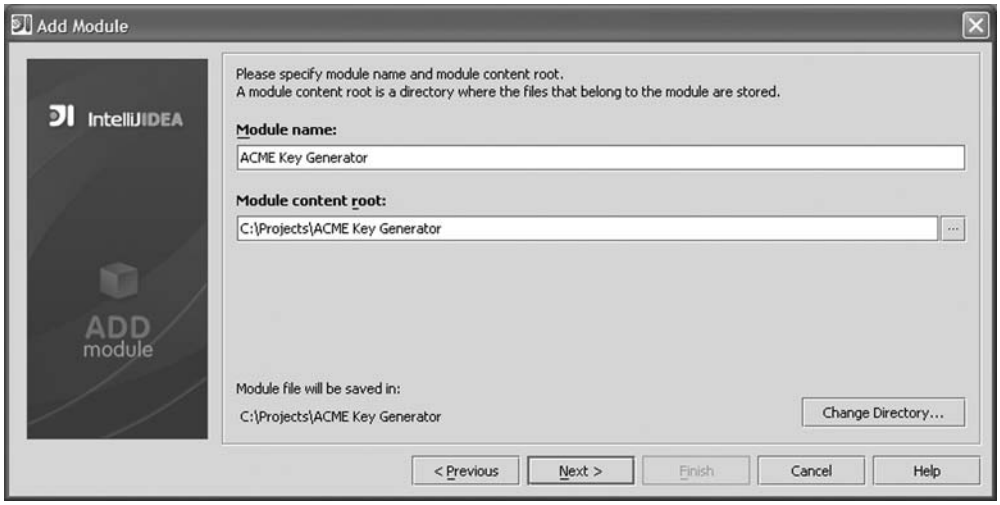

Figure 4.10 Step 2 of the Add Module Wizard involves naming and specifying a location for your new module.

location of the module file you're creating. In addition, however, it wants the location of the module's content root folder.

The module's content root folder is the folder that contains all the files that make up the module, including Java source files, Ant files, scripts, resources, and other related files. In the next step, you'll have the chance to specify a subfolder of your module's content root folder for source code.

By default, IDEA creates your new module's content root folder as a subdirectory of your project's root folder, based on the module's name. You can change the location manually or by clicking the ellipsis button to select an alternate location. Likewise, IDEA assumes that you want your module's `IML` file to be located in the content root folder, but you can change this by clicking the **Change Directory** button.

TIP Although it isn't required, the default project/module layout is recommended. Keeping your modules contained within their projects not only helps you stay better organized but also makes sharing them with others easier, because it eliminates absolute path problems. This approach also makes it easier to check your entire project into your source control system. One exception is situations where you're sharing modules between several projects, in which case it may make sense to locate modules above the project level.

Specifying the Java source folder

IDEA needs to identify the root of the folder (or folders) that contain your Java source files. In the **Add Module Wizard** dialog shown in figure 4.11, you're required to specify the subfolder that contains them. This folder path is relative to your module's content root folder, and IDEA will create it if it doesn't already exist.

This folder forms the root of your Java package hierarchy. You can direct IDEA to use an existing source directory by clicking the ellipsis button and navigating the directory browser to the right location. If you want to specify additional source folders, you can do so through the project's **Paths** settings, discussed a little later.

You don't have to store your sources in a subfolder: You can enter a period (.) as your source path to mark the entire content root folder as a source folder. Of course, this means any auxiliary files are mixed in with your Java source files, which can make things confusing.

If you don't want to specify a source directory, or you aren't ready to create it now (you can always add it later), select the **Do not create source directory** option.

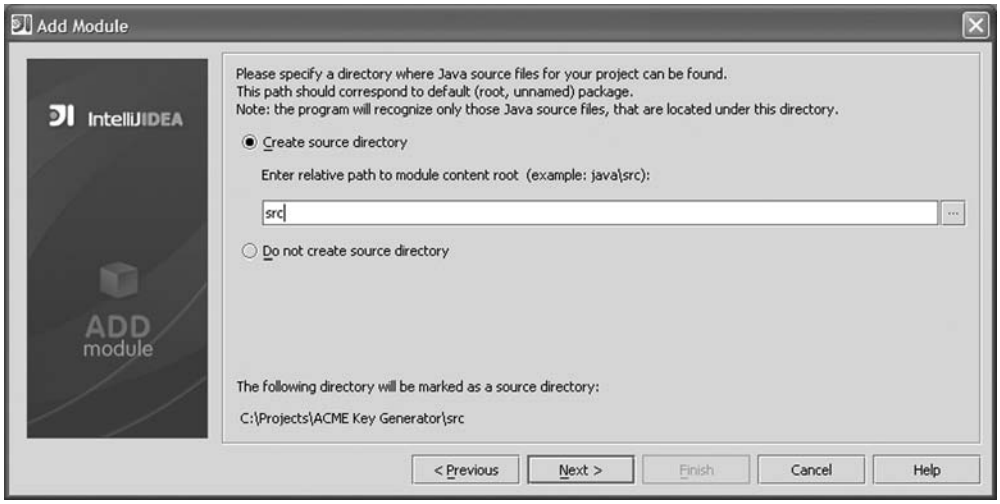

Figure 4.11 Step 3 of the Add Module Wizard prompts you to specify the main source folder where Java source files can be found.

