The NetBeans™ Tools Platform

A Technical Overview

The NetBeans™ integrated development environment (IDE) is open source, modular, standards-based and integrated. Because it is written in the Java™ language, it will run on any platform with a Java Virtual Machine that is compliant with the Java 2 Platform. The fully capable NetBeans IDE is a strong platform that can be used to deliver useful developer tools. This enables vendors to concentrate on their core competencies rather than on becoming full-blown IDE vendors. Because the NetBeans IDE is modular, developers can:

- Add modules which provide editing, debugging, syntax coloring, error highlighting, and other functions for the Java language and others. The IDE can work with C, C++, UML, IDL, XML, and others, as well as with the Java language.
- Switch any of the IDE modules on or off. By switching off unneeded modules, the IDE consumes less memory and no longer offers unnecessary information and actions.
- Write modules that add new features or replace functionality in the IDE.
- Update the IDE online through the Update Center.

Recently added to the NetBeans source code is a new, standards-based Metadata Repository that makes it easier to build modules to support other programming languages, in addition to enhancing performance and refactoring-related features. By supporting standardized models for metadata in a language-neutral way, it also makes it easier to integrate third-party products, such as UML tools.

Overview

The NetBeans IDE is based on a thin core that is responsible for basic services and infrastructure such as windowing, actions, and file management. This core implements the NetBeans OpenAPIs, to which the modules are written.

This core can be extended with plug-in modules written to the NetBeans OpenAPIs. In fact, the basic functions of the IDE (editor, Java language support, compilation execution, and debugging services) are implemented in modules written to these APIs. Virtually every aspect of the NetBeans IDE is extensible. At the center of the architecture, the APIs cleanly abstract the IDE's functionality, creating a modular, dynamic environment. There are more than 400 classes, shipped in 15 API sets, along with descriptions of how to use them and Javadoc™ documentation.

Key points about the NetBeans APIs are:

- They are standalone — none of the classes contained therein refer to any other parts of the IDE. There is no reference to code outside of the APIs, except to code contained in the standard Java runtime environment and a couple of basic libraries — such as XML parsing and regular expressions.
- Most APIs have what are essentially inner and outer components. This refers to the concept that APIs can be used to either create new functionality (called the service provider interface, or SPI), or use the functionality accessible through the API (the client API).
- The APIs create a dynamic environment — new functionality can be added or removed simply by adding or removing modules, without recompiling or even restarting the IDE, on any platform. The module author builds a module and the user installs and uses it.
- The APIs are provided in separate JAR files, and can be easily used as standalone libraries. For example, writing a Java application using the FileSystems library does not affect the application's independence. Instead, its functionality is enhanced, client code is simplified, and development time is reduced.

Core APIs

All of the NetBeans OpenAPIs are designed to be used to implement modules. Modules permit the IDE to be extended dynamically. By abstracting functionality into a well-defined API, module authors and users alike are assured that new functionality can be added quickly and easily. Modules may range in complexity from a single Java class, which may do something as elementary as adding a menu item, to new actions or analyses that can be performed on Java source files. As well, a module can contain a full-scale integration of a major external application, such as a Java profiling suite. All modules are distributed and installed as JAR files, with special entries in the JAR manifest that are recognized by the IDE.

The FileSystems API, for example, is a storage-neutral abstraction of a hierarchical file system. It is the only part of the IDE that knows about the physical storage of data, so other modules use it to access files. The result is that modules can be installed in the IDE to allow access to files inside a JAR archive, an FTP site, or a database-based repository, to list three examples, and the physical storage will be transparent to all other parts of the IDE.

Note that modules can publish additional APIs. Module authors may choose to build a public interface to modules they develop, so others can add to or extend the functionality of their products.

Modules

This API provides a standard way to install additional functionality in the IDE. A module is a JAR file containing the classes that implement new functionality, and a manifest that identifies it to the IDE as a module and describes the tasks necessary to install it. Conceptually the Modules API is based on the Java Extension Mechanism. Features include module versioning and cross-compatibility checks, as well as Web-based download and installation of modules into a running IDE. This API also specifies the location of JavaHelp™ documentation for the modules. More broadly considered, the Modules API includes descriptions of how a module is actually installed and manipulated.

Nodes

Nodes form a presentation layer which uniformly presents objects in the IDE. Nodes exist in a hierarchical structure, which functions as a dynamic object model through which most functionality in the IDE is accessed. As an extension of the JavaBeans™ model, nodes add support for dynamic properties, actions, cookies, and hierarchy. A node does not (and should not) contain data itself, but represents a DataObject or other type of object within the IDE. Many disparate parts of the IDE are exposed as nodes, from configuration options and source files, to thread stacks in the debugger. The Explorer is the default way of viewing and interacting with nodes.

Nodes are not static — they are live components of the IDE. For example, actions taken in one part of the system will frequently cause open Explorer views to refresh and display the updated node structure.

