CHAPTER 2

Structure of the NetBeans Platform

To give you an overview of how a rich client application is structured and to show the relationship of the application you're creating to the NetBeans Platform, this chapter will discuss the architecture of the NetBeans Platform. It will also introduce the independent building blocks of the NetBeans Platform and the responsibilities that the NetBeans runtime container handles for you. Finally, this chapter will explain the structure of the NetBeans classloader system along with the role it plays in applications built on top of the NetBeans Platform.

NetBeans Platform Architecture

The size and complexity of modern applications is steadily increasing. At the same time, professional applications need to be flexible, above all, so that they can be quickly and easily extended. This makes it necessary to divide an application into distinct parts. As a result, each distinct part is a building block that makes up a modular architecture. The distinct parts must be independent, making available well-defined interfaces that are used by other parts of the same application, with features that other parts of the application can use and extend.

The division of an application into modules—that is, logically interdependent parts—enhances the design of an application enormously. In contrast to a monolithic application, in which every class can make use of code from any other class, the architecture is far more flexible and, more importantly, far simpler to maintain. Although it is possible in Java to protect a class from access from the outside world, but such class-level protection is too finely grained to be useful to most applications. It is exactly this central aspect of modern client applications that the NetBeans Platform tackles. Its concepts and structures support the development and conceptualization of flexible and modular applications.

The basic building block of the NetBeans Platform is modules. A module is a collection of functionally-related classes together with a description of the interfaces that the module exposes, as well as a description of the other modules that it needs in order to function. The complete NetBeans Platform, as well as the application built on top of it, are divided into modules. These are loaded by the core of the NetBeans Platform, which is known as the *NetBeans runtime container*. The NetBeans runtime container loads the application's modules dynamically and automatically, after which it is responsible for running the application as well.

The NetBeans IDE is a very good example of a modular rich client application. The functionality and characteristics of an IDE, such as its Java language support or the code editor, are created in the form of modules on top of the NetBeans Platform, as shown in Figure 2-1. This offers a great advantage because the application can be extended by additional modules and adapted to specific user needs, allowing particular modules that are not used to be deactivated or uninstalled.

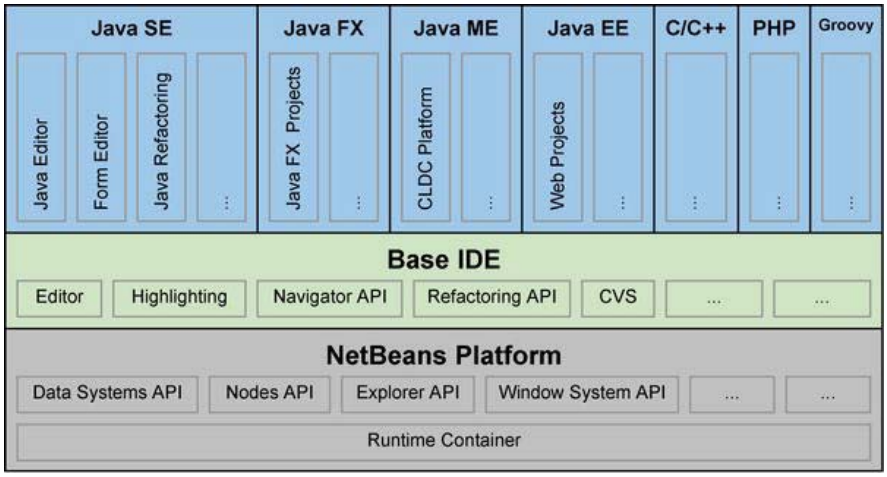

Figure 2-1. *Conceptual structure of the NetBeans IDE*

To enable your applications to attain this level of modularity, the NetBeans Platform on the one hand makes mechanisms and concepts available that enable modules to be extendable by other modules, and on the other hand enables them to communicate with each other without being dependent on each other. In other words, the NetBeans Platform supports a loose coupling of modules within an application.

To optimize the encapsulation of code within modules, which is necessary within a modular system, the NetBeans Platform provides its own *classloader system*. Each module is loaded by its classloader and, in the process, makes a separate independent unit of code available. As a result, a module can explicitly make its packages available, with specific functionality being exposed to other modules. To use functionality from other modules, a module can declare dependencies on other modules. These dependencies are declared in the module's manifest file and resolved by the NetBeans runtime container, ensuring that the application always starts up in a consistent state. More than anything else, this loose coupling plays a role in the declarative concept of the NetBeans Platform. By that I mean that as much as possible is defined in description and configuration files, in order to avoid a hard-wired connection of these concepts with the Java source code.

A module is described by its manifest file's data together with the data specified in related XML files and therefore does not need to be explicitly added to the NetBeans Platform. Using XML files, the NetBeans Platform knows the modules that are available to it, as well as their locations and the contracts that need to be satisfied for them to be allowed to be loaded. These dependencies are declared in the module's manifest file and resolved by the NetBeans runtime container, ensuring that the application always starts up in a consistent state. The NetBeans Platform itself is formed from a group of core modules (see Figure 2-2), which are needed for starting the application and for defining its user interface. To this end, the NetBeans Platform makes many API modules and service provider interface (SPI) modules available, simplifying the development process considerably. Included in this group (shown in Figure 2-2) are, for example, the Actions API, which makes available the often needed action classes; the powerful Nodes API; and the Options SPI, which helps your own options dialogs to be easily integrated into the application. In addition to these, there are also complete reusable components in the NetBeans Platform, such as the Output Window and the Favorites module.

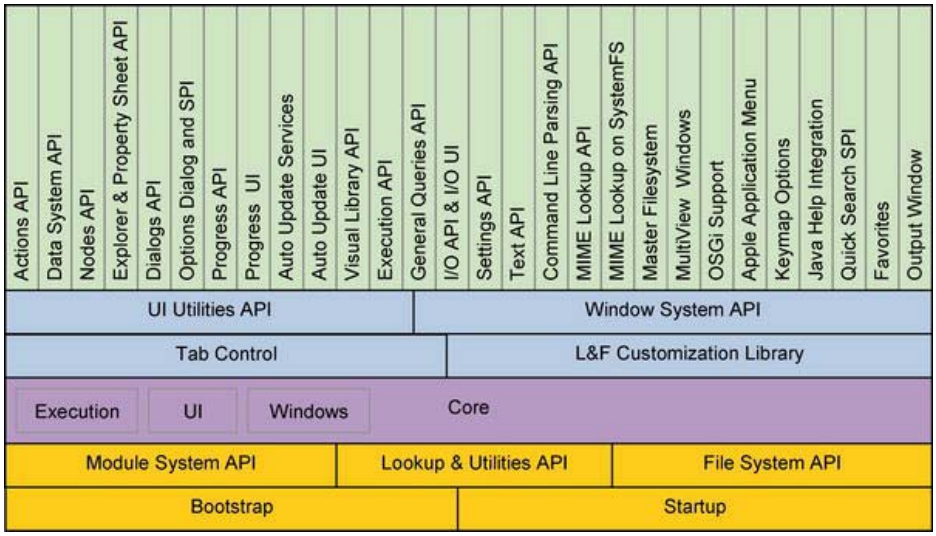

Figure 2-2. *NetBeans Platform architecture*

NetBeans Platform Distribution

Normally you do not need to separately download a distribution of the NetBeans Platform, because the NetBeans Platform is already a basic part of the NetBeans IDE, which is a rich client application itself. Developing your application in the NetBeans IDE, the Platform is extracted from the NetBeans IDE. However, there is the possibility of also adding multiple NetBeans Platforms to the NetBeans IDE. To that end, you can download a separate distribution of the NetBeans Platform from the official site, at `http://netbeans.org/features/platform`.

Now let's take a closer look at the important parts of the NetBeans Platform distribution:

- The modules `org.netbeans.bootstrap`, `org.netbeans.core.startup`, `org-openide-filesystems`, `org-openide-modules`, `org-openide-util`, and `org-openide-util-lookup` comprise the NetBeans runtime container, which is the core of the platform and is responsible for the development of all other modules.
- The NetBeans Platform also supports OSGi technology. The modules needed for this are `org.netbeans.libs.felix`, `org.netbeans.core.osgi`, `org.netbeans.core.netigso`, and `org.netbeans.libs.osgi`.
- The modules `org-netbeans-core`, `org-netbeans-core-execution`, `org-netbeans-core-ui`, and `org-netbeans-core-windows` provide basic functionalities for the API modules.
- `org-netbeans-core-output2` is a complete application module which can be used as a central output window.
- The module `org-netbeans-core-multiview` is a framework for MultiView Windows, (e.g., the form editor window) and provides an API for it.

- The module `org-openide-windows` contains the Window System API, which is the API that is probably used most. It includes basic classes for developing windows and the windows manager, among others. From the windows manager you can access information about all existing windows.
- The update functionality of an application is implemented by the module `org-netbeans-modules-autoupdate-services`. This module provides the complete functionality for finding, downloading, and installing modules. The module `org-netbeans-modules-autoupdate-ui` provides the Plugin Manager, which enables the user to manage and control modules and updates.
- With the `org-netbeans-modules-favorites` module you can display random data and directory structures and thereby influence their actions via the Data Systems API.
- The `org-openide-actions` module provides a set of frequently used actions, such as copy, cut, and print. The functionality of these actions can be implemented in a context-sensitive manner.
- A very powerful module is `org-openide-loaders`, which contains the Data Systems API. This can be used to create dataloaders that can be linked with certain types of files and then create data objects for it. A special behavior can be added to these data objects in a simple way.
- The Nodes API of the module `org-openide-nodes` is a very central feature of the NetBeans Platform. Nodes can, for example, be displayed in an explorer view; by doing so, nodes can provide actions and property sheets for data objects.
- The `org-openide-explorer` module provides a framework to develop explorer views as used, for example, in the projects or file view of the NetBeans IDE.
- The `org-netbeans-modules-editor-mimelookup` module provides an API to find MIME type–specific settings, services, and other objects, such as an SPI to implement your own MIME type–specific data provider. The `org-netbeans-modules-editor-mimelookup-impl` module is a special implementation of this SPI which is responsible for finding objects in the directory structure of the System Filesystem.
- `org-netbeans-modules-javahelp` contains the JavaHelp runtime library and provides an implementation to the modules API, which makes it possible for application modules to integrate their own helpsets by means of the JavaHelp technology.
- The QuickSearch SPI for implementing and providing your own providers is located in the module `org.netbeans.spi.quicksearch`.
- The master filesystem module `org-netbeans-modules-masterfs` provides an important wrapper filesystem.
- The module `org-netbeans-modules-options-api` provides an option dialog and an SPI, making it easy to add your own option panels.

- Long-running tasks can be managed centrally by the module `org-netbeans-api-progress`. The module `org-netbeans-modules-progress-ui` provides a visualization of this with which it is possible to stop separate tasks.
- `org-netbeans-modules-queries` provides a general query API with which modules can query information about files. An SPI is also provided to supply your own query implementations.
- `org-netbeans-modules-sendopts`, this module provides a *Command Line Parsing API* and an SPI with which your own handlers can become registered for command lines.
- The `org-netbeans-modules-settings` module provides an API to save module-specific settings in a user-defined format. It also provides several useful setting formats.
- The `org-openide-awt` module includes the *UI Utilities API*, by which the different help classes for creating the user interface are provided.
- In the module `org-openide-dialogs` an API for displaying standard and application-specific dialogs is provided. The Wizard Framework is located in this module.
- `org-openide-execution` provides an API for executing long-running asynchronous tasks.
- `org-openide-io` provides an API and an SPI for the input and output of files. This module also provides a standard implementation with which you can write on the Output Window module.
- The Text API in the module `org-openide-text` provides an extension of the `javax.swing.text` API.
- The modules `org-netbeans-swing-plaf` and `org-netbeans-swing-tabcontrol` are responsible for the adaptation of the look and feel and displaying the tabs. The module `org-jdesktop-layout` is a wrapper module of the Swing layout extenstion library.
- The Visual Library API is provided by the module `org-netbeans-api-visual`.

Furthermore, it is possible to add modules out of the IDE distribution to the listed modules.

NetBeans Runtime Container

The basis of the NetBeans Platform and its modular architecture is called *NetBeans Runtime Container*. It consists of the following five modules:

- *Bootstrap*: This module is executed initially. It executes all registered command-line handlers, creates a boot classloader which loads the startup module, and then executes it.
- *Startup*: This module deploys the application by initializing the module system and the file system.

- *Module System API*: This API is responsible for the management of the modules and for their settings and dependencies.
- *File System API*: This API provides a virtual file system which provides a platform-independent access . It is mostly used for loading resources of the modules.
- *Lookup & Utilities API* : This component provides an important base component which is used for the intercommunication of the modules. The Lookup API is located in an independent module, so it can be used independently of the NetBeans Platform.

The arrows in Figure 2-3 show the dependencies of these five basic modules.