TIP If there are already Java source files in the directory structure of the module's content root folder, IDEA automatically searches the directory tree for Java source code. All of the root directories of the Java package structures it finds are automatically provided as options for source folders. This is handy when you download a full project from CVS or another source and want IDEA to use the structure already provided.

Specifying the compiler output path

In the next step of the **Add Module Wizard**, you need to specify the folder you want to compile to. Regardless of how many source folders you define in your module, they will all build their class files into the folder you specify here. If your project includes multiple modules, you may want to compile all of them into the same output folder; but by default IDEA picks a subfolder of your content root folder, as shown in figure 4.12. (More on how all this works in a bit.)

When you click the **Finish** button, your new module is created.

4.3.3 Managing Java module settings

The **Add Module Wizard** does a fine job of creating a basic module, but you'll probably need to fine-tune your module settings. The definition of a Java module includes a content root folder, source and output paths, libraries, a dependency mapping between it and other modules, and documentation.

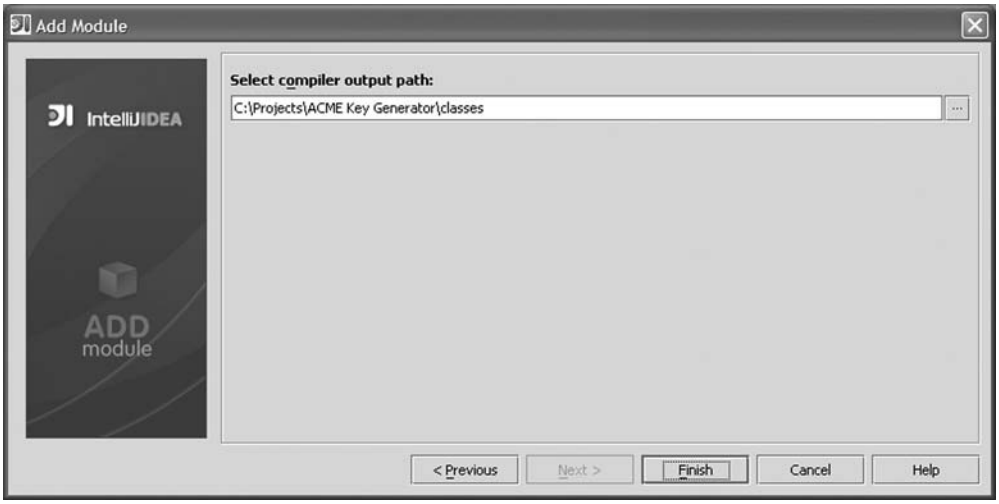

Figure 4.12 Step 4 of the Add Module Wizard prompts you to specify the directory where the compiler output (the class files generated from your Java sources) will be placed.

You can also access these settings by selecting **File | Settings**, by using the shortcut **Ctrl+Alt+S**, or by clicking the **IDE Settings** icon in the main toolbar (the wrench and machine nut), as shown earlier in figure 4.4. All module-specific configurations are found under the **Paths** icon on the **Paths** configuration screen shown in figure 4.13. This screen lets you configure a project by adding and removing modules from that project, and it also allows you to configure the included modules. Each module has configuration options for its **Paths**, **Libraries (Classpath)**, **Dependencies**, **Order/Export** (of the classpath) and **Javadoc**.

Configuring the content root folder

Modules have at least one content root folder, selected at creation time through the **Add Module Wizard** as you've just seen. All the files and paths that make up your module must fall under a content root folder. You can create additional content root folders if your module's files and folders are spread out in several directories. For most projects, this won't be necessary; one content root folder with subfolders for source code, class files, and so forth, will suffice.

The left panel in the **Paths** tab in figure 4.13 shows all the path folders defined for the currently selected module. These paths are organized by the content root folder they belong to, because some modules have more than one content root folder. You can add more content root folders by clicking the **Add Content Root**

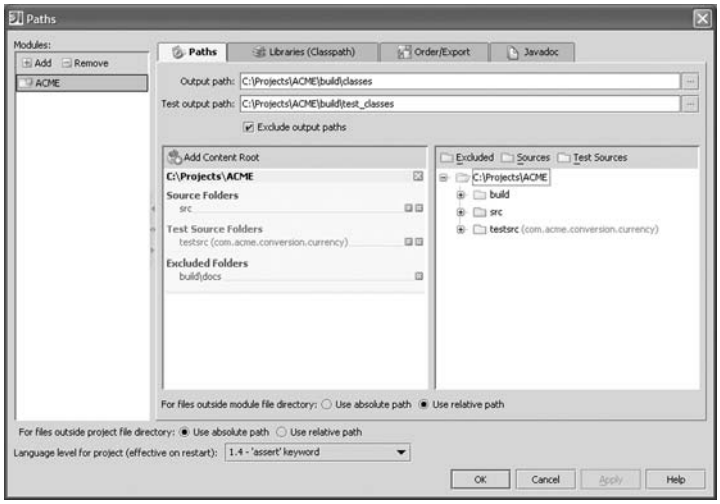

Figure 4.13 The Paths window is used to specify which directories in your modules contain Java source for production, Java source for testing purposes, and artifacts that should be excluded.

button. Content root folders can be removed easily by clicking the cross icon to the right of the content root item in the panel and confirming its removal.

Each content root folder contains folder paths under three important designations:

- *Source folders* contain Java source files.
- *Test source folders* contain Java source files related to testing only.
- *Excluded folders* are folders that IDEA should ignore when searching, compiling, and performing other similar actions.