DataSystems

The DataSystems API is built around an abstraction called a DataObject. The DataObject associates data stored either in a FileSystem or as a FileObject with specific capabilities, such as editing or compiling. DataObjects form a layer above THE FileObjects layer, transforming the file-based view of the world into a object-oriented view.

Developers use this API primarily to cause the IDE to recognize new data types, such as UML model files, add user-visible actions to nodes representing a custom or standard type, implement templates, and so on. Where appropriate, DataObjects may also represent logical groupings of physical files. DataObjects are controlled by both inner and outer APIs, which are bidirectional. This enables module authors to manipulate data objects supplied by others, or create new functionality of their own.

Loaders associate files together into groups, assigns types to data, creates data objects from files, and defines how cookies attach behavior to these objects. *Cookies* function as a lookup pattern for finding or querying the specific functionality or capabilities of Nodes and DataObjects. Using cookies removes the requirement that all interfaces presented by the cookie holder — usually a data object or node — must be implemented by the Java class of the object. Cookies are also used in the Nodes API.

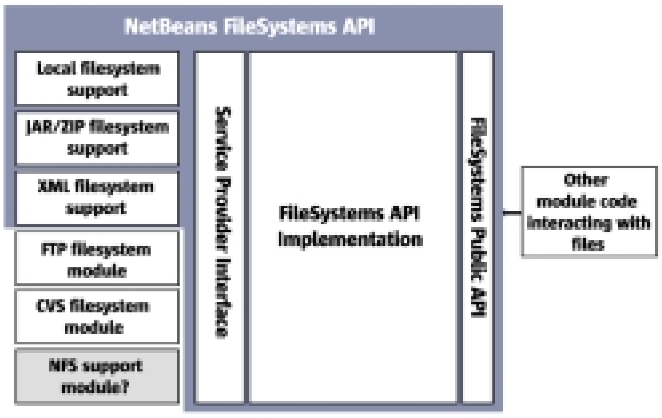

Figure 1: The NetBeans FileSystems API in relation to other modules. The area in blue contains those elements contained in the default NetBeans distribution. The model for other APIs is similar.

FileSystems

This API provides a storage-independent abstraction of files and file-like objects. It is made up of objects and classes that represent files, and is used to display files and directories. It can also display the file entries contained in JAR and ZIP archives, and supports integration with the version control systems.

The FileSystems API permits module authors to access file-like objects in a uniform manner. For example, you may access a file and be unaware of whether it is stored on local disk in a repository, in an auxiliary directory, or in a JAR archive. Alternately, you may want to implement a custom file system. For example, a vendor tool being integrated into the IDE may handle its own local or remote storage of files in a special fashion; using this API, the rest of the IDE will be able to seamlessly work with these files.

Explorer

The Explorer serves as a user interface for viewing the hierarchy of nodes. The Explorer offers a tree-like view of the node hierarchy, though various visual representations — ListView, IconView, Menu, TableView, and so on — are also available. In addition the Explorer provides a view of the properties associated with each of the nodes. Developers may compose the Explorer view so that the nodes can communicate with each other, enabling customization and flexibility in the overall interface.

Actions

The Actions API is a standard representation of the actions a user can invoke. It provides an interface to such UI elements as toolbars, menus, and keyboard shortcuts — allowing third parties to create context-sensitive actions that can be invoked in more than one way.

Options

The Options API provides a place for modules to expose a configuration interface to the user. It offers persistent user-level options in Global Settings or existing Project Settings. Options installed into the system are automatically saved to disk when the IDE exits, and reloaded when it starts up again.

Execution

This API provides a way to run applications, including runtime options. It controls the running of user code, whether it is in applets, applications, or non-standalone components such as JPanels. The API permits execution of user classes (including handling classloader issues), and allows a custom execution engine — say for profiling — to be integrated. Examples include running a simple Java applet, installing a servlet into a Web server host, and deploying Enterprise JavaBeans™ components into an application server.

Compiler

This API controls the execution of compilers in the IDE. It provides ways to add custom compilers, invoke existing compilers, associate compilers with specific files, monitor the progress and status of compilers, and clustering compiler requests into ordered jobs. In general, the Compiler API is a comprehensive mechanism for assembling output products from input files. It offers various controls with internal and external dependencies. For example, it can be used to compile Java code, or recreate a JAR file.

The MetaData Repository

The Metadata Repository (MDR) provides support for modules that need to create, store and retrieve metadata. Metadata refers to data describing the structure and characteristics of program elements or data; for example, the structure and method signatures of a Java class file. Another example would be a model of Java class structures, indicating the set of members any element can contain, the fact that classes can contain inner classes and methods, and so on.

The MDR contains an implementation of the Meta Object Facility (MOF) — an abstract language for describing metamodels defined by the OMG. The set of constructs used by the MOF for metamodeling is based on the Unified Modeling Language (UML) model.

The MDR offers a number of benefits:

- It provides support for developers who implement modules which generate or use metadata; for example, modules supporting a programming or modeling language. The MDR is based on open standards, including MOF, XMI, and OCL, and this support enables the IDE to interoperate with other tools that support these standards. For example, many UML modelers already support XMI.

