Developing OpenTools

JBuilder® 2005

Borland®
Excellence Endures™

Borland Software Corporation
100 Enterprise Way
Scotts Valley, California 95066-3249
www.borland.com

Refer to the file `deploy.html` located in the `redist` directory of your JBuilder product for a complete list of files that you can distribute in accordance with the JBuilder License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

For third-party conditions and disclaimers, see the Release Notes on your JBuilder product CD.

Printed in the U.S.A.

JB2005devot 8E7R1004
0405060708-9 8 7 6 5 4 3 2 1
PDF

Contents

Chapter

1

Developing OpenTools

JBuilder is designed to be an extremely open, extensible product. The concepts introduced in this document describe the basic organization of the product and some of the basic terminology necessary to get started. References in this introduction to the detailed documentation for each subsystem will get you directly to the information you need to tailor the IDE.

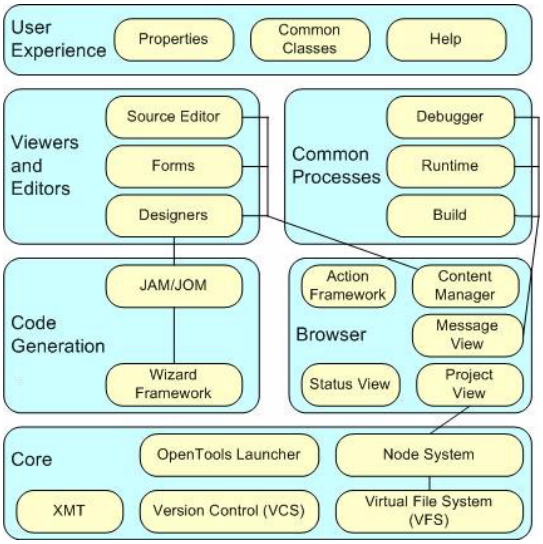
Packages start with `com.borland.primetime` or `com.borland.jbuilder`.

- Everything necessary for general IDE-style support is in the `primetime` packages.
- Behavior specific to handling Java code is in the `jbuilder` packages.
- The principle subsystems, or *categories*, provide an additional layer of logical grouping.

This structure makes the IDE extremely flexible.

Note The `primetime` package, the Primetime system, and the `PrimeTime` class are distinguished from each other by capitalization and font. This helps to clarify which element we mean when we're discussing how the Primetime system works, what is in the `primetime` package, or what can be done with the `PrimeTime` class.

Architectural Overview



Legend

Core	The Core category manages the management subsystems: OpenTools, files, collections of files, and projects.
Browser	The Browser category provides a consistent GUI framework for interacting with projects and files.
Viewers and Editors	The Editor and Designers provide a rich set of end-user tools for manipulating code. Each of these subsystems is extensible in its own way.
Common Processes	Primetime and JBuilder know how to compile and run basic projects, but the subsystems that manage the Build and Runtime processes are extensible to allow their behavior to be tailored.
Code Generation	The Wizard Framework provides a common look and feel for wizards and JOT provides the infrastructure to read, alter, and regenerate source code.
User Experience	The Properties System manages user preferences. Utility Classes support a consistent look and feel, providing a variety of additional libraries.

The Core

The Core category of subsystems in JBuilder provide functionality that's available to commandline tools, as well as extensions to the full-blown graphical IDE.

Core: OpenTools Loader

Document: Chapter 3, "OpenTools Loader concepts"

Package: `com.borland.primetime`

Classes: `Command`
`PrimeTime`

The OpenTools Loader is a critical mechanism that all OpenTools authors need at least a passing familiarity with. The loader is responsible for discovering all extensions

to the IDE and initializing them when the product is launched. The loader is also responsible for parsing the command line. Objects that implement the `Command` interface can be registered with `PrimeTime` to provide this command-line behavior.

Core: Virtual File System (VFS)

Document: Chapter 5, "VFS concepts"

Package: `com.borland.primetime.vfs`

Classes:
`VFS`
`Filesystem`
`Url`
`Buffer`

The Virtual File System provides seamless access to a number of different storage techniques, each defined by an implementation of the `Filesystem` interface. The contents of entries in ZIP files, physical files, and New "untitled" documents are provided as three `Filesystem` implementations in the core VFS implementation.

Much as File instances are used in Java to represent filenames, logical files in the VFS are identified using instances of a special `Url` class (note that this is different from the `java.net.URL` class.) All virtual file manipulation is done through `static` methods on the `VFS` class and passing one or more `Url` instances as parameters.