Understanding source folders

When you designate a folder as a source path, you're telling IDEA that this folder and its subfolders contain Java source code that should be compiled as part of the build process. The top of this folder is considered the root of a package hierarchy; `.java` files in it are considered part of the default package, whereas subfolders designate a standard Java package structure.

Test source folders are singled out for special treatment so you can keep code used for testing (such as unit tests) separate from the production code. This makes it easy to package your application for distribution free from any testing-related code. We'll look further into IDEA's support for unit testing in chapter 7.

Understanding excluded folders

Excluded paths let you hide certain folders from IDEA. Files in folders designated as excluded aren't parsed, watched, searched, or compiled by IDEA. Typical candidates are temporary build folders, generated output, logs, and other project output.

TIP You can use the **Compiler** settings control panel to hide individual files or folders from the compiler without excluding them from the rest of IDEA. Doing so makes it possible to have source folders that are parsed and accessible for editing without having to compile them.

Configuring folders within the content root folder

The right panel of the dialog in figure 4.13 is used to locate and manage folder allocations within the content root folders. Selecting a content root item in the list shows the corresponding folder structure on the right side. To add a folder to the content root, click to select the folder in the panel, and then click the folder type button: **Excluded**, **Sources**, or **Test Sources**. The folder is added to the current content root folder selected, and its color is updated to reflect its status. You can remove folder items from the content root folder by clicking the cross button to the right of the folder items.

TIP You can do most of the actions of adding and removing folders from the content root item in the right panel. Right-clicking a folder displays a list of options that define what type of folder the selection can become. Selecting the same option again returns it to normal folder status. As an extra bonus, you can add new folders to the project by right-clicking and selecting the **New Folder** option.

Configuring source folders

Adding and removing source folders may be all that most projects require. However, because source folders are known to hold Java source files, the smart people at JetBrains have taken things a step further. Typically, source files exist in a subdirectory that matches their package structure, and thus source and test source directories are assumed to map to the default package. Some people think that's overkill for a module that specifies a single package that's deeply nested. Rather than create an arbitrarily deep directory structure to satisfy this convention, IDEA lets you specify a package root prefix, which means the package tree in that directory starts in the package you define and not the default package. In figure 4.13, the testing source directory is using a package root prefix.

Setting the package root prefix for a source folder (either a testing source folder or otherwise) is easy. Click the **P** button, and you'll see a dialog where you can type the package prefix. IDEA takes this prefix into account when compiling, viewing source, and other such tasks.

Understanding and configuring output paths

For each of the two types of source folders, IDEA maintains a separate output path for use during compilation. When you build the module, test sources are compiled to the test output path, and standard source files are compiled to the regular output path. Both output paths are automatically added to your project's classpath. You configure the output paths by editing the text in the paths at the top of the **Paths** editing dialog. For convenience, you can also use a visual file browser by clicking the ellipsis buttons to the right of each path.

Excluding output paths

You can add both output paths to your list of excluded paths by selecting the **Exclude output paths** option. (This option is enabled by default.) The output paths are still considered part of your classpath for running and debugging your project; but because they're generated by IDEA, there's typically no need for them to be monitored, searched, and so on.

4.4 Working with libraries

Using IDEA's library features makes it easy to manage the often-burdensome task of building applications utilizing third-party toolkits. IDEA manages not only the classes and JAR libraries for you, but the source and reference documentation as well. IDEA supports three different library configurations, which determine the scope and reusability of the library within your environment:

- *Project libraries* are defined within a project for its exclusive use.
- *Module libraries* are defined within a module for its exclusive use.
- *Global libraries* can be used by any project.

4.4.1 Understanding library basics

The three types of libraries supported by IDEA are configured and behave a bit differently, but they perform the same mission and share many of the same traits. Figure 4.14 shows the interface for configuring global libraries; it's essentially the same for all types of libraries. In many ways, configuring a library is similar to the

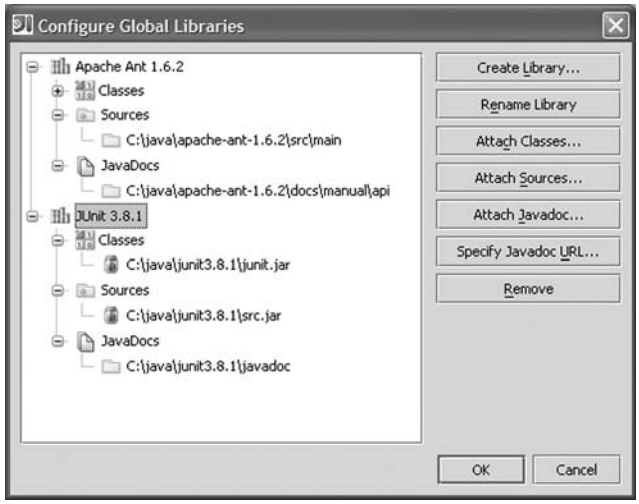

Figure 4.14
The Configure Global Libraries window demonstrates how a library is defined by a collection of class files, optional source files, and optional JavaDoc files. These files may be part of an exploded directory or packaged in an archive such as a JAR or ZIP file.

process of configuring your JDK from chapter 1; you specify a set of class files and then optional source and JavaDoc reference files.