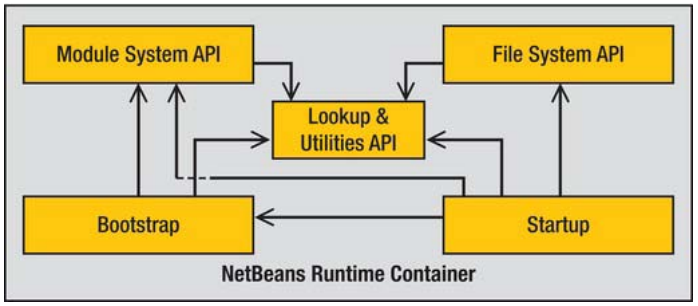

Figure 2-3. *NetBeans runtime container*

The runtime container is the minimal form of a rich client application and can be executed as such without further modules. If there are no tasks to do, the runtime container would directly shut down again after starting up. It is interesting to note that, on the one hand, the Netbeans Platform can create applications with an extensive user interface, and on the other, can also use this runtime container for a modular command-line application. Starting the runtime container, it finds all available modules and creates an internal registry out of them. Usually, one module is just loaded when needed. First, it is registered as existing. However, a module has the possibility to do tasks right at the start. This is done by the Module Installer, which will be discussed in Chapter 3. The runtime container also facilitates dynamic loading, unloading, installing, and uninstalling of modules during runtime. This functionality is especially necessary for users when updating an application (with the auto update function). It is also necessary for deactivating unneeded modules within an application.

For a complete understanding of the process of a rich client application, it is also important to mention that the Bootstrap module (the first module executed) is started by a platform-specific launcher. This launcher is also responsible for identifying the Java Runtime Environment. The launcher is part of the NetBeans Platform and is operating system (or OS) specific, so that, for example, on Windows systems it is an `.exe` file.

NetBeans Classloader System

The NetBeans classloader system is part of the NetBeans runtime container and a precondition for encapsulating the module and the structure of a modular architecture. This system consists of three different types of classloaders. These are the *module classloader*, the *system classloader*, and the *original classloader*.

- Most classes are loaded by the module classloader.

- The system classloader is only used in certain cases, such as when resources must be accessed outside a module.
- The original classloader loads resources out of the class path of the launcher of the application.

The module classloader and the system classloader are multiparent classloaders; they can have not just one classloader as parent, as usual, but any number of parents. Figure 2-4 shows the connections of the single classloader types.

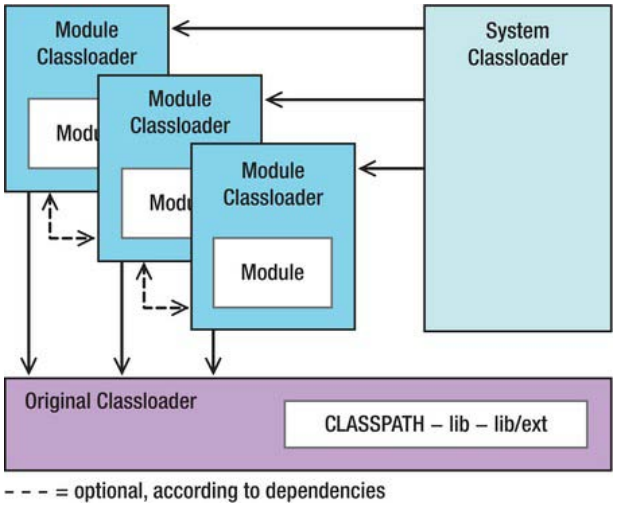

Figure 2-4. *NetBeans classloader system*

Module Classloader

For each module registered in the Module System, an instance of the module classloader is created, by means of which every module obtains its own namespace. This classloader primarily loads classes from the module's JAR archive, but it may load from multiple archives, as often happens with library wrapper modules. You will learn more about this in Chapter 3.

The original classloader is implicitly a parent classloader of every module classloader, and is the first on the parent's list. Further parents are those of related modules, on which dependencies have been set. How dependencies are set is described in Chapter 3.

This multiparent module classloader enables classes to be loaded from other modules, while avoiding namespace conflicts. The loading of classes is delegated to the parent classloader, rather than the modules themselves. In addition to the classes of the module JAR archive, this classloader is also responsible for loading the Locale Extension Archive (see Chapter 34) from the subdirectory `locale`, as well as the patch archives under the subdirectory `patches`, if these are available.

System Classloader

The system classloader is, by default, a multiparent classloader. It owns all the instantiated module classloaders as its parents. As a result, it is theoretically possible to load everything provided by a module with this classloader. Access to the system classloader can be obtained in one of two different ways: via

Lookup (about which you will read much more later), as well as the context classloader of the current thread. This is the default (insofar as you have not explicitly set other context classloaders) of the system classloader.

```
ClassLoader cl = (ClassLoader) Lookup.getDefault().lookup(ClassLoader.class);
```

or

```
ClassLoader cl = Thread.currentThread().getContextClassLoader();
```

Original Classloader

The original (application) classloader is created by the launcher of the application. It loads classes and other resources on the original `CLASSPATH` and from the `lib` directories and their `ext` subdirectories as well. If a JAR archive is not recognized as a module (that is, the manifest entries are invalid), it is not transferred to the module system. Such resources are always found first: if the same resource is found here as in the module JAR archive, those found in the module are ignored. This arrangement is necessary for the branding of modules, for example, as well as for the preparation of multiple language distributions of a particular module. As before, this classloader is not used for loading all related resources. It is much more likely to be used for resources that are needed in the early start phase of an application, such as for the classes required for setting the look and feel classes.

Summary

This chapter examined the structure of the NetBeans Platform beginning with a look at its architecture, the core of which is provided by its runtime container. The runtime container provides the execution environment of applications created on top of the NetBeans Platform and also provides an infrastructure for modular applications. The NetBeans classloader system, which ensures the encapsulation of modules, was introduced and explained. Aside from the runtime container, many modules form parts of the NetBeans Platform and this chapter looked briefly at each of these, finally noting that the NetBeans IDE is itself a rich client application consisting of modules reusable in your own applications.

CHAPTER 3

The NetBeans Module System

The *NetBeans Module System* is responsible for managing all modules. This means it is responsible for tasks such as creating the classloader, loading modules, or activating or deactivating them. The NetBeans module system was designed using standard Java technologies, as much as possible. The basic idea for the module format originates from the Java extension mechanism. The fundamental ideas of the package versioning specification are used to describe and manage dependencies between application modules and applications of system modules.

Basic properties, such as the description of a module and the dependencies on another module, are described in a manifest file. This file uses the standard manifest format with additional NetBeans-specific attributes. The Java Activation Framework and Java Development Kit (JDK) internal functions (such as the support of executable JAR archives) were models for the module specification. Most modules do not need a special installation code, except of the attributes in the manifest file, meaning they are declaratively added to the Platform. An XML file, the *layer.xml* file, provides user-specific information and defines the integration of a module into the Platform. In this file everything is specified that a module wants to add to the Platform, ranging from actions to menu items to services, among others.

Structure of a Module

A module is a simple JAR archive which usually consists of the following parts:

- Manifest file (*manifest.mf*)
- Layer file (*layer.xml*)
- Class files
- Resources like icons, properties bundles, helpsets, etc.

Only the manifest file is obligatory, because it identifies a module. All other content depends on its modules task. For example, if the module just represents a library, no layer file is needed. The structure of a module is shown in Figure 3-1.

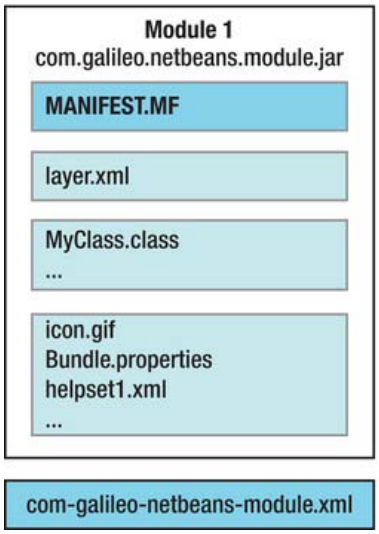

Figure 3-1. *NetBeans module*

Furthermore, an XML configuration file (`com-galileo-netbeans-module.xml`), which is located outside the JAR archive, belongs to each module. This is the first file read by the module system; that is, it announces the module to the Platform.

Configuration File

Each module is declared in the module system by an XML configuration file, located outside the module in the directory *config/Modules* of a cluster. This directory is read by the module system when the application is started. The modules are loaded according to this information. In this configuration file the name, version, and the location of the module are defined and whether and how a module is loaded is defined. This file has the following structure, as shown in Listing 3-1.

Listing 3-1. *Module configuration file: com-galileo-netbeans-module.xml*

```
<module name="com.galileo.netbeans.module">
   <param name="autoload">false</param>
   <param name="eager">false</param>
   <param name="enabled">true</param>
   <param name="jar">modules/com-galileo-netbeans-module.jar</param>
   <param name="reloadable">false</param>
   <param name="specversion">1.0</param>
</module>
```

The `enabled` attribute defines whether a module is loaded, and therefore whether it is provided to the application. There are three ways to determine at which point a module should be loaded:

- *Regular*: Most application modules are this type. They are loaded when starting the application. The application loading time is extended by the time of module initialization. Therefore, it is recommended to keep the module initialization very short. Normally, it is not necessary to run anything during module loading, because many tasks can be defined declaratively.
- *Autoload*: These modules are just loaded, when another module requires them. Autoload modules correspond to the principle of *Lazy-Loading*. This mode is usually used for those modules acting as libraries.
- *Eager*: Eager modules are only loaded when all dependencies are met. This is another possibility for minimizing the starting time. For example, if a module X depends on the modules A and B which are actually not even available, it makes no sense to load module X.

If the value of both attributes `autoload` and `eager` is `false`, a module is type Regular. If one of these values is `true`, the module type is Autoload or Eager. The module type is defined in the API Versioning section of the modules Properties (see Figure 3-7). Regular mode is used, by default.

Manifest File

Each module running within the NetBeans Platform has a manifest file. This file is a textual description of the module and its environment. When loading a module, the manifest file is the first file read by the module system. A NetBeans module is recognized if the manifest file contains the `OpenIDE-Module` attribute. This is the only mandatory attribute. Its value can be any identifier (typically the code name is the base of the used module—for example, `com.galileo.netbeans.module`). Therefore, conflicts cannot occur between modules, even if created by various developers. This identifier is used to clearly distinguish a non-ambiguous module which is necessary for upgrades or dependency definitions, for example.

Attributes

In the following, frequently used manifest attributes are listed. A module can be textually described with those manifest attributes. Additionally, those attributes determine the integration of a module into the Platform.

- *`OpenIDE-Module`*: This attribute defines a unique name for the module used for recognition as a module by the module system. Defining this attribute is obligatory.

  `OpenIDE-Module``: com.galileo.netbeans.module`

- *`OpenIDE-Module-Name`*: This defines a displayable name of the module which is also displayed in the plugin manager.

  `OpenIDE-Module-Name:` `My First Module`

- *`OpenIDE-Module-Short-Description`*: A short functionality description by the module.

  `OpenIDE-Module-Short-Description``:`
  `This is a short description of my first module`

- *OpenIDE-Module-Long-Description*: With this attribute the functionality of the module can be better described. This text is also displayed in the plugin manager. Thus, it makes sense to always use this attribute, to inform the user about the module's features.

  ```
  OpenIDE-Module-Long-Description:
     Here you can put a longer description with more than one
     sentence. You can explain the capability of your module.
  ```

- *OpenIDE-Module-Display-Category*: With this attribute, modules can be grouped to a virtual group so it can be presented to the user as a functional unit.

  ```
  OpenIDE-Module-Display-Category: My Modules
  ```

- *OpenIDE-Module-Install:* A module installer class can be registered with this attribute (see the section Lifecycle), in order to execute actions at certain points of time of the module life cycle.

  ```
  OpenIDE-Module-Install: com/galileo/netbeans/module/Installer.class
  ```

- *OpenIDE-Module-Layer*: This is one of the most important attributes. With it the path of the layer file is specified (see the section *Layer File*). The integration of a module into the Platform is described by the layer file.

  ```
  OpenIDE-Module-Layer: com/galileo/netbeans/module/layer.xml
  ```

- *OpenIDE-Module-Public-Packages*: To support encapsulation, accessing classes in other modules is denied by default. With this attribute, packages can be explicitly declared as public so other modules can access it. This is especially essential with libraries.

  ```
  OpenIDE-Module-Public-Packages:
     com.galileo.netbeans.module.actions.*,
     com.galileo.netbeans.module.util.*
  ```

- *OpenIDE-Module-Friends*: If only certain modules can access the packages which are declared as public with the attribute `OpenIDE-Module-Public-Packages` then those may be stated here.

  ```
  OpenIDE-Module-Friends:
     com.galileo.netbeans.module2,
     com.galileo.netbeans.module3
  ```

- *OpenIDE-Module-Localizing-Bundle*: Here, a properties file can be defined that is used as a localizing bundle (see Chapter 8).

  ```
  OpenIDE-Module-Localizing-Bundle:
     com/galileo/netbeans/module/Bundle.properties
  ```

Versions and Dependencies

Different versions and dependencies can be defined with the following attributes. In the section *Versioning and Dependencies* you find a detailed description of the application and of the whole functionality of these attributes.