The VFS also supports an in-memory representation of a file as an instance of the `Buffer` class. Buffers can be hold changes before being discarded or written to disk, and, while a buffer exists, all VFS operations treat the contents of the buffer as the "current" state of the virtual file.

Core: Node System

Document: Chapter 21, "Adding a file node type and a viewer"

Packages:
`com.borland.primetime.node`
`com.borland.jbuilder.node`

Classes:
`primetime.node.Project`
`primetime.node.Node`
`primetime.node.FileNode`
`primetime.node.LightweightNode`
`primetime.node.ProjectStorage`
`jbuilder.node.JBProject`

The Node System provides the infrastructure for managing projects. All projects are instances descending from the generic Primetime `Project` class, and most projects in a JBuilder environment will actually be instances of the `JBProject` subclass. The project manages a hierarchy of individual `Node` objects, typically either `FileNode` objects that represent a file in the Virtual File System, or `LightweightNode` objects that represent logical concepts such as folders and packages.

The Node System provides a way to associate any number of property values with each `Node` in a project. Lastly, `ProjectStorage` implementations to be registered to read (and optionally write) foreign project file formats.

Core: The XMT

Document: Chapter 7, “XMT concepts”

Packages:
```
com.borland.primetime.xmt
com.borland.primetime.xmt.model
com.borland.primetime.xmt.creator
com.borland.primetime.xmt.configurator
com.borland.primetime.xmt.view.*
com.borland.primetime.xmt.visitor
com.borland.primetime.xmt.validator
```

The XMT (extensible modeling toolkit) is a framework for creating and assembling two-way tools on the Primetime platform. This means that designers, forms, and structure views, all using the same set of data, can maintain synchronization and load quickly when switching from one view to another. XMT provides abstract classes and interfaces for you to extend.

Core: Version Control System (VCS)

Document: Chapter 13, “Version Control System (VCS) concepts”

Packages:
```
com.borland.primetime.teamdev.vcs
com.borland.primetime.teamdev.frontend
```

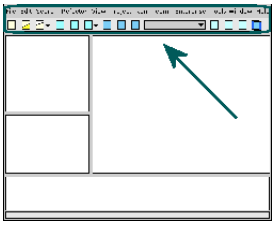
Classes:
```
vcs.VCS
vcs.VCSFactory
frontend.OpenTool
```

The VCS System allows you to integrate additional version control/revision management systems, building upon the available supporting classes. The History viewer and Commit Browser are two examples of the GUI which are driven by an implementation of the VCS class.

The Browser

The Browser is the metaphor used by the graphical IDE to display a workspace that allows the user to manipulate projects and the contents of individual files within a project. Each section of the browser is described briefly below.

Browser: Action Framework — Menus and Toolbars



Document: “Browser concepts” on page 23

Packages:
```
com.borland.primetime.ide
com.borland.primetime.actions
```

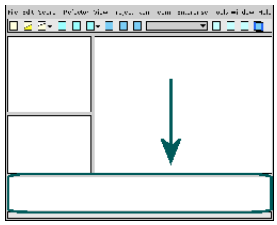
Classes: `ide.Browser`
`ide.BrowserAction`
`actions.UpdateAction`
`actions.StateAction`
`actions.DelegateAction`
`actions.ActionToolBar`
`actions.ActionMenuBar`

The main menu and toolbar are constructed dynamically from collections of Swing `Action` objects. The standard menu and toolbars can be extended with new `Action` objects, and new user interface additions can incorporate additional `Action`-based menu bars and toolbars.

Primetime makes extensive use of several convenient `Action` subclasses that offer more features and flexibility than the basic Swing `Action`:

- `UpdateAction` instances dynamically change their state. Primetime uses this to change the name of menu items dynamically, and to enable or disable actions as appropriate.
- `BrowserAction` instances always receive a reference to a valid `Browser` instance. Most Primetime actions need a reference to the browser that was active in order to get access to other OpenTools services.
- `StateAction` instances maintain a `boolean` state for toggled behavior. The menu automatically displays checkmarks next to each action whose state is `true`, and the toolbar gives these actions the appearance of being pressed.
- `DelegateAction` instances dynamically connect menus and toolbar buttons to new actions when the focus changes. These actions are used to implement the cut and paste actions but can just as easily be applied to other situations.

Browser: Message View



Document: “Message View concepts” on page 40

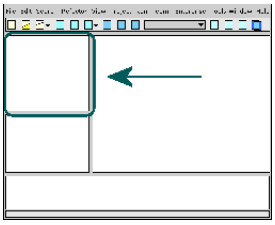
Package: `com.borland.primetime.ide`

Classes: `MessageView`
`MessageCategory`
`Message`

The Message View automatically appears when content is added to it, in order to communicate that content to the user. Content is organized into groups, each group visible as a tab at the bottom of the message area. Tabs can organize information differently: some tabs organize simple messages into lists or hierarchies, and some contain complex user interfaces created by defining a custom component.