Understanding class folders and JAR files

Every library must have at least one class folder or JAR archive added to it. When you add a library to a module or project, you're adding the library's classes to its classpath. Without class files, a library wouldn't do anything. Click the **Attach Classes** button, and select a directory of class files or a JAR archive of class files to add class path entries to the library. You may add multiple class entries if necessary.

Now, any module or project that uses this library will automatically recognize the classes and include them in the classpath when running or testing your application. Classes are all that is required to create a library, but if you attach source code or JavaDoc entries as described next, you'll gain further advantages.

Adding library source code

If the source code for the classes in the library is available, you should use the **Attach Source** button to add it to the library, even if you only have a portion of it. IDEA won't attempt to use the source code to rebuild the library; it only uses the source code to provide the same level of integration with libraries as it does with your own classes. When source is available, IDEA can provide inline documentation through the use of JavaDoc it extracts from the source code; it can also use the original parameter names in method signature help. In addition, IDEA lets you drill down into the source code (marked as read-only) while editing.

Adding JavaDoc reference

Like the source code reference, JavaDoc entries are optional. If they're present, you can use IDEA's **View | External JavaDoc** (**Shift+F1**) command to quickly load the JavaDoc for the method or class referenced at the cursor into your browser.

TIP Instead of folders, you can specify JAR or Zip archives that contain your class files, source folders, and JavaDocs. Just remember that the root of the archive should correspond to the root of the Java package structure. IDEA's file requestor lets you expand archives and select individual folders if you don't want to include the entire archive.

4.4.2 Adding libraries to the project

The **Libraries** tab of the project **Path** settings provides access to the selection and configuration of libraries (see figure 4.15). In addition, you can create global libraries outside the project through the **JDK & Global Libraries** control panel under the IDE Settings.

Adding global libraries

Global libraries exist outside the project and are available for use by any module, in any project on your machine. Global libraries were designed as a way to encapsulate third-party packages that, while not compiled as part of your application, are required to run it. Examples of this might be your web application server's JSDK implementation, a set of JDBC drivers, a third-party plugin API, and so on.

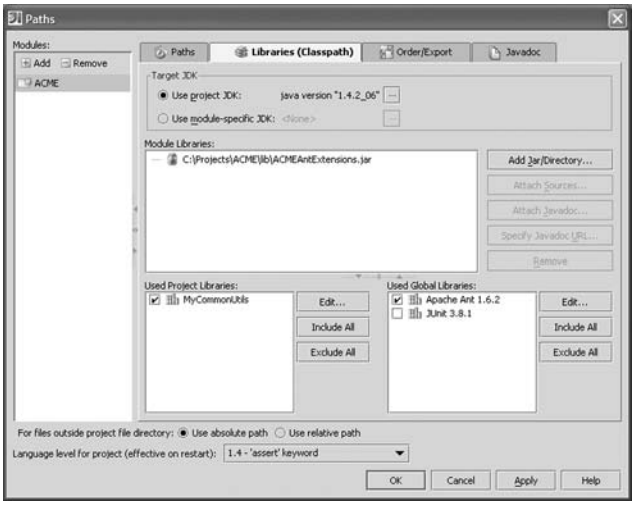

Figure 4.15
The Libraries tab of the Paths configuration window lets you specify module, project, and global libraries.

You add global libraries to a project by enabling them for each module that requires them. Once configured, you'll see each global library listed in the **Used Global Libraries** section of your module settings. To include a global library in the currently selected module, select the check box next to the name. The **Edit** button next to the list lets you edit exiting global libraries as well as include new ones.

Adding project libraries

Project libraries have the same look and feel as global libraries but are created at the project level. Therefore, they're only available to modules in the current project. If the module exists in more than one project, it doesn't share the project library in the other project. Available project libraries are shown in the **Project Libraries** list; like global libraries, they can be selected and deselected on a per-module basis.

Adding module libraries

Module libraries are defined within a particular module and are available to that module only. However, they can be carried with the module from project to project. Module libraries are anonymous; they aren't given individual names like global and project libraries, because the list of module libraries is the classpath for this module, with optional source file and JavaDoc attachments. Module library entries are often used to include items like resources, property files, and other non-class file entries into your project. They're also useful when the JAR files that make up your library are included as part of the module itself rather than in an external directory.

TIP Use IDEA's path variables feature (described later in this chapter) to make it easy to relocate, share, and upgrade libraries. Create a path variable pointing to the root of the library's installation folder; then each user can create a path variable corresponding to the libraries location on their machine.

Specifying search order (modules, classes, libraries)

In some situations, the order of items for the module's classpath may be important: for example, when one of your libraries is pulling in an older version of a class you require, or properties files in a library JAR conflict with one another. Whatever the situation is, you can use the **Order/Export** tab, shown in figure 4.16, to help correct the problem. Use the **Move Up** and **Move Down** buttons to reorder the module's classpath entries appropriately. Notice that in addition to your