- *OpenIDE-Module-Module-Dependencies*: With this attribute the dependencies between modules are defined, and the least-needed module version can also be specified.

  ```
  OpenIDE-Module-Module-Dependencies:
     org.openide.util > 6.8.1,
     org.openide.windows > 6.5.1
  ```

- *OpenIDE-Module-Package-Dependencies*: A module may also depend on a specific package. Such dependencies are defined with this attribute.

  ```
  OpenIDE-Module-Package-Dependencies: com.galileo.netbeans.module2.gui > 1.2
  ```

- *OpenIDE-Module-Java-Dependencies*: If a module requires a specific Java version, it can be set with this attribute.

  ```
  OpenIDE-Module-Java-Dependencies: Java > 1.5
  ```

- *OpenIDE-Module-Specification-Version*: This attribute indicates the specification version of the module. It is usually written in the Dewey-Decimal format.

  ```
  OpenIDE-Module-Specification-Version: 1.2.1
  ```

- *OpenIDE-Module-Implementation-Version*: This attribute sets the implementation version of the module, usually by a timestamp. This number should change with every change of the module.

  ```
  OpenIDE-Module-Implementation-Version: 200701190920
  ```

- *OpenIDE-Module-Build-Version:* This attribute has only an optional character and is ignored by the module system. Typically, a timestamp is given.

  ```
  OpenIDE-Module-Build-Version: 20070305
  ```

- *OpenIDE-Module-Module-Dependency-Message*: Here, a text can be set. This text is displayed if a module dependency cannot be resolved. In some cases, it can be quite normal to have an unresolved dependency. In this case, it is a good idea to show the user a helpful message, informing them where the required modules can be found or why none are needed.

  ```
  OpenIDE-Module-Module-Dependency-Message:
     The module dependency is broken. Please go to the
     following URL and download the module.
  ```

- *OpenIDE-Module-Package-Dependency-Message*: The message defined by this attribute is displayed if a necessary reference to a package fails.

  ```
  OpenIDE-Module-Package-Dependency-Message:
     The package dependency is broken. The reason could be…
  ```

- *OpenIDE-Module-Deprecated*: Use this to mark an old module which is no longer supported. A warning is displayed if the user tries to load the module into the Platform.

  ```
  OpenIDE-Module-Deprecated: true
  ```

- *OpenIDE-Module-Deprecation-Message*: Use this attribute to add optional information. Both the information and the deprecated warning are displayed in the application log so, for example, you can tell the user which module to use instead. Note that this message will only be displayed if the attribute `OpenIDE-Module-Deprecated` is set to `true`.

  ```
  OpenIDE-Module-Deprecation-Message:
     Module 1 is deprecated, use Module 3 instead.
  ```

Service Interfaces and Service Implementations

The following attributes are used to define certain service provider interfaces and implementations. Further information on this topic can be found in Chapter 5.

- *OpenIDE-Module-Provides*: Use this attribute to declare a service interface to which this module furnishes a service provider.

  ```
  OpenIDE-Module-Provides: com.galileo.netbeans.spi.ServiceInterface
  ```

- *OpenIDE-Module-Requires*: Here, a service interface can be declared for which the module needs a service provider. It does not matter which module provides an implementation of the interfaces.

  ```
  OpenIDE-Module-Requires: org.openide.windows.IOProvider
  ```

- *OpenIDE-Module-Needs*: This attribute is an understated version of the require attribute and does not need any specific order of modules. This may be useful with API modules which require a specific implementation.

  ```
  OpenIDE-Module-Needs: org.openide.windows.IOProvider
  ```

- *OpenIDE-Module-Recommends*: Using this attribute, you can realize an optional dependency. For example, if there is a module which provides a `java.sql.Driver` implementation, this module is activated and access is granted. Nevertheless, if no provider of this token is available, the module defined by the optional dependency can be executed.

  ```
  OpenIDE-Module-Recommends: java.sql.Driver
  ```

- *OpenIDE-Module-Requires-Message*: Like the two previous attributes, a message can be defined with this attribute. This message is displayed if a required token is not found.

  ```
  OpenIDE-Module-Requires-Message:
     The required service provider is not available. For more
     information go to the following website.
  ```

OPERATING SYSTEM-DEPENDENT MODULES

The manifest attribute `OpenIDE-Module-Requires` allows you to define modules that tend to be used on a specific operating system. This attribute is used to check the presence of a particular token. The following tokens are available:

```
org.openide.modules.os.Windows
org.openide.modules.os.Linux
org.openide.modules.os.Unix
org.openide.modules.os.PlainUnix
org.openide.modules.os.MacOSX
org.openide.modules.os.OS2
org.openide.modules.os.Solaris
```

The module system ensures that the tokens are only available on the appropriate operating systems. For example, a module that is automatically activated by the module system using a Windows system would automatically be deactivated with all others. To provide a module that automatically loads on Windows systems but automatically deactivates on other operating systems, set the module type to `eager` and add the following entry to the manifest file:

```
OpenIDE-Module-Requires: org.openide.modules.os.Windows
```

Visibility

With the following attributes, the visibility of modules within the plugin manager is controlled. This way, modules can be hidden, which are not important to the end user of your application.

- *`AutoUpdate-Show-In-Client`*: This attribute determines whether a module is displayed in the plugin manager. It can be set to `true` or `false`.

  ```
  AutoUpdate-Show-In-Client: true
  ```

- *`AutoUpdate-Essential-Module`*: With this attribute you can mark modules that are part of your application. A module which is marked like this cannot be deactivated or uninstalled in the plugin manager by the user. It can be set to `true` or `false`.

  ```
  AutoUpdate-Show-In-Client: true
  ```

In conjunction with these two attributes, so-called *kit* modules were introduced in the NetBeans Platform. Each visible module (`AutoUpdate-Show-In-Client: true`) is treated as a kit module in the plugin manager. All modules on which the kit module defines a dependency are treated the same way, except for invisible modules which belong to other kit modules, too. This means if a kit module is deactivated, all dependent modules will be deactivated as well.

This way you can build wrapper modules to several logically related modules. Then you can display them to the end user as a unit. You can create an empty module in which the `AutoUpdate-Show-In-Client` attribute is set to `true`, while defining a dependency on all modules which belong to this kit module. Then you set the attribute `AutoUpdate-Show-In-Client` to `false` in the dependent modules so they are not displayed separately.

Example

Listing 3-2 shows a manifest file with some typical attributes.

Listing 3-2. *Example of a Manifest File*

```
OpenIDE-Module: com.galileo.netbeans.module
OpenIDE-Module-Public-Packages: -
OpenIDE-Module-Module-Dependencies:
   com.galileo.netbeans.module2 > 1.0,
   org.jdesktop.layout/1 > 1.4,
   org.netbeans.core/2 = 200610171010,
   org.openide.actions > 6.5.1,
   org.openide.awt > 6.9.0,
OpenIDE-Module-Java-Dependencies: Java > 1.6
OpenIDE-Module-Implementation-Version: 200701100122
OpenIDE-Module-Specification-Version: 1.3
OpenIDE-Module-Install: com/galileo/netbeans/module/Install.class
OpenIDE-Module-Layer: com/galileo/netbeans/module/layer.xml
OpenIDE-Module-Localizing-Bundle: com/galileo/netbeans/module/Bundle.properties
OpenIDE-Module-Requires:
   org.openide.windows.IOProvider,
   org.openide.modules.ModuleFormat1
```

Layer File

In addition to the manifest file of a module with which mainly the interfaces and the environment of a module are described, there is also a *Layer* file. This is the central configuration file, in which virtually everything is defined what a module adds to the Platform. So, the layer file is the interface between the module and the NetBeans Platform, declaratively describing the integration of a module into the Platform. Firstly, the attribute `OpenIDE-Module-Layer` makes public if a layer file exists in the manifest file. During that process the path of the file is defined, usually using layer.xml as the file name.

```
OpenIDE-Module-Layer: com/galileo/netbeans/module/layer.xml
```

This file format is a hierarchical file system containing folders, files, and attributes. Starting the application, all existing layer files are summarized to one virtual file system. This is the so-called *System Filesystem* which is the runtime configuration of the NetBeans Platform.

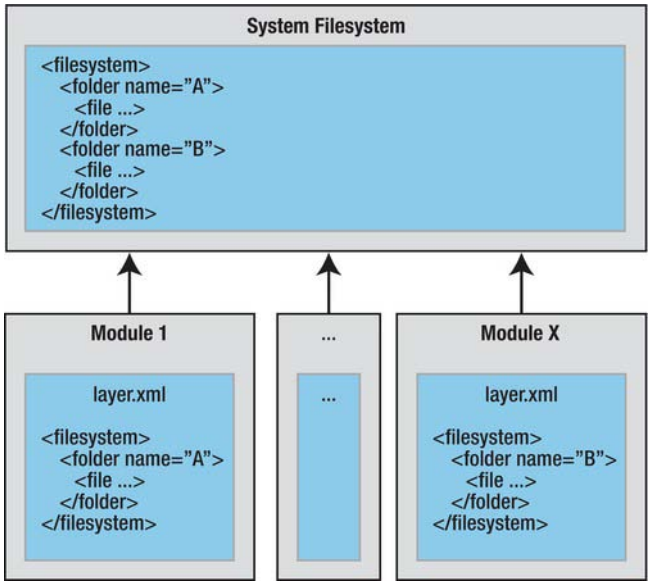

Figure 3-2. *System Filesystem*

This layer file contains certain default folders. They are defined by different modules which are extension points. So, there is the default folder `Menu`, for example, which looks like Listing 3-3.

Listing 3-3. *Default Folder of the Layer File*

```
<folder name="Menu">
   <folder name="Edit">
      <file name="MyAction.shadow">
         <attr name="originalFile"
               stringvalue="Actions/Edit/com-galileo-netbeans-module-MyAction.instance"/>
      </file>
   </folder>
</folder>
```

In this example, the action class `MyAction` is added to the Edit menu. Do not worry about the exact syntax at this point; it is explained in the context of respective standard folders in later chapters. First of all, we elaborate this basic structure of the layer file. In addition, the NetBeans Platform provides practical features for working with the layer file, as shown in subsequent chapters, when our first module will be created. You can also find an index with the most important extension points in this book's appendix.

Every module is able to add new menu items or create new toolbars. Since each layer file of a module is merged to the System Filesystem, the entire menu bar content is assembled. The window system, which is responsible for generating the menu bar, now has to read the `Menu` folder in order to gain the content of the entire menu bar.

This System Filesystem also contributes significantly to the fact that modules can be added or removed at runtime. Listeners can be registered on this System Filesystem. This is done by the window

system, too, for example. If any changes occur because of an added module, the window system or the menu bar itself can update its content.

Order of Folders and Files

The order in which the entries are read out of the layer file (and the order they are displayed in the menu bar), can be determined by a position attribute as shown in Listing 3-4.

Listing 3-4. *Defining the Order of Entries in the Layer File*

```
<filesystem>
  <folder name="Menu">
    <folder name="Edit">
      <file name="CopyAction.shadow">
        <attr name="originalFile"
              stringvalue="Actions/Edit/org-openide-actions-CopyAction.instance"/>
        <attr name="position" intvalue="10"/>
      </file>
      <file name="CutAction.shadow">
        <attr name="originalFile"
              stringvalue="Actions/Edit/org-openide-actions-CutAction.instance"/>
        <attr name="position" intvalue="20"/>
      </file>
    </folder>
  </folder>
</filesystem>
```

Thus, the copy action would be shown before the cut action. If necessary, you can also use this attribute to define the order of the `folder` elements. In practice, positions are chosen with greater distance. This simplifies the subsequent insertion of additional entries. Should the same position be assigned twice, a warning message is logged while running the application.

In order to easily position the layer content, the NetBeans IDE offers a layer tree in the projects window, in which all entries of the layer files are shown. There, their order can be defined by drag-and-drop. The respective entries in the layer file are then handled by the IDE. Where exactly to find these layer trees is already explained in the section *Creating Modules* after creating our first module. You can determine the order of actions with the wizard of the NetBeans IDE (see Chapter 6). The respective attributes are then created by the wizard.

Should positions of entries in the layer tree be changed, some entries will be added to the layer file. These entries overwrite the default positions of the entries affected by the change. The position of an entry (also of an entry of a NetBeans Platform module) is overwritten as follows:

```
<attr name="Menu/Edit/CopyAction.shadow/position" intvalue="15"/>
```

Use the complete file path of the affected entry in front of the attribute name `position`.