- Given a language model, the MDR will generate all the interfaces representing an API for support of that language within the IDE. It provides a language-neutral standard, and saves developers the effort of writing their own APIs for each language. Since all MDR APIs are generated, they have a similar look — developers can become very familiar with the API by examining the metamodel.
- Some operations in the IDE will be faster when implemented using metadata, such as locating all references to a method or class within a body of code, or searching for all subclasses of a given class. The MOF standard can be used to integrate various types of tools for modeling, code generation, code analyses, dependency management, and so forth.
- Because of the reflective package contained in the MOF, it is possible to write generic, language-neutral tools for working with metadata, such as class browsers, search tools, and others.
- Objects can listen to events fired by the MDR, such as class creation, class or method renaming or signature changes. For example, when a designated event is detected, modules can apply code changes or offer to refactor the affected code.

Supporting New Languages

Module developers who want to add support for new languages or other types of metadata do not need to design and implement a new API — instead, they can focus on creating the metamodel. The following steps are involved in supporting a new language or type of metadata:

- First, create the metamodel. The MDR provides an MOF modeling tool to make this easier to do, or metamodels can be created using UML and converted to MOF. The resulting MOF model is saved in an XMI file. For some languages, such as UML, an MOF-complaint XMI model already exists. Creating the MOF is not difficult, and should be within the reach of anyone with a basic understanding of UML class diagrams.
- Next, load the metamodel XMI file into the MDR and generate an API for accessing metadata that follows this model. This process is called MOF-to-Java mapping, and is specified in the Java Metadata Interface (JMI) specification (JSR-40). The MDR contains a mapping tool to make this a simple step. The output of this process is a set of Java interfaces.
- The MDR is also able to generate an XML DTD for any metamodel, using MOF-to-XML mapping as specified in the XMI specification. Combined with the ability to save metamodels as XMI, this makes it possible for the MDR to exchange metadata and metamodels with any MOF or XMI-compliant repository.
- Finally, the developer must provide an implementation of the API. This is also facilitated by the IDE. If an operation such as counting the number of fields on a class is implemented in the metamodel, an implementation will need to be supplied. For most cases no implementation of it must be written. Developers do not need to implement getters and

setters of attributes, or define references between objects. The bytecode implementing these aspects of the generated API is created by the MDR when the new module is installed in the IDE, including the metadata persistence mechanism. This allows developers to focus on implementing the tool and module semantics.

Note that in addition to attributes, the MOF allows modeling of metaobject operations. A completed language-support module should include the metamodel in XMI, the generated interfaces, and classes that implement them.

Creating Metadata

In the NetBeans IDE, primary data such as source files are represented by DataObjects inside a FileSystems repository similar to the UNIX® model of mountable file systems. The DataSystems API supports synchronizing DataObjects with their physical storage, such as load and save operations.

Querying Metadata

The JMI interfaces — the generated APIs — are the main method used to programmatically query or modify the metadata if the developer is familiar with the metamodel. If there is no prior knowledge of the metamodel, the reflective interfaces of the MOF are used for querying and modifying metadata. An extended version of OCL serves as declarative language for querying. OCL is used primarily for checking constraints as they are defined in metamodels, but it can also be used for querying the metadata in the MDR.

MDR Storage Mechanism

The default storage mechanism for the MDR is a b-tree based index file. It is possible to to implement other storage mechanisms and provide them in the form of IDE modules. For example, a JDBC™ API-based mechanism could be created to store metadata in a more robust relational database system.

Summary

Although this document focused on the core NetBeans APIs, including the new MDR capabilities, the complete NetBeans IDE contains the necessary modules for a comprehensive solution. This enables tool developers to focus on developing specific functionality, without worrying about the rest of the infrastructure.

The NetBeans tools platform is developed as an open source project because open source is the ideal environment for building such a platform. The NetBeans IDE reflects the input from the community that uses and supports it. Due to its modular nature, new functionality can be added to the environment, independent of conventional release cycles. Developers can customize the IDE to fit the programmers, users, and the project. Users get what they want, when and how they want it.

NetBeans.org

At www.netbeans.org, developers may download stable or development builds of the NetBeans IDE. The source code is available via a CVS server at cvs.netbeans.org, under the Sun Public License, which is a variant of the Mozilla Public License. Users are encouraged to fix or change the code, post patches and contribute to the development of NetBeans. There are a number of mailing lists where development planning and discussion happen. Users can suggest new features, file bug reports, and write their own modules. The source code for contributed modules can be hosted at NetBeans.org.

NetBeans at JavaOne^{SM}

Come and visit us at JavaOne, Sun's 2001 Worldwide Java Developer Conference^{SM}, June 4-8 at the Moscone Convention Center in San Francisco, California. Please join our NetBeans engineers as they present *Developing Modules for the NetBeans API*, in addition to other sessions. They will also be available in the NetBeans booth on the show floor to answer your questions. For more information, please visit: www.netbeans.org/javaone.html

NetBeans

Sun microsystems

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
1.800.786.7638
1.512.434.1511