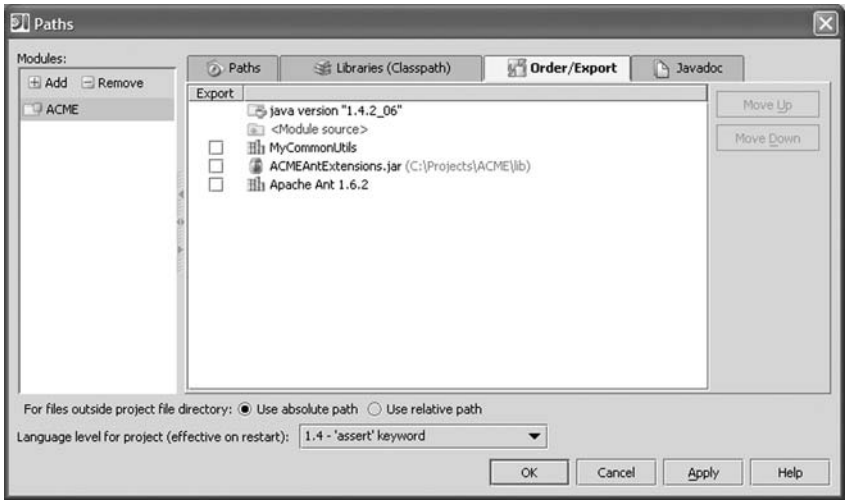

Figure 4.16 The Order/Export tab of the Paths configuration window can help you deal with issues such as classloading conflicts by explicitly setting which directories and class archives are loaded/searched first.

libraries and module class files, the module's sources and the JDK are also included in the list.

This window also includes an **Export** checkbox for each included library. By selecting this checkbox, you signify that the library should be exported to all other modules that have dependencies on this one.

4.4.3 Migrating projects from IDEA 3.x

Between version 3.x and version 4.x of IDEA, significant changes were made in the files that store the definition and configuration of projects: Version 3.x and previous lack the concept of a module. When you attempt to open an older-format IDEA project file, IDEA does its best to convert the project to the new format. Note, however, that there is no going back from this step; the older version of IDEA will no longer be able to read the project. As a convenience, IDEA creates a backup copy of the old project file by appending *_old* to the original filename.

When converting to the new format, IDEA creates a new project with a single Java module including all the source paths and libraries present in the original project. Any classpath entries are converted into module libraries, in accordance with the current way of thinking.

Because of conceptual differences between the two versions, some projects may suffer from conversion problems or missing elements. In particular, any

paths in 3.x projects that don't follow under the project root will be missing from the updated project file. Fortunately, this is an uncommon configuration and is easy to correct.

4.4.4 Sharing projects with others

All project files can be shared among a team of developers. Although you can certainly distribute copies of them, your best bet is to place these items under source code control along with the rest of your project's files to ensure that they're always up to date. You probably don't want to share workspace (`IWS`) files, because they contain information specific to each developer's IDE.

By default, all path-related settings in the project file are stored relative to the project file itself; the same is true for module files. This allows for easy relocation of projects and modules and lets each developer use their project from any location they choose. Any items outside the project or module directories (such as third-party libraries and your JDK settings) are stored by IDEA as absolute paths. This obviously can cause problems between machines. There are several possible solutions:

- Whenever possible, keep the content of your modules under the same path as the module file itself.
- In your project settings, set the option **For files outside project directory** to **Use relative paths**. This will avoid the use of absolute path names, making your project files easier to share between individuals. Modules have a similar option. You still need to make sure that each developer's source tree is arranged similarly in order for the paths to resolve correctly.
- Use path variables, as described next.

4.4.5 Using path variables

Path variables are project-independent macros defined in your IDE Settings. Any time IDEA encounters a path that includes a path variable reference (path references, like other variables in IDEA, are delineated by a pair of dollar sign characters), it uses your path variable settings to resolve the actual path.

Path variables allow you to hide the locations of files outside of your project or modules behind a named variable whose value can be changed on each installation of IDEA. This enables developers to share projects containing absolute path references.

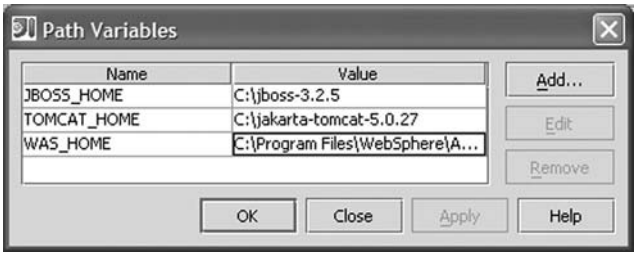

Figure 4.17
The Path Variables configuration window lets you define variables that make your project and module files more portable.

Using path variables to facilitate shared projects

Path variables are particularly useful when you're working with third-party libraries. For example, if your project uses the JDOM XML libraries, stored on your machine as `C:\libs\jdom-1.0.0`, all references to files below the root folder (its classes, sources, and JavaDocs) can instead be derived from the logical folder $JDOM_HOME$. Then, users of the project can define the actual location of the JDOM home directory on their own workstation, allowing IDEA to resolve the path correctly.

Creating and editing path variables

You manage your collection of path variables, and create new ones, through the **Path Variables** section of the **IDE Settings**. A typical example is shown in figure 4.17. In this example, we've configured copies of home folders for third-party libraries as well as our application server and Java Servlet SDK. Using this settings panel, you can add, edit, and remove path variables at any time. When you add a path variable's value, you're asked to select the directory value using the directory requestor—you can't edit the values by hand. This ensures that variables resolve to actual directories.