File Types

There are different file types provided within the System Filesystem. You will be confronted with them at some points again, when developing your application. For example, registering actions and menu entries in the layer file. I want to explain two frequently used file types in the following sections.

instance Files

Files of the type *.instance* in the System Filesystem describe objects of which instances can be created. The filename typically describes the full class name of a Java object (for example, `com-galileo-netbeans-module-MyClass.instance`), which makes a default constructor or static method create an instance.

```
<filesystem>
   <file name="com-galileo-netbeans-module-MyClass.instance"/>
</filesystem>
```

An instance is created by using the File Systems and Data Systems API, as follows:

```
FileObject o = FileUtil.getConfigFile(name);
DataObject d = DataObject.find(o);
InstanceCookie c = d.getLookup.lookup(InstanceCookie.class);
c.instanceCreate();
```

If you want a more convenient name for an instance, the full class name can be defined by using the `instanceClass` attribute. This enables much shorter names to be used:

```
<file name="MyClass.instance">
   <attr name="instanceClass" stringvalue="com.galileo.netbeans.module.MyClass"/>
</file>
```

In classes that do not have a parameterless defauld contructor, create the instance via a static method defined by the `instanceCreate` attribute.

```
<file name="MyClass.instance">
   <attr name="instanceCreate" methodvalue="com.galileo.netbeans.module.MyClass.getDefault"/>
</file>
```

Doing so, the `FileObject` of the entry is passed to the `getDefault()` method, if declared so in the factory method signature. With this `FileObject` you can read self-defined attributes, for example. Assuming, you want to define the path of an icon or any other resource in the layer file as an attribute:

```
<file name="MyClass.instance">
   <attr name="instanceCreate" methodvalue="com.galileo.netbeans.module.MyClass.getDefault"/>
   <attr name="icon" urlvalue="nbres:/com/galileo/icon.gif"/>
</file>
```

The `getDefault()` method with which an instance of the `MyClass` class can be created could thus look like the following:

```
public static MyClass getDefault(FileObject obj) {
   URL url = (URL) obj.getAttribute("icon");
   ...
   return new MyClass(...);
}
```

As you will recognize, I specified the path with a `urlvalue` attribute type. Therefore, a URL instance is directly delivered. In addition to the already known attribute types `stringvalue`, `methodvalue`, and `urlvalue` there are several others. We will take a closer look at them in the section *Layer File*.

One or more instances of a certain type can also be generated by a Lookup rather than via an `InstanceCookie`, as previously shown. Contrary to what was shown previously, you can easily produce multiple instances of a certain type and create the Lookup for a particular folder of the System

Filesystem. Using the `lookup()` or the `lookupAll()` method, one or more instances (if several have been defined) can be delivered.

```
Lookup lkp = Lookups.forPath("MyFolder");
Collection<? extends MyClass> c = lkp.lookupAll(MyClass.class);
```

Such a `Lookup` is used in Chapter 10 to extend the content menu of the top component with your own actions defined in the layer file. The basic class or the interface can be user defined by the `instanceOf` attribute in the layer file. This allows a more efficient working of `Lookup` and avoids `Lookup` having to initiate each object in order to determine from which base class the class will inherit, or which interface it implements. This way, the `Lookup` can directly create only the desired object type instances.

If the class `MyClass` from the prior entry implements, for example, the `MyInterface` interface, we can complete the entry as follows:

```
<file name="com-galileo-netbeans-module-MyAction.instance">
   <attr name="instanceOf" stringvalue="com.galileo.netbeans.module.MyInterface"/>
</file>
```

shadow Files

.shadow files are a kind of link of reference to an *.instance* file. They are mainly used when singleton instances of objects, as with actions, are used. These are defined by an .instance file in the `Actions` folder. An entry in the `Menu` or `Toolbars` folder then refers to the action by using the .shadow file. A .shadow file refers to files in the System Filesystem as well as to files on disk. This way, the Favorites module stores its entries. The path to the .instance file is specified by the attribute `originalFile` (see Listing 3-5).

***Listing 3-5**, Connecting a .shadow File with an .instance File*

```
<folder name="Actions">
  <folder name="Window">
    <file name="com-galileo-netbeans-module-MyAction.instance"/>
  </folder>
</folder>
<folder name="Menu">
  <folder name="Window">
    <file name="MyAction.shadow">
      <attr name="originalFile"
            stringvalue="Actions/Window/com-galileo-netbeans-module-MyAction.instance"/>
    </file>
  </folder>
</folder>
```

Attribute Values

Mostly, file entries are expanded by attributes in the System Filesystem. The attributes can have quite different meanings, though. For example, the name of a registered action is defined by the attribute. It is possible to define class names or factory methods by attributes elsewhere. The System Filesystem provides a series of types with which the attribute values become available; with those attributes, the different attribute values can be read out. The most common types and their meaning are shown in Table 3-1. All types are listed in the Filesystem DTD (see Appendix).

Table 3-1. *Types of Attribute Values and Their Meanings*

Type	Meaning / Usage
`intvalue`	Specification of numerical values, e.g., for the location of files and folders by the `position` attribute.
`boolvalue`	Specification of `true` or `false`, e.g., for defining whether an action shall be executed asynchronously by the `asynchronous` attribute.
`stringvalue`	Specification of textual constants, e.g., naming an action.
`urlvalue`	Specification of paths, e.g., assigning an icon of an action as follows: `nbres:/com/galileo/netbeans/module/icon.gif`
`methodvalue`	With this you can define a factory method with which a class shall become instantiated. To get there, specify the code name base, the class name, and the method as follows: `com.galileo.netbeans.module.MyClass.getDefault`
`newvalue`	Use this type when a class shall be instantiated with its default constructor. Specify the class name with code name base: `com.galileo.netbeans.module.MyClass`
`bundlevalue`	Using this type, the attribute value is read from a properties bundle. This is very helpful with names of actions, for example. Like this, you can outsource text constants so they can be localized easier. The key follows the complete name of the bundle, separated by the # symbol: `com.galileo.netbeans.Bundle#CTL_MyFirstAction`

A factory method which is indicated by the type `methodvalue` has different signatures:

```
static MyClass factoryMethod();
static MyClass factoryMethod(FileObject fo);
static MyClass factoryMethod(FileObject fo, String attrName);
static MyClass factoryMethod(Map attrs);
static MyClass factoryMethod(Map attrs, String attrName);
```

You get access on the according entry in the System Filesystem by the `FileObject` parameter in a simple way. With this object you can access the referring attributes (compare the section *File Types*). You directly get the attributes when using a `Map` as parameter of your factory method.

Accessing System Filesystem

Of course it it possible that your own module folder, files, and attributes are used from the layer file to provide module extension points to others. You get access on the System Filesystem by the following call for reading entries:

```
FileUtil.getConfigRoot();
```

This call provides the root of the System Filesystem as `FileObject`. From there, you can access the whole content. You can also use the following method if you want to access a certain path in the System Filesystem:

```
FileUtil.getConfigFile(String path);
```

In Chapter 10 I will show you an example of how to define your own entries in the layer file, read them, and thus provide an extension point to other modules.

Creating Modules

Now it is time to create your first module. A good introduction to module development is also offered by the sample applications already integrated in the NetBeans IDE. For simplicity's sake, you will just design a single module here.

First, create a NetBeans Platform Application or a Module Suite. This way you will be able to execute and test the module easier, and you can define dependencies to your own modules and libraries (see the "Defining Dependencies" section). Afterward, you can even create an independent rich client distribution (see Chapter 35).

The NetBeans IDE provides a wizard to help you apply a NetBeans Platform Application project.

1. Start the NetBeans IDE and then select *File* ➤ *New Project…* in the menu. Different project categories are displayed in the dialog that appears on the left side. Select *NetBeans Modules* there. Then choose the project type *NetBeans Platform Application* on the right side (see Figure 3-3).

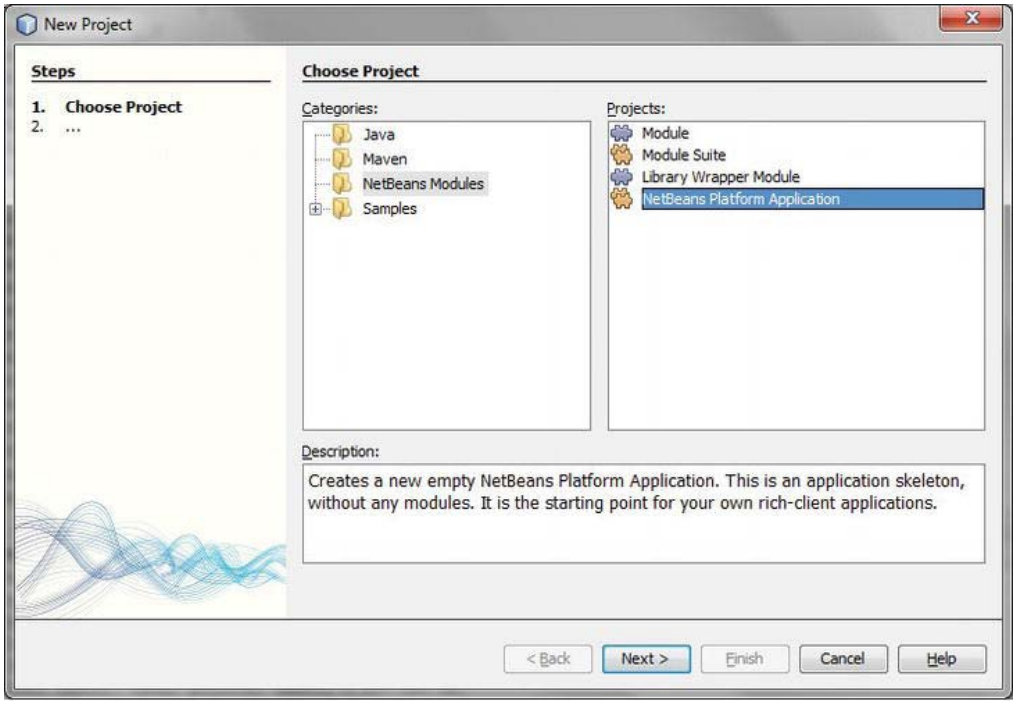

Figure 3-3. *Creating a new NetBeans Platform application project*

2. On the next page, name the application (for example, My Application) and choose the location where the project is to be saved. The remaining fields can be left blank.

3. Click the *Finish* button to create the NetBeans Platform application project.

4. Now the first module can be created; another wizard is available for this task. Open the *File* ➤ *New Project …* menu. Choose the category *NetBeans Modules*, and then the project type *Module* on the right side.

5. Click the *Next* button to go to the next page, for naming the project. Enter here, for example, My Module, and then select the option *Add to Module Suite*, and select the previously created NetBeans Platform Application or Module Suite from the list.

6. On the last page, define the Code Name Base and a module display name. The default value for the Localizing Bundle can be kept. To make it complete, activate the option *Generate XML Layer* due to create a layer file. If it is not needed, it can be deleted later, together with the referring entry in the manifest file.

7. Click the *Finish* button and let the wizard generate the module, as shown in Figure 3-4.

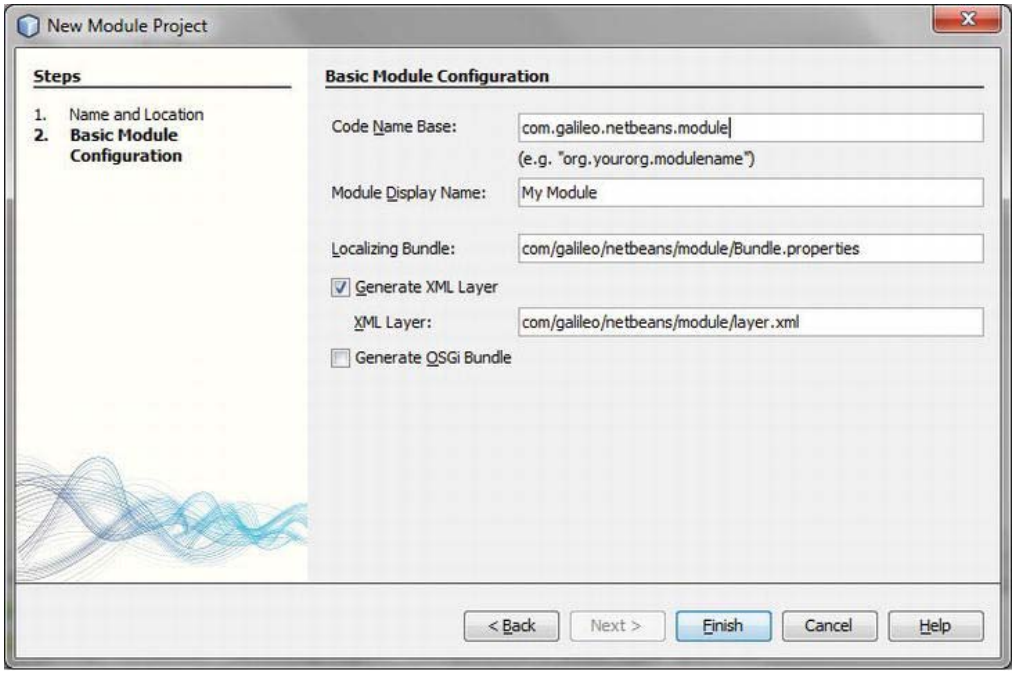

Figure 3-4. *Configuration of a new module*

Looking at the module in the Projects window, you see the folder *Source Packages*. At the moment, this folder contains only the files *Bundle.properties* and *layer.xml*. The file *Bundle.properties* only provides a localizing bundle for the information registered in the manifest file. The so-called *Layer Tree*, a special view, which you can find in the folder *Important Files*, is provided for the *layer.xml* file. It provides two different views. On the one hand, there is the folder <*this layer*>, in which only the content of your layer file is displayed. On the other hand, there is the folder <*this layer in context*>, in which the entries of the layer files of the modules (belonging to your NetBeans Platform application) are displayed. This view is represented as System Filesystem, too, as provided to the Platform during the runtime.