The user has control over removal of tabs and may hide the entire message view at any time.

Browser: Project View



Document: “Project View concepts” on page 31

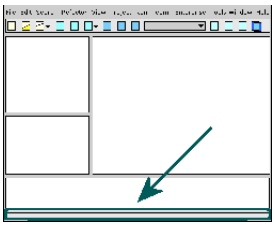
Package: `com.borland.primetime.ide`

Class: `ProjectView`

The Project View reflects the state of the active project, representing the available information visually through the Node System, including the nodes’ display hierarchy, display names, and icons. Most interaction with the project view happens *indirectly*, by defining new node types or modifying the node hierarchy through the Node System.

A `ProjectView` instance can be queried for the currently active project and the currently selected nodes. The list of open projects is also available, but is shared among all project views. Lastly, the context menu available on the project view can be directly influenced by registering `ContextActionProvider` objects with the `ProjectView` class.

Browser: Status View



Documents: “Status View concepts” on page 39

Package: `com.borland.primetime.ide`

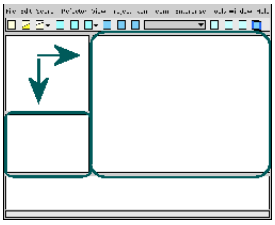
Class: `StatusView`

The Status View displays a single message in the bar at the bottom of the IDE. The message can be replaced on demand.

Informative messages about the last user action are typically displayed here. The status bar normally displays full descriptions for toolbar buttons and menu items, and can provide information about cursor location.

Longer messages, messages that the user must be guaranteed to see and respond to, and messages that the user needs to interact with, should use the Message View facility instead of the Status View.

Browser: Content Manager



Document: "Content Manager concepts" on page 26

Package: `com.borland.primetime.ide`

Classes: `Browser`
`NodeViewer`
`NodeViewerFactory`

The Browser manages the set of open files through the `ContentManager`, which manages the main display/worksurface area in the IDE. While OpenTools cannot replace the `ContentManager`, they can register viewer tabs through a static `Browser` method.

Each viewer tab is registered as a `NodeViewerFactory`. The factory evaluates whether or not its viewer type is available for a given node.

For available viewers, the factory constructs a `NodeViewer` implementation. The viewer instance provides the tab name and creates a pair of `JComponent` instances: one for the structure pane, one for the content area.

Viewers and Editors

Several standard viewers and editors are automatically registered with the `ContentManager` in Primetime. The source code editor, UI designer, UML, and Javadoc viewer are all examples of standard viewers included with the product. Some of these can, in turn, be enhanced by other OpenTools.

Viewers and Editors: Source Editor

Documents: Chapter 8, "Editor concepts"
Chapter 9, "Keymap concepts"

Packages: `com.borland.primetime.editor`
`com.borland.primetime.node`

Classes: `editor.Scanner`
`editor.EditorAction`
`node.TextFileNode`

The Source Editor in JBuilder is automatically displayed for descendants of `TextFileNode`. Each individual node can provide its own syntax highlighting with a `Scanner` implementation, a unique structure pane, and a CodeInsight implementation.

The editor can be customized by registering new user-selectable keymappings, which can borrow freely from our existing collection of actions implemented in `EditorActions`. Much of the extensible keymap functionality has been implemented in the IDE's GUI, but the hooks are still available for fun.

Common Processes

Common processes include running, debugging, and building.

Common Processes: Build System

Document: Chapter 6, “Build system concepts”

Packages:
```
com.borland.primetime.build
com.borland.jbuilder.build
```

Classes:
```
primetime.build.BuildProcess
primetime.build.Builder
primetime.build.BuilderManager
```

The Build System is responsible for scheduling a series of tasks whenever the user requests a compile or rebuild operation. Each task is then invoked in turn and may report errors or warnings that are accumulated on the Compile tab of the Message view. OpenTools can inject tasks into the build process, based on the type of nodes selected for the compile.

Common Processes: Runtime System

Packages:
```
com.borland.primetime.runtime
com.borland.jbuilder.runtime
```

Classes:
```
primetime.runtime.Runner
primetime.runtime.RuntimeManager
jbuilder.runtime.JavaRunner
```

Projects can be configured to run in one of a variety of ways, using Run|Configurations. The available runtime configurations are registered with a `RuntimeManager` as implementations of the `Runner` interface. Each `Runner` provides a name, a configuration panel, and behavior for running and debugging.