Using path variables in your project

IDEA automatically uses any path variables you've defined in your setup, replacing corresponding path references in your projects and modules with the path variable reference. Note that if you're sharing your project with other developers, you've now required them to define the path variables as well. Otherwise, IDEA won't know how to resolve the path correctly, in which case the path (with the path variable visible) will be

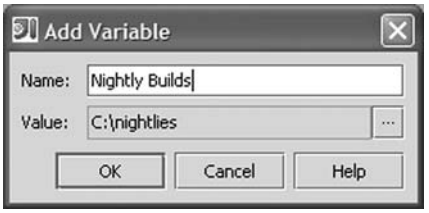

Figure 4.18 Adding a new path variable is as simple as specifying a name and choosing a path.

shown in red to alert the user to the missing path variable. Note that after you define the missing path variable, IDEA may require you to close and reopen the project before recognizing it.

4.5 Using the Project tool window

The first tool window we'll discuss in detail is probably the one you'll use most often. The **Project** window lets you navigate, edit, and explore the contents of your project's module and source files. All the files that are part of your project (and their libraries) are accessible from this view, organized by their parent modules. Figure 4.19 shows a typical example.

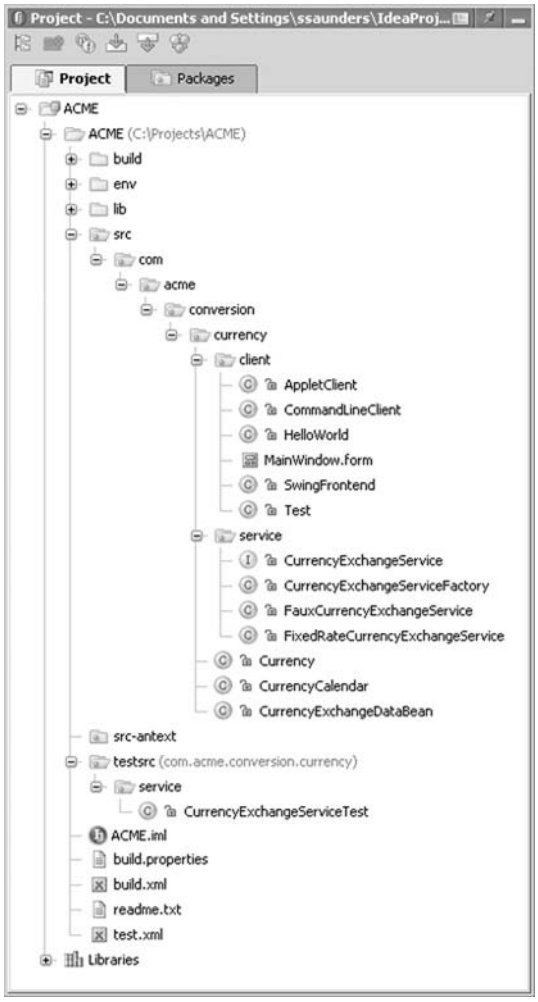

Figure 4.19 The Project tool window, navigating in Project mode. This view provides an overview of the directory structure that forms your project, including all modules and libraries.

4.5.1 Understanding the Project and Packages views

The two tabs at the top of the **Project** window let you switch between a module/file oriented view of your project and one representing all the available packages and classes that make up your source trees. (A third tab, which we'll talk about in chapter 11, gives you a J2EE-oriented view of web and enterprise applications. This tab only appears when your project contains these types of modules.) The **Packages** tab is unique in that it pays no attention to module boundaries or file-system structure. Instead, it provides a combined view of your entire source path, organized into the logical

Java package structure. An example **Packages** view is shown in figure 4.20.

4.5.2 Configuring the Project window

The options toolbar (see figure 4.21) along the top of the **Project** window lets you filter the **Project** views to your liking. These settings are saved along with your project and restored each time you open it. Note that some of the options aren't relative for all views.

Flattening packages

The **Flatten Packages** option removes the hierarchy from your package structure and collapses package families into their flattened form, as you might see them referenced with the code itself. As you can see in figure 4.22, this view is much more compact then the hierarchical view, allowing you to fit more packages within the view than normal. In the **Project** view, this only affects library entries; source directories remain in their file-system structure.

Showing and hiding middle packages

The **Hide Empty Middle Packages** option, available only when the **Flatten Packages** option is engaged, eliminates an empty package hierarchy. For example, if all your classes reside in the `com.acme.conversion.currency` package,

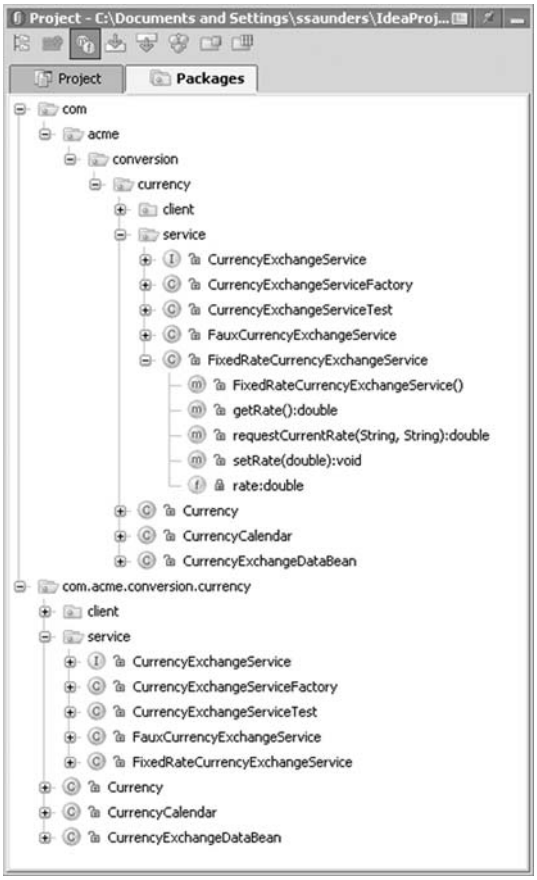

Figure 4.20 The Project tool window, navigating in package mode. In this view, the boundaries and divisions of disparate modules, directories, and libraries are hidden. It shows the project strictly as a set of packages and classes.