In this view entries of the module (in which you look at the folders) are displayed in bold. This gives you an overview of the most important default folders, and you can directly move, delete, or add entries. Furthermore, you can find the manifest file, which was also created by the wizard, in the folder *Important Files (see Figure 3-5)*.

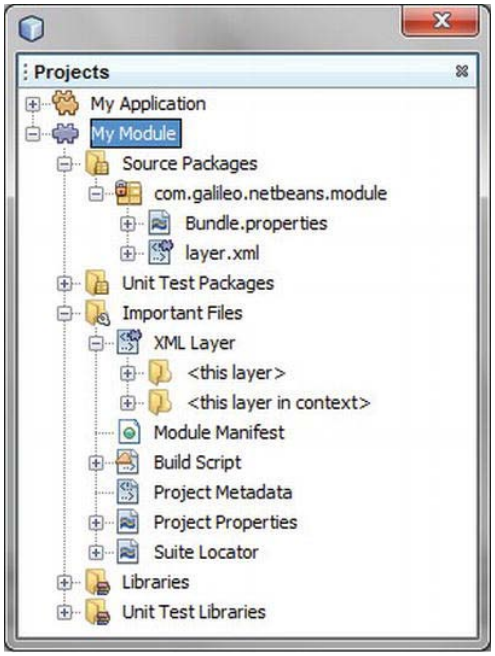

Figure 3-5. *Module in the projects window*

You can already start the created module as a rich client application. To do so, call *Run ➤ Run Main Project (F6)* in the menu or *Run* in the context menu of your NetBeans Platform application project. (See Figure 3-6.)

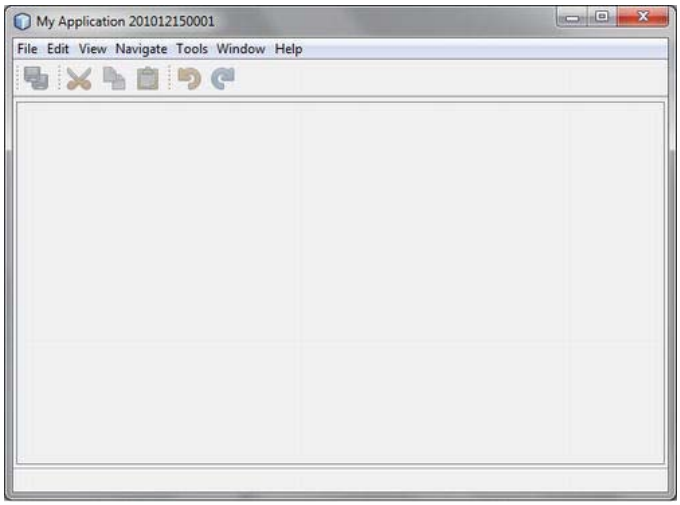

Figure 3-6. *The basic structure of your NetBeans Platform application*

We applied the basic structure of a NetBeans Platform application in just a few steps. In the following chapters, we will equip our module with functionalities, such as windows and menu entries, step by step. In this way we will enrich the rich client application.

Versioning and Dependencies

To ensure that a modular system remains consistent and maintainable, it is crucial that the modules within the system prescribe the modules they need to use. To that end, the NetBeans Platform allows definition of dependencies on other modules. Only by defining a dependency can one module access the code from another module. Dependencies are set in the manifest file of a module. That information is then read by the module system when the module is loaded.

Versioning

To guarantee compatibility between dependencies, you must define versions; for example, the *Major Release Version*, the *Specification Version*, and the *Implementation Version*.These versions are based on the Java Package Versioning Specification and reflect the basic concepts of dependencies. You can define and edit dependencies in the Properties dialog of your module, which you can access via *Properties* ➤ *API Versioning* (see Figure 3-7).

First, define the *Major Release Version* in this window. This is the version notifying the user of incompatible changes, compared to the previous version of the module. Here, the slash is used to separate the code name base from the version within the manifest file:

```
OpenIDE-Module: com.galileo.netbeans.module/1
```

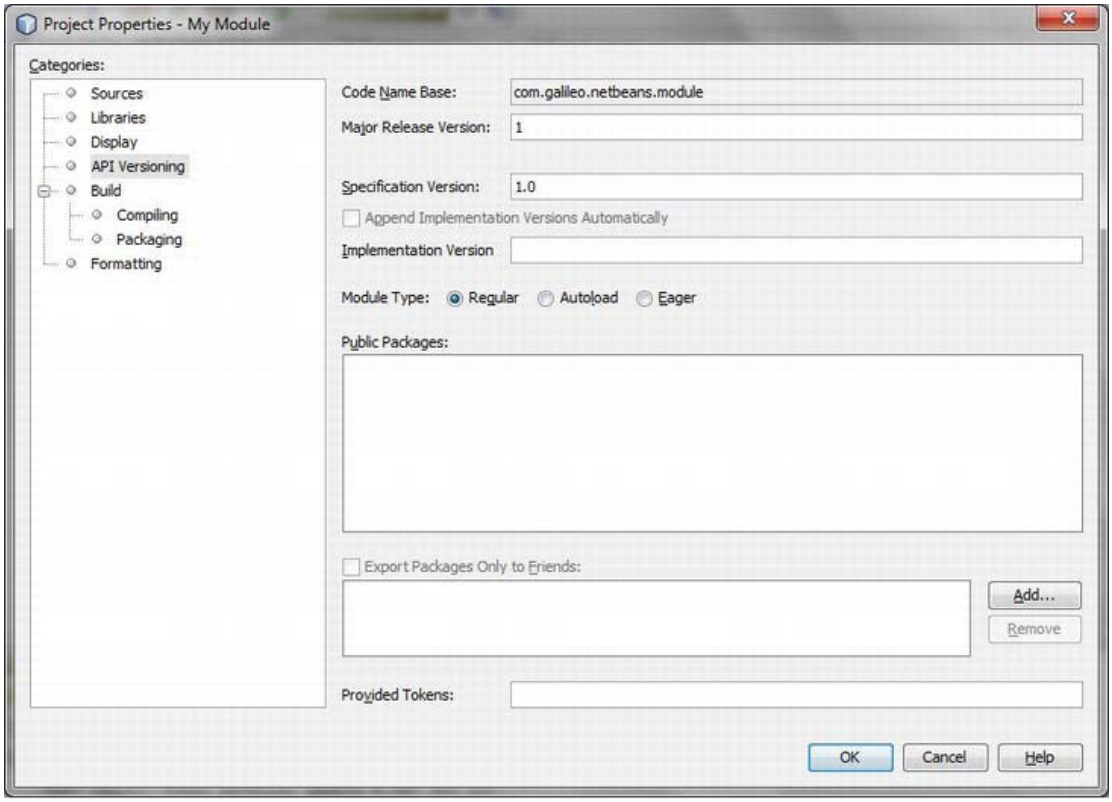

Figure 3-7. *Setting the module version*

The most important version is the *Specification Version*. The Dewey-Decimal-Format is used to define this version:

```
OpenIDE-Module-Specification-Version: 1.0.4
```

The *Implementation Version* is freely definable text. Typically, a timestamp is used, providing the date and time. In that way, you determine it is unique. If not explicitly set in the Properties dialog of the module, the IDE adds the implementation version when the module is created, using the current timestamp, set within the manifest file:

```
OpenIDE-Module-Implementation-Version: 200701231820
```

On the other hand, if you define your own implementation version in the Properties dialog, the IDE adds the `OpenIDE-Module-Build-Version` attribute with the current timestamp.

In the list of Public Packages, all packages in your module are listed. To expose a package to other modules, check the box next to the package you want to expose. In doing so, you define the API of your module. Exposed packages are listed as follows in the manifest file:

```
OpenIDE-Module-Public-Packages:
   com.galileo.netbeans.module.*,
   com.galileo.netbeans.module.model.*
```

To restrict access to the public packages (for example, to allow only your own modules to access the public packages), you can define a module's *Friends*. You define them beneath the list of public packages in the API Versioning section of the Properties dialog. These are then listed as follows in the manifest file:

```
OpenIDE-Module-Friends:
   com.galileo.netbeans.module2,
   com.galileo.netbeans.module3
```

Defining Dependencies

Based on these various versions, define your clear dependencies. To that end, three different types of dependencies are available: a module depends on a module, a package, or a version of Java.

NO ACCESS WITHOUT DEPENDENCIES

To use classes from another module, including the NetBeans Platform's own modules, you must first define a dependency, as described in the following sections. That means, if you use a NetBeans Platform class in your module and the code editor cannot find the desired class, the problem can normally be fixed by simply setting a dependency on the module that provides the class.

Module Dependencies

You define and edit module dependencies via *Properties* ➤ *Libraries*, as shown in Figure 3-8.

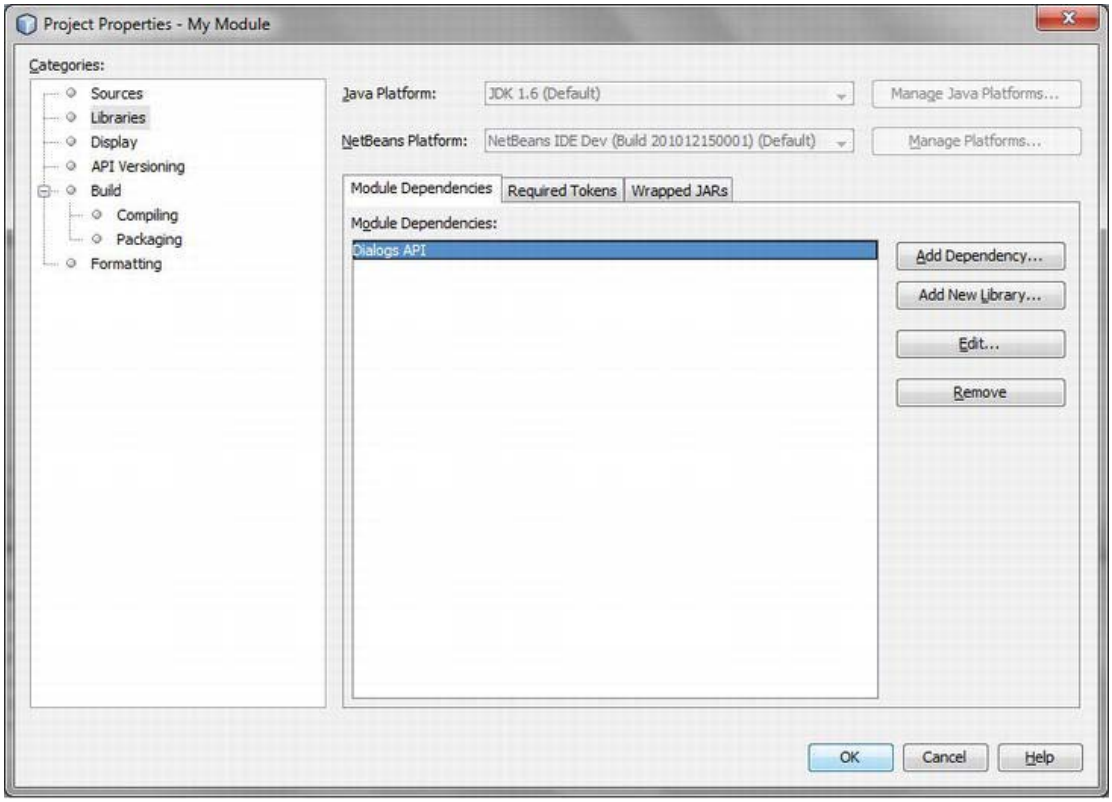

Figure 3-8. *Definition of module dependencies*

In this window, use *Add Dependency…* to add dependencies to your module. The NetBeans module system offers different methods to connect dependencies to a particular module.

In the simplest case, no version is required. That means there should simply be a module available, though not a particular version (although, where possible, you still specify a version):

```
OpenIDE-Module-Module-Dependencies: com.galileo.netbeans.module2
```

In addition, you may require a certain specification version. In this case, the module version should be greater than version 7.1. This is the most common manner of defining dependencies:

```
OpenIDE-Module-Module-Dependencies: org.openide.dialogs > 7.1
```

If the module on which you want to depend has a major release version, it must be specified via a slash after the name of the module:

```
OpenIDE-Module-Module-Dependencies: org.netbeans.modules.options.api/1 > 1.5
```

Additionally, you may also specify a range of major release versions:

```
OpenIDE-Module-Module-Dependencies: com.galileo.netbeans.module3/2-3 > 3.1.5
```

To create tight integration to another module it is possible to set an *Implementation Dependency*. The main difference and the reason for this approach is to make use of all the packages in the module, regardless of whether the module has exposed them or not. A dependency of this kind must be set with care, since it negates the principle of encapsulation and the definition of APIs. To enable the system to guarantee the consistency of the application, the dependency must be set precisely on the version of the given implementation version. However, this version changes with each change to the module.

```
OpenIDE-Module-Module-Dependencies: com.galileo.netbeans.module2 = 200702031823
```

Select the required dependency in the list (see Figure 3-8) and click the *Edit…* button. As shown in Figure 3-9 you can set various types of dependencies.

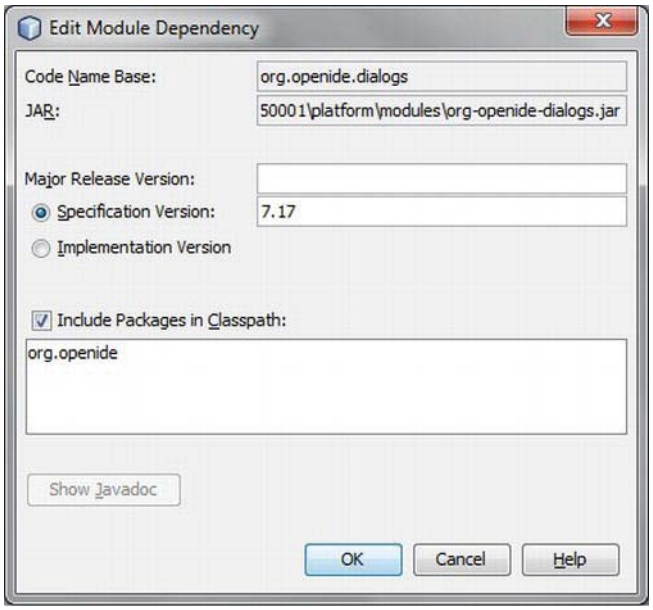

Figure 3-9. *Editing module dependencies*

Java Package Dependency

NetBeans lets you set a module dependency on a specific Java package. A dependency of this kind is set in the manifest file:

```
OpenIDE-Module-Package-Dependencies: javax.sound.midi.spi > 1.4
```

Java Version Dependency

If your module depends on a specific Java version, such as Java 6, you can also specify that in the module properties under *Properties* ➤ *Sources*, using the *Source Level* setting. Aside from that, you can require a specific version of the Java Virtual Machine:

```
OpenIDE-Module-Java-Dependencies: Java > 1.6 VM > 1.0
```

You can require an exact version using the equal sign or require a version that is greater than the specified version.

Lifecycle

You can implement a so-called *Module Installer* to influence the lifecycle of a module and thus react on certain events. The Module System API provides the `ModuleInstall` class, from which we derive our own module installer class. Doing so, the following methods of the desired events can be overwritten. The following methods or events are available:

- *`validate()`*: This method is called before a module is installed or loaded. If necessary, certain load sequences, such as the verification of a module license, are set here. Should the sequence not succeed and the module not be loaded, an `IllegalStateException` can be thrown. This exception prevents loading or installing the module.
- *`restored()`*: This method is always called when an installed module is loaded. Here, actions can be initialized starting a module.
- *`uninstalled()`*: This method is called when a module is removed from the application.
- *`closing()`*: Before a module is ended, this method is called. Here, you can also test whether the module is ready to be removed or if there are still activities to be executed. If the return value is `false`, the module and the whole application is not ended, because this method is always called before ending a module. The application is just ended when all modules are set `true`. You can, for example, show the user a dialog to confirm whether the application should really be closed.
- *`close()`*: If all modules are ready to end, this method is called. Here, you can call the actions before shutting down a module.

■ **Note** Using these methods, consider whether the actions you are calling could be set declaratively instead. However, always check if the desired action could go a declarative way. In particular, in the cases of the methods `validate()` and `restored()`, consider that these methods influence the startup time of the whole application. For example, when services are registered, you could either use entries in the layer file or the Java Extension Mechanism (see Chapter 5). This way they are loaded at their first usage and doesn't extend the startup time of the application as a whole.

Listing 3-6 shows the structure of a module installer class.

Listing 3-6. *Structure of a Module Installer Class*

```
public class Installer extends ModuleInstall {
   public void validate() throws IllegalStateException {
      // e. g. check for a license key and throw an
```

```
        // IllegalStateException if this is not valid.
    }
    public void restored() {
        // called when the module is loaded.
    }
    public void uninstalled() {
        // called when the module is deinstalled.
    }
    public boolean closing() {
        // called to check if the module can be closed.
    }
    public void close() {
        // called before the module will be closed.
    }
}
```

To record the state of the module installer class over different sessions, overwrite the methods `readExternal()` and `writeExternal()` from the `Externalizable` interface, which is implemented by the `ModuleInstall` class. There you store and retrieve necessary data. When doing so, it is recommended to first call the methods to be overwritten on the superclass. To let the module system know at startup if a module provides a module installer, and where to find it, register it in the manifest file:

```
OpenIDE-Module-Install: com/galileo/netbeans/module/Installer.class
```

Now you want to create your first module installer. The NetBeans IDE provides a wizard to create this file (see Figure 3-10). Go to *File* ➤ *New File…* and choose the file type *Installer / Activator* in the category *Module Development.*

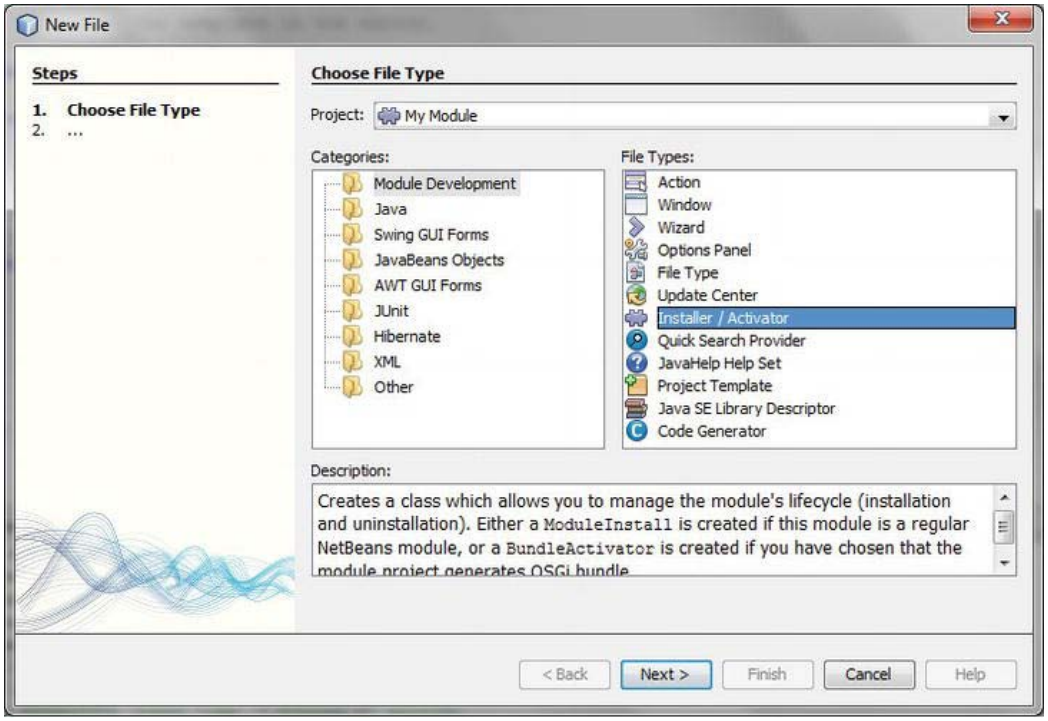

Figure 3-10. *Creating a module installer*

Click *Next* and then click *Finish* on the next page to complete the wizard. Now the module installer class is created in the specified package and registered in the manifest file. You just need to overwrite the required methods of this class. For example, you can overwrite the `closing()` method to show a dialog confirming whether the application should really be shut down. You can implement this as shown in Listing 3-7.

Listing 3-7. *Dialog for Shutting Down the Application*

```
import org.openide.DialogDisplayer;
import org.openide.NotifyDescriptor;
import org.openide.modules.ModuleInstall;

public class Installer extends ModuleInstall {
   public boolean closing() {
      NotifyDescriptor d = new NotifyDescriptor.Confirmation(
         "Do you really want to exit the application?",
         "Exit",
         NotifyDescriptor.YES_NO_OPTION);
      if (DialogDisplayer.getDefault().notify(d) == NotifyDescriptor.YES_OPTION) {
         return true;
      } else {
```

```
            return false;
        }
    }
}
```

Be aware that this module requires a dependency on the Dialogs API to be able to use the NetBeans dialog support. Defining dependencies was described previously in the section *Versioning and Dependencies*, while information about the Dialogs API can be found in Chapter 13.

To try this new functionality, invoke *Run ➤ Run Main Project (F6)*. When the application shuts down, the dialog is shown and you can confirm whether or not the application should actually be shut down.

Module Registry

Modules do not normally need to worry about other modules. Nor should they need to know whether other modules exist. However, it might sometimes be necessary to create a list of all available modules. The module system provides a `ModuleInfo` class for each module, where all information about modules is stored. The `ModuleInfo` objects are available centrally via the Lookup, and can be obtained there as follows:

```
Collection<? extends ModuleInfo> modules = Lookup.getDefault().lookupAll(ModuleInfo.class);
```

The class provides information such as module name, version, dependencies, current status (activated or deactivated), and the existence of service implementations for the current module. Use the `getAttribute()` method to obtain this information from the manifest file. To be informed of changes, register a `PropertyChangeListener`, which informs you both of the activation and deactivation of modules in the system (`ModuleInfo` object). You can also register a `LookupListener` that informs you of the installation and uninstallation of modules. For example, a listener could be defined as shown in Listing 3-8.

Listing 3-8. *Reacting on Changes in the Module System*

```
Lookup.Result<ModuleInfo> result = Lookup.getDefault().lookupResult(ModuleInfo.class);
result.addLookupListener(new LookupListener() {
   public void resultChanged(LookupEvent lookupEvent) {
      Collection<? extends ModuleInfo> c = result.allInstances();
      System.out.println("Available modules: " + c.size());
   }
});
result.allItems(); // initialize the listener
```

Using Libraries

When developing rich client applications, you will more than likely need to include external libraries in the form of JAR archives within your application. Since the whole application is based on modules, it is desirable to integrate the external JAR file in the form of a module. That has the advantage of setting dependencies on the module, enhancing the consistency of the application as a whole. You can also bundle multiple JAR files into a single module, after which you will no longer need to put the physical JAR files on the application classpath, as is normally done when developing applications.

Library Wrapper Module

To achieve the scenario just outlined, create a *Library Wrapper Module*. The NetBeans IDE provides a project type and a wizard for this purpose.

1. To create a new library wrapper project, go to *File ➤ New Project…*, and use the dialog shown in Figure 3-11 to choose the category *NetBeans Modules*, followed by the project type *Library Wrapper Module*.

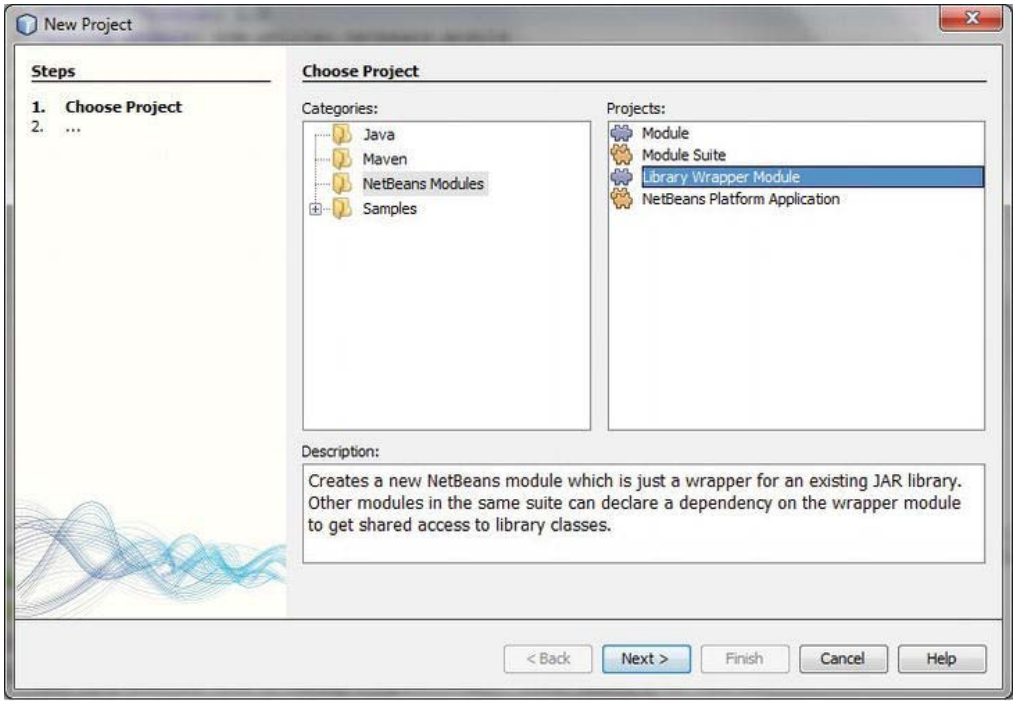

Figure 3-11. *Creating a library wrapper module*

2. Click *Next* to choose the required JAR files. You can choose one or more JAR files here (hold down the Ctrl key to select multiple JAR files). You are also able to add a license file for the JAR you are wrapping as a module.
3. In the next step, provide a project name, as well as a location to store the new module. Specify the Module Suite or Platform Application to which the library wrapper module belongs.
4. Click *Next* again to fill out the *Basic Module Configuration* dialog, as shown in Figure 3-12.