Figure 4.21 The Project tool window's options toolbar includes these buttons: Flatten Packages, Hide Empty Middle Packages, Show Members, Autoscroll to Source, Autoscroll from Source, Show Structure, Show Modules, and Show Libraries Contents. Most tool windows use a toolbar control like this for setting and unsetting options.

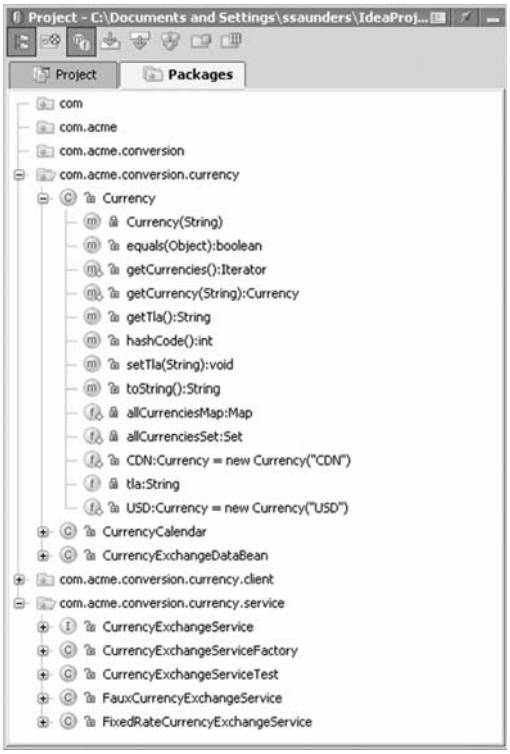

Figure 4.22
When the Flatten Packages option is enabled, the package view of the Project tool window shows top-level directories with package names instead of a deep directory structure. In this example, the Show Members option is also enabled, which adds nodes for methods and fields of each class to the overview.

enabling this option hides the three empty levels of packages above the `currency` directory. If your package structure is deep or complex, this option can give you back lots of real estate. It's also smart enough to not hide empty packages at the lowest or highest levels of hierarchy, because you probably intend to use these eventually (hence, the term *middle* packages.)

IDEA 5

When you're using the **Flatten Packages** option, a new option has been added to the **Project** view: **Abbreviate Qualified Package Names**. Enabling abbreviations shortens the package name considerably by replacing the leading package names with single letters. For example, the package `sun.net.www.protocol.http` becomes `s.n.w.p.http`. This saves you real estate, at the expense of readability.

The **Compact Empty Middle Packages** option lets you ignore empty packages for more convenient display. Its view is very similar to the **Flatten Packages** view, except that packages containing no classes are omitted from the tree. For example, if a project includes source code in the `com.texism` and in the `com.`

`texism.examples.book` packages, but no code at any other package level, this option shows only two packages on the tree: `com.texism` and `com.texism.examples.book`. The `com` and `com.texism.examples` packages are not shown. For all practical purposes, this option enables package flattening for empty package structures. This option is only available when you haven't enabled the **Flatten Packages** option.

Showing and hiding members

Want even more fine-grained access to the items in your project? Enabling the **Show Members** option exposes the methods and fields that belong to each class listed in the **Project** view, as shown in figure 4.22. Double-clicking a field or method item in the list loads its class's source file and positions the cursor at the point the member is declared. This is an excellent way to quickly explore the structure of your code and navigate right to the method of interest. However, it takes up a lot of room, so there is the alternative structure view, which we'll get to in a second.

Configuring autoscrolling to source

Enabling the **Autoscroll to Source** option turns all those double-click navigation operations we've mentioned into single clicks. Selecting any item in the **Project** window takes you to that item in the editor. It even works with keyboard navigation: Arrow around the tree, and watch the editor follow you.

The inverse option—**Autoscroll from Source**—provides the opposite functionality: While you navigate your code in the editor, the current semantic item (class, method, or field) is automatically selected in the **Project** window.

TIP With the **Project** window active, begin typing the name of the class, method, or field you want to navigate to. The selection automatically selects the next matching item. Then, use the up and down arrows to navigate between matching items.

Showing class structure

An alternative way to examine class members is handled by the **Show Structure** option. Enabling the **Structure** view adds a panel to the bottom of the **Project** view (as shown in figure 4.23) that exposes the structure of the selected class. This gives you the ability to navigate directly to a particular method without eating up the whole window. You can even resize the pane as necessary. For even more

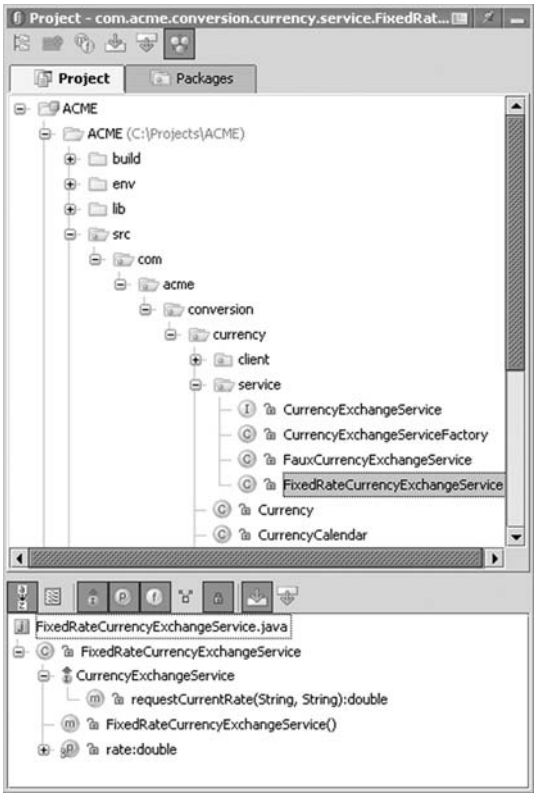

Figure 4.23
Enabling the Show Structure option in the Project tool window dedicates a portion of the tool window to a more detailed view into the class currently open in the main editor window.

structure fun, IDEA provides a dedicated tool window (the aptly named **Structure** window). We'll discuss the particulars in chapter 9 when we talk about analysis and refactoring.

Showing modules

The **Show Modules** option is available only in the **Packages** view. Enabling it adds the module structure pack to the **Packages** view, grouping packages under their parent module. Even if two modules share common packages, the class files appear only under the module whose source path they're defined in.

IDEA 5

The **Project** view now lets you group your project's modules into a tree structure of your choosing. This is purely for organizational purposes; but it can be convenient if your project includes many different modules, because you can selectively collapse and expand the groups you're working with at any given time.

To group a module, right-click the module in the project browser and select the **Move Module to Group** option, either selecting an existing group, creating a new group, or moving the module outside of all the existing groups. Creating a module grouping adds nodes to the project tree, allowing you to customize your view. Your code structure and the locations of your modules and source files are unaffected by the grouping operations.

Showing library contents

The **Show Libraries Contents** option, applicable only to the **Package** view, toggles the inclusion of library classes into the view. When enabled, classes that live in libraries appear in the package structure, although they're read-only. By default, IDEA assumes you're more interested in editing and navigating among your own classes, but this option can be handy in some circumstances.

IDEA 5

A new **Favorites** tool window has been introduced in the new release. It's basically a shelf where you can store an ad hoc selection of modules, packages, classes, files, or libraries for quick access. It's like a mini version of the **Project** browser. You can even create multiple groups of favorites (represented by tabs in the **Favorites** view) to further organize things.

To add an item to your list of favorites, right-click the item either in the **Project** view or through its symbol in the editor and select the **Add to Favorites** option from the context menu. Here you have the option of placing it into one of your existing groups of favorites or creating a new one.

4.6 Summary

Project configuration and management is an important aspect of software design, and any worthy IDE provides functionality to address this need. Without it, an engineer may as well be writing software in a plain text editor. The creators of IDEA have invested much thought and effort into making the project-management feature set within their IDE support the needs of their audience.

IDEA uses the concepts of projects, modules, and libraries to decompose the traditional concept of a software project. Projects are the highest level; they equate roughly to the products you're trying to produce. Modules are wholly contained subcomponents—individually buildable, runnable, and testable—that can be assembled into a larger solution. Libraries are static modules that aren't dynamically built or altered but that can be leveraged within the context of a module. By defining these layers and making them self-contained and modular (as Java components are touted to be), software designers can begin to reuse code in multiple projects and also manage that task with a minimum of effort.

JAVA

IntelliJ IDEA IN ACTION

COVERS IDEA v5

Duane K. Fields ▪ Stephen Saunders ▪ Eugene Belyaev

If you work with IDEA, you know its unique power and have already seen a jump in your productivity. But because IDEA is a rich system you, like many others, are probably using just a small subset of its features. You can overcome this syndrome and see your productivity take another leap forward—all you need is this book.

For new users, this book is a logically organized and clearly expressed introduction to a big subject. For veterans, it is also an invaluable guide to the expert techniques they need to know to draw a lot more power out of this incredible tool. You get a broad overview and deep understanding of the features in IDEA.

The book takes you through a sample project—from using the editor for entering and editing code, to building, running and debugging, and testing your application. The journey then continues into the far corners of the system. Along the way, the authors carefully explain IDEA's features and show you fun tricks and productivity-enhancing techniques that are the result of their combined years of experience.

What's Inside

- INTELLIGENT EDITING OF Java • JSP • XML • HTML • custom file types
- HOW TO add plugins • customize and extend IDEA • fine tune your code • inspect and analyze • refactor • develop Swing and J2EE applications

Duane K. Fields is a software developer and manager. He co-authored Manning's best-selling *Web Development with JavaServer Pages*. Stephen Saunders is a software engineer with experience in knowledge management, financial services, and data management. Eugene Belyaev is the cofounder, president, and chief technology officer of JetBrains, the company that created IDEA.

"An absolute winner! A must for every IDEA developer's shelf!"

—Mark Monster
Software Developer, BT

"… straightforward and very clear."

—Mark Woon
Software Developer
Stanford University

"… an entertaining read with excellent examples. Even a seasoned IDEA user will find it helpful."

—Sean Garagan
Technical Director
Versata, Inc.

www.manning.com/fields3

ISBN 1-932394-44-3

MANNING $44.95 US/$60.95 Canada