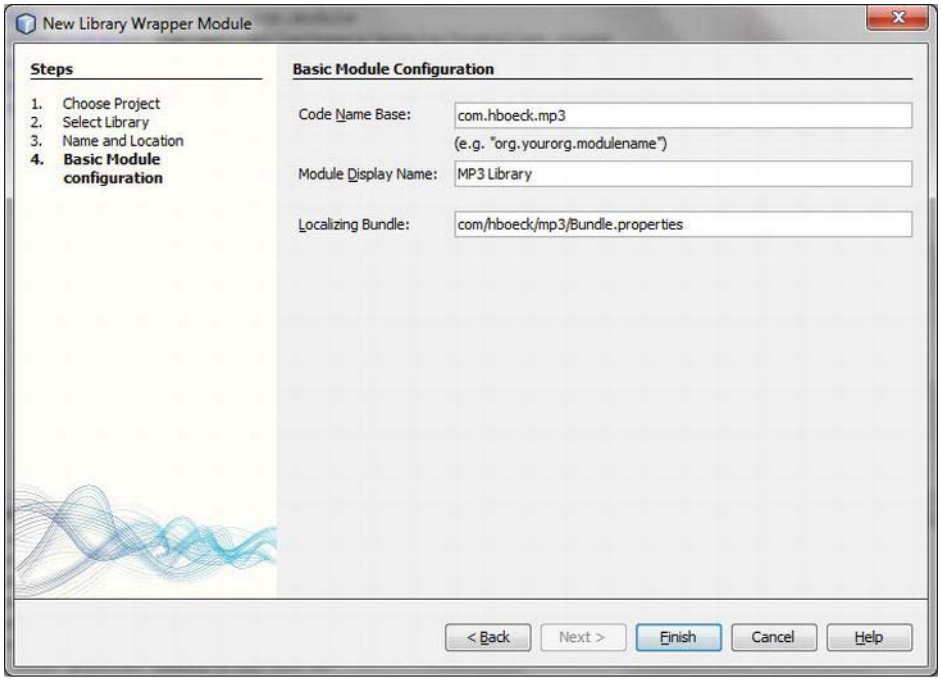

Figure 3-12. *Library wrapper module configuration*

5. This is where you can define the code name base. Normally this field is prefilled with the name of the selected JAR archive of the read code name base. Furthermore, you can provide the module with a name and a Localizing Bundle to localize the module manifest information. With a click on the *Finish* button, you create the new project.

When you are looking at the newly created library wrapper module in the *Projects* window, and you additionally open the *Source Packages* folder, you will see that the *Bundle.properties* file of the manifest file is located here. The library, which is encapsulated by the module, was copied in the directory *release/modules/ext* of the project folder.

To understand how a library wrapper module works, take a look at the related manifest file which is found in the projects structure in the folder *Important Files*. Note that the manifest information, which is depicted in the Listing 3-9, may not be found directly in the manifest file. Certain information, such as the public packages, are just written when you build the module (when calling *Build Project*). To see the entire manifest file, create the module and then open the manifest file within the created module JAR archive (located in the *build/cluster/modules* directory of your NetBeans Platform application. You can see which packages are exposed in the properties of your library wrapper module under *API Versioning*. There, you can delete packages from the *Public Packages* list later.

Listing 3-9. *Manifest File of a Library Wrapper Module*

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
```

```
Created-By: 1.6.0-b105 (Sun Microsystems Inc.)
OpenIDE-Module: com.hboeck.mp3
OpenIDE-Module-Public-Packages:
   com.hboeck.mp3.*,
   com.hboeck.mp3.id3.*,
   ...
OpenIDE-Module-Java-Dependencies: Java > 1.4
OpenIDE-Module-Specification-Version: 1.0
OpenIDE-Module-Implementation-Version: 101211
OpenIDE-Module-Localizing-Bundle:
   com/hboeck/mp3/Bundle.properties
OpenIDE-Module-Requires: org.openide.modules.ModuleFormat1
Class-Path: ext/com-hboeck-mp3.jar
```

Two very important things have been accomplished by the wizard. On the one hand, it marked all packages of the library with the attribute **OpenIDE-Module-Public-Packages**, making all these packages publicly accessible. This is useful because a library is supposed to be used by other modules, too. On the other hand, the wizard marked the library (located in the distribution in the directory *ext/*) with the **Class-Path** attribute, putting it on the module class path. This way, the classes of the library can be loaded by the module classloader. The type *Autoload* was automatically assigned to the library wrapper module (see the section *Configuration File*). This way, it is just loaded when needed.

Adding a Library to a Module

It is advisable to always use a library wrapper module when integrating a library into an application, as seen in the preceding section. Creating a new module in this way for a third-party library adds to the value and maintainability of the application as a whole, because you can then set dependencies on the library with the module that wraps it. In some cases, it can be desirable to add a library to the existing module (your own application module). To do this is simple and works similarly to creating a library wrapper module.

To add a library, open the features of the desired module with *Properties* in the context menu. In the category *Libraries,* in which dependencies on other modules are defined, you find the tab *Wrapped JARs* on the right side. There you can add the wanted library with the *Add JAR* button.

Doing so, a `class-path-extension` entry is added to the *Project Metadata* file for each library. The path defined by the `runtime-relative-path` attribute is the path within which the library is located in the distribution (this is where it is automatically copied when creating the module). The location where the original of the library is located is specified by the `binary-origin` attribute. As you can see, it is the same directory as with the library wrapper modules. (See Listing 3-11.)

Listing 3-11. *Project Metadata File with Class Path Extension*

```
<class-path-extension>
   <runtime-relative-path>ext/com-hboeck-mp3.jar</runtime-relative-path>
   <binary-origin>release/modules/ext/com-hboeck-mp3.jar</binary-origin>
</class-path-extension>
```

With this entry into the project metadata file the library is copied into the *ext/* directory and is added to the manifest of the module with the entry `Class-Path: ext/com-hboeck-mp3.jar` when creating the module. In contrast to a library wrapper module, the packages of the library are not exposed. As a result, they can only be used by the module. (In most cases, this is the reason for the direct addition of the

library: it should not be made public). It is also possible to define the packages of the library as being public, which is automatically the case with a library wrapper module.

WHEN TO USE WHICH APPROACH?

Bear in mind that you should create a library wrapper module of a library whenever possible rather than directly adding libraries because of modularity and maintainability. As a rule, only add a library to a module directly, when the library is solely used by this module and if it is not a problem to distribute the library together with the module that uses it. Furthermore, note that you cannot load the same library from two different modules with the `Class-Path`. This could lead to unforeseen problems. Also, do not try to use the `Class-Path` attribute to refer to the module JAR archives or to libraries found in the NetBeans *lib/* directory.

Reusing Modules

Usually, you create a NetBeans platform application project for applications which are not that big, then add the whole logic of the application to this project in the form of modules. In case you then want to implement, for example, a big application for enterprises in a team, it can be useful to break down the application into multiple parts, each containing an amount of modules. For this purpose, the NetBeans IDE offers the opportunity to add both a single module or a complete cluster (folder with NetBeans modules) as dependency. So if you develop, for example, a series of base modules, whose functionality you want to use in multiple applications, it is best to use a module suite.

You can create a module suite with *File* ➤ *New Project…* ➤ *NetBeans Modules*. Within this module suite you can develop and test your base modules encapsulated from special application modules. Starting the build process of your modules, all modules are stored in a cluster. Then you can add this cluster to another NetBeans Platform project as follows: call *Properties* ➤ *Libraries* in the desired application. There, you find the button *Add Cluster…* (see Figure 3-13) with which you can select the cluster of the module suite.

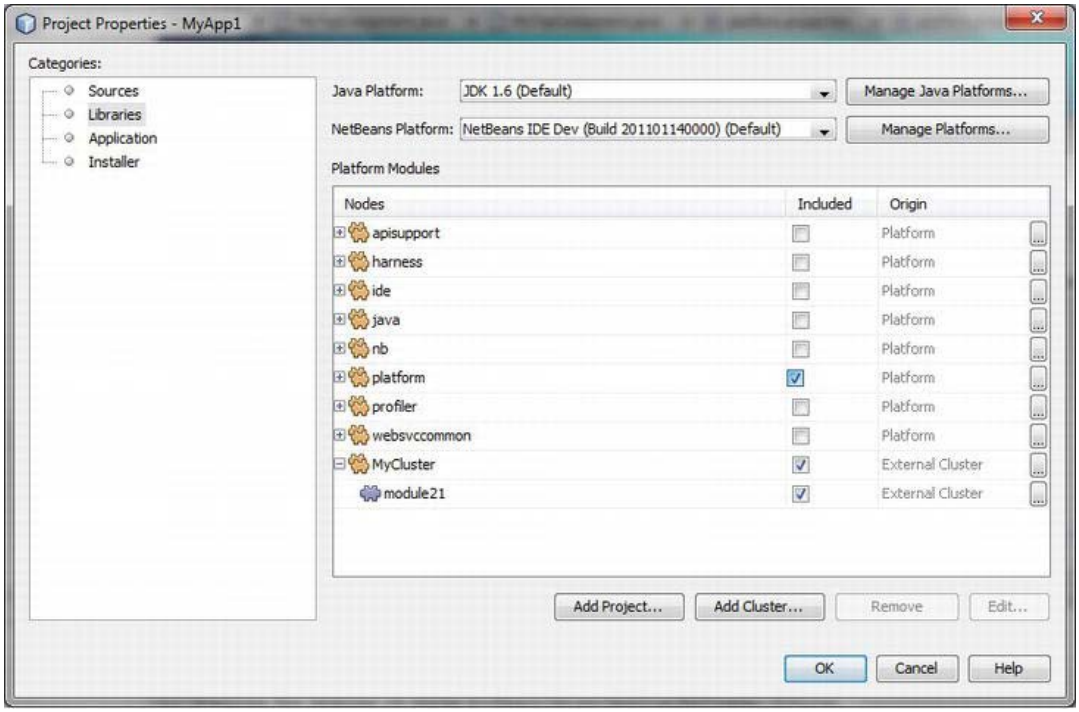

Figure 3-13. *Adding a complete cluster for reusing external modules.*

If you did not select a real cluster, the NetBeans IDE asks you to select the desired modules of the folder, and then creates a respective cluster. This means the NetBeans IDE automatically creates a cluster from a folder with modules.

Now, you can access further NetBeans modules from different NetBeans Platform applications with the possibility of reusing modules, as described. This way, you can, for example, implement your generic modules centrally, independent from special applications.

MODULE SUITE VS. NETBEANS PLATFORM APPLICATION

This book is primarily about developing independent applications based on the NetBeans Platform. This is why it constantly uses the *NetBeans Platform Application* project type to create an application in the following chapters. For this project type only the NetBeans Platform modules are provided, by default, since it will become an independent application and not an extension of the NetBeans IDE. However, you have the possibility of accessing any modules of the NetBeans IDE. For this purpose, call *Properties* ➤ *Libraries* in your NetBeans Platform application project. There you can activate the desired modules. Your own modules can only define dependencies on modules which are activated there. You can switch between a NetBeans Platform application and a module suite anytime under *Properties* ➤ *Libraries*. The branding support and creating an installer are logically just provided with a NetBeans Platform application.

Summary

In this chapter you learned how the underlying module system of NetBeans Platform applications is structured and how it functions. The module system is part of the runtime container. First, we looked at the structure of a NetBeans module. You learned about the many configuration options that are defined in the manifest file. In addition to the manifest file, a module optionally provides a layer file. You learned how to make contributions to the whole application, via registration entries in a module layer file.

You created your first module, learned how modules use code from other modules, and explored the lifecycle of modules and how third-party libraries integrate in a module via a library wrapper module. Finally, you discovered how those kinds of modules work, and you got some hands on experience with them.

CHAPTER 25

Auto Update Services API

With its plugin manager the NetBeans Platform provides a useful tool to the user. With this tool, individual parts of applications (single modules or several related modules) can be installed, uninstalled, activated, deactivated, or updated. It is possible to directly access these functionalities by means of the Auto Update Services A PIand create really different applications.

It is possible to access all existing modules because of the class `UpdateManager`. One module (unit) is represented by the class `UpdateUnit`; this class is a kind of wrapper for different elements. Such elements could be

- The installed module itself
- The module which had been installed for the current locale setting
- The predecessor of an installed update (backup)
- A list of available updates (in update centers)
- A list of available localizations (in update centers)

The named elements are each represented by the class `UpdateElement`. This class provides the name, the category, the author, or the icon of the module, among other things. Actions such as installing or uninstalling are managed by an `OperationContainer` instance. To create such an instance a factory method is provided for each action.

The following sections demonstrate the usage of the Auto Update Services API by means of typical applications.

Automatic Update in the Background

Business applications often require that updates are automatically installed on the clients without user interaction. This ensures that all users work with the same and the latest version of the application.

Search Updates

The first step is to search the modules that must be installed. The search takes place in all registered update centers. That means in the first delivery of your application, at least one update center should be registered. Via this update center the new modules should be included. You can register an update center quite easily by means of the NetBeans IDE. To do so, just call *File ➤ New File ➤ Module Development ➤ Update Center*. You can find more information on this in Chapter 36.

Search for both the latest and the updated modules by the method shown in Listing 25-1. It is important to make the desired (or all) update provider(s) load information from the update centers

about the provided modules. In this example you determine all providers via the `UpdateUnitProviderFactory` and execute its `refresh()` method. After that, you determine all provided modules that can be updated or reinstalled via the `UpdateManager`, which also takes all providers into account. To do so, filter with `!unit.getAvailableUpdates().isEmpty()` all modules which are already installed and which are not updated. Now you just have to distinguish between a new module and an updated module using the method `getInstalled()`. If it says `null`, a version of this module is not yet installed; that means it is a new module. According to this query you add the modules to the appropriate list.

Listing 25-1. *Searching New and Updated Modules*

```
private List<UpdateElement> install = new ArrayList<UpdateElement>();
private List<UpdateElement> update = new ArrayList<UpdateElement>();

public void searchNewAndUpdatedModules() {
   for (UpdateUnitProvider provider : UpdateUnitProviderFactory.
           getDefault().getUpdateUnitProviders(false)) {
      try {
         provider.refresh(null, true);
      } catch (IOException ex) {
         LOG.severe(ex.getMessage());
      }
   }
   for (UpdateUnit unit : UpdateManager.getDefault().getUpdateUnits()) {
      if (!unit.getAvailableUpdates().isEmpty()) {
         if (unit.getInstalled() == null) {
            install.add(unit.getAvailableUpdates().get(0));
         } else {
            update.add(unit.getAvailableUpdates().get(0));
         }
      }
   }
}
```

In the approach shown in Listing 25-1, all update centers are queried by the `UpdateManager`. You can also restrict the search to a separate provider to ensure that the modules are automatically installed out of a dedicated update center, as shown in Listing 25-2. You determine the desired update provider by the `UpdateUnitProviderFactory` via name (according to the definition in the layer file). In contrast to the previous approach, you do not search for the update manager then, but determine it directly on the provider.

Listing 25-2. *Searching Modules in a Special Update Center*

```
private static final String UC_NAME = "com_galileo_netbeans_module_update_center";

public void searchNewAndUpdatedModulesInDedicatedUC() {
   for (UpdateUnitProvider provider : UpdateUnitProviderFactory.
           getDefault().getUpdateUnitProviders(false)) {
      try {
         if (provider.getName().equals(UC_NAME)) {
            provider.refresh(null, true);
```

```
            for (UpdateUnit u : provider.getUpdateUnits()) {
               if (!u.getAvailableUpdates().isEmpty()) {
                  if (u.getInstalled() == null) {
                     install.add(u.getAvailableUpdates().get(0));
                  } else {
                     update.add(u.getAvailableUpdates().get(0));
                  }
               }
            }
         } catch (IOException ex) {
            LOG.severe(ex.getMessage());
         }
      }
   }
```

Installing and Restarting Updates

Operations on modules are executed via an `OperationContainer`. It is possible to add any amount of `UpdateElement` instances to this container. Please note that each operation container is responsible for one certain action. That means in this case that two different containers are needed. These are created via the referring factory methods `OperationContainer.createForInstall()` and `OperationContainer.createForUpdate()`. Adding the elements does not depend on the type. This is why you create a helper method for your purposes. (See Listing 25-3.)

Listing 25-3. *Adding the Update Elements to an Operation Container*

```
public OperationContainer<InstallSupport> addToContainer(
        OperationContainer<InstallSupport> c,
        List<UpdateElement> modules) {
   for (UpdateElement e : modules) {
      if (container.canBeAdded(e.getUpdateUnit (), e)) {
         OperationInfo<InstallSupport> operationInfo = c.add(e);
         if (operationInfo != null) {
            c.add(operationInfo.getRequiredElements());
         }
      }
   }
   return container;
}
```

The helper method shown in Listing 25-3 first checks whether a module is compatible with the container. For each module this is checked by the method `canBeAdded()`. In that case you add the module; an `OperationInfo` instance is delivered when the module has not been found in the container yet. This instance helps adding the needed dependencies from the current module by means of the method `getRequiredElements()`. In this example, this call could be skipped, because all available modules are added anyway. However, with the method `getBrokenDependencies()` you could check whether a dependency cannot be fulfilled.

In addition to filling the container with elements, downloading and installing modules works the same way; a helper method is useful for this step. An `OperationContainer` is passed to this method.

Through this operation container the modules can then be installed. After that you download, check, and finally install the modules by the methods `doDownload()`, `doValidate()`, and `doInstall()` of the `InstallSupport` instance. If the method `doInstall()` returns an `OperationSupport.Restarter`, a reboot is necessary to finish the installation. However, you do not want to reboot at this time, and instead leave the decision about when to reboot to the user and so use the method `doRestartLater()`. (See Listing 25-4.)

Listing 25-4. *Downloading and Installing the Module and Afterward Notifying the User*

```
public void installModules(OperationContainer<InstallSupport> container) {
   try {
      InstallSupport support = container.getSupport();

      if (support != null) {
         Validator vali = support.doDownload (null, true);
         Installer inst = support.doValidate(vali, null);
         Restarter restarter = support.doInstall(inst, null);

         if (restarter != null) {
            support.doRestartLater(restarter);
            if (!isRestartRequested) {
               NotificationDisplayer.getDefault().notify(
                  "Die Anwendung wurde aktualisiert",
                  ImageUtilities.loadImageIcon("com/galileo/netbeans/module/rs.png", false),
                  "Click here to restart",
                  new RestartAction(support, restarter));
               isRestartRequested = true;
            }
         }
      }
   } catch (OperationException ex) {
      LOG.severe(ex.getMessage());
   }
}
```

The user still needs to be informed about the necessary reboot of the application; for this the notification displayer, which is integrated in the status bar of the NetBeans Platform, works well. The hint is displayed to the user in a balloon. The practical advantage of this is that you can add an action which is displayed as a link. That way the user can directly reboot. You need the `InstallSupport` and the `Restarter` instance for the reboot. You give both to the `RestartAction` which then executes the reboot (as shown in Listing 25-5).

Listing 25-5. *Action Class to Execute a Reboot*

```
private static final class RestartAction implements ActionListener {
   private InstallSupport support;
   private OperationSupport.Restarter restarter;

   public RestartAction(
           InstallSupport support,
           OperationSupport.Restarter restarter) {
```

```
      this.support = support;
      this.restarter = restarter;
   }

   @Override
   public void actionPerformed(ActionEvent e) {
      try {
         support.doRestart(restarter, null);
      } catch (OperationException ex) {
         LOG.severe(ex.getMessage());
      }
   }
}
```

Automatically Starting Installation

Finally you have to start the automatic execution of the process in the background. One possibility is to install the updates when starting the application. You can do this in a simple way by a warm-up task. This would look like the methods which were implemented into the whole as shown in Listing 25-6.

Listing 25-6. *Executing the Automatic Update Installation by a Warm-Up Task*

```
import org.netbeans.api.autoupdate.InstallSupport;
import org.netbeans.api.autoupdate.OperationContainer;
import org.netbeans.api.autoupdate.OperationException;
import org.netbeans.api.autoupdate.OperationSupport;
import org.netbeans.api.autoupdate.UpdateElement;
import org.netbeans.api.autoupdate.UpdateManager;
import org.netbeans.api.autoupdate.UpdateUnit;
import org.netbeans.api.autoupdate.UpdateUnitProvider;
import org.netbeans.api.autoupdate.UpdateUnitProviderFactory;
import org.openide.awt.NotificationDisplayer;
import org.openide.util.RequestProcessor;
...
public class AutoInstaller implements Runnable {
   private static final Logger LOG = Logger.getLogger(AutoInstaller.class.getName());

   @Override
   public void run() {
      RequestProcessor.getDefault().post(
         new AutoInstallerImpl(), 1000);
   }

   private static final class AutoInstallerImpl implements Runnable {
      private List<UpdateElement> install = new ArrayList<UpdateElement>();
      private List<UpdateElement> update = new ArrayList<UpdateElement>();
      private boolean isRestartRequested = false;

      @Override
      public void run() {
         searchNewAndUpdatedModules();
```

```
            OperationContainer<InstallSupport> installContainer =
               addToContainer(OperationContainer.createForInstall(), install);
            installModules(installContainer);

            OperationContainer<InstallSupport> updateContainer =
                addToContainer(OperationContainer.createForUpdate(), update);
            installModules(updateContainer);
        }

        public OperationContainer<InstallSupport> addToContainer(
                OperationContainer<InstallSupport> container,
                List<UpdateElement> modules) { ... }

        public void installModules(
               OperationContainer<InstallSupport> container) { ...}

        public void searchNewAndUpdatedModules() { ... }
    }

    private static final class RestartAction
            implements ActionListener { ... }
}
```

A warm-up task is executed asynchronously when starting the application. Only the `Runnable` interface has to be implemented. It is also possible to hold back the start for a certain time by the `RequestProcessor` class. The warm-up task only has to be registered in the layer file as follows:

```
<folder name="WarmUp">
   <file name="com-galileo-netbeans-module-AutoInstaller.instance"/>
</folder>
```

Deactivating Modules Automatically

In the example showing how to automatically update an application in the previous section you got to know the Auto Update Services API for finding, downloading, installing, and updating modules. Furthermore, the API makes it possible to activate or deactivate certain modules. There are very interesting applications for this. For example, you can switch off certain modules or functionalities for certain user groups (by the login information). The method is quite similar to that described in the previous section. First, you determine all available modules (including the already installed modules) via the `UpdateManager`. By the additional filter `UpdateManager.TYPE.MODULE` you ensure that only application modules and not localization modules are delivered. Via the `getInstalled()` method you check whether you are dealing with an installed module. This way new modules are filtered. Search for modules to deactivate in the remaining modules that are activated. Deactivating works via the unique code name base. (See Listing 25-7.)

Listing 25-7. *Searching for Certain Active Application Modules*

```
List<String> modules = Collections.singletonList("com.galileo.netbeans.module3");
OperationContainer<OperationSupport> cont = OperationContainer.createForDirectDisable();
for (UpdateUnit unit : UpdateManager.getDefault().getUpdateUnits(UpdateManager.TYPE.MODULE)) {
```

```
   if (unit.getInstalled() != null) {
      UpdateElement elem = unit.getInstalled();
      if (elem.isEnabled()) {
         if (modules.contains(elem.getCodeName())) {
            if (cont.canBeAdded(unit, elem)) {
               OperationInfo<OperationSupport> operationInfo = cont.add(elem);
               if (operationInfo != null) {
                  cont.add(operationInfo.getRequiredElements());
               }
            }
         }
      }
   }
}
```

For deactivating modules you create a referring container with the factory method `OperationContainer.createForDirectDisable()`. Add the desired modules to it. Before doing so, check again whether the modules are compatible with the container. If the module is successfully added you can add all modules that are dependent of the module to deactivate via the `OperationInfo` instance. Finally, ensure that the container is not empty. Then deactivate the modules via the method `doOperation()` (see Listing 25-8.)

Listing 25-8. *Deactivating Application Modules*

```
if (!cont.listAll().isEmpty()) {
   try {
      Restarter restarter = cont.getSupport().doOperation(null);
   } catch (OperationException ex) {
      LOG.severe(ex.getMessage());
   }
}
```

Summary

You can access all functions, which your plugin manager provides, using the Auto Update Services API, making it possible to realize quite interesting applications. In this chapter you learned how you can execute an application-specific automatic update in the background and how you can disable modules programmatically.