Most custom runners will want to share the standard Java runtime and debugging framework. The class `JavaRunner` provides a `Runner` implementation that can be subclassed and customized.

The Java debugger cannot be directly customized, but any custom runtime that uses the standard debugging framework can take full advantage of the debugger.

Code Generation

The IDE can be used to generate readable, predictable, organized code. These subsystems can be customized to extend that capability.

Code Generation: Wizard Framework

Document: Chapter 10, “Wizard concepts”

Packages:
```
com.borland.primetime.actions
com.borland.primetime.wizard
```

Classes:
```
wizard.BasicWizard
wizard.BasicWizardPage
wizard.WizardManager
```

The wizards all share a common look and feel, courtesy of the Wizard Framework. This framework can be used to create multi-page wizards by subclassing `BasicWizard` to provide one or more `BasicWizardPage` subclasses, and registering the resulting wizard instance with the `WizardManager`.

Code Generation: Java Abstract Modeler and Java Object Modeler (JAM/JOM)

Document: Chapter 16, "JBuilder JAM/JOM concepts"

Packages: `com.borland.jbuilder.jom`
`com.borland.jbuilder.jam`

JAM/JOM provides a full Java parsing and code generation engine. It starts where Java reflection leaves off.

JAM is a read-only, reflective layer that provides information from class or source files. JOM is a read/write layer that works with files and the constructs within those files, such as statements, blocks, expressions, etc. They work together to extract information from source or class files and then do something new with the information.

The User Experience

Users prefer a consistent interface that can remember their settings and help them out when needed. These subsystems simplify the process of providing a coherent, predictable user experience.

User Experience: Properties System

Document: Chapter 12, "Properties system concepts"

Packages: `com.borland.primetime.properties`
`com.borland.primetime.actions`

Classes: `properties.Property`
`properties.PropertyGroup`
`properties.PropertyManager`
`properties.PropertyPageFactory`
`properties.PropertyPage`

The Properties System provides a standardized way of reading and writing user preferences. Each setting is represented by an instance of a `Property` subclass, where different subclasses manage different kinds of storage and provide default values for the property.

Users typically interact with properties through pages on property dialogs. Registering a `PropertyGroup` with the `PropertyManager` gives your OpenTool a chance to provide a `PropertyPageFactory` whenever a property dialog is displayed for a given topic. Topics can include global properties, properties associated with subsystems such as the editor, and individual projects and files.

User Experience: Utility Classes

Documents: Chapter 14, "Util package concepts"
Chapter 11, "UI package concepts"

Packages: `com.borland.primetime.util`
`com.borland.primetime.ui`

There are a variety of utility classes that can help OpenTools maintain a consistent look and feel. Perhaps the key technique is to ensure that all tree-controls in your OpenTool descend from `SearchTree` to enable the type-to-search mechanism.

Chapter

2

OpenTools basics

There are no limits to what an OpenTool can do — it could register a new menu item, a new file type, a new viewer, or search the web for email and display its own user interface. OpenTools are written in Java and can do just about anything, though typically they will take advantage of methods exposed as a part of the OpenTools API to add features to the IDE.

You'll need to follow these steps to create your first OpenTool:

1. Create a new project.
2. Add the OpenTools SDK library to your project.
3. Create a new Java class.
4. Import referenced classes.
5. Define an initOpenTool() method.
6. Compile the OpenTool class.
7. Create an OpenTools JAR.
8. Test your new OpenTool.
9. Add an OpenTools JAR to the IDE.

Every time the IDE starts, it dynamically discovers the available OpenTools and gives each a chance to perform its own initialization at the appropriate time.

A single OpenTool can make as many, or as few, changes to the IDE as desired. Whether you create a single OpenTool class or many is up to you, but these tutorials will focus on adding selected functionality with individual OpenTools.

For additional information, see the `borland.public.jbuilder.opentools` newsgroup.

Create a new project

This is the first thing you do to create an OpenTool. You can actually define any number of OpenTools in a single project.

1. Select the File|New Project menu item and specify the location of your new project.

   For the purposes of this tutorial, we will assume that `c:/HelloOpenTool/HelloOpenTool.jpx` is the location of the project file.

2 Don't click the Finish button yet.

   If you already have, open Project|Project Properties|Paths — the next step can also be done there.

Add the OpenTools SDK library to your project

This library contains all of the classes that comprise the OpenTools API and are now available as part of your project classpath. Any meaningful OpenTool will need to use at least some of these definitions to extend the IDE.

You should be either in Step 2 of the Project wizard or in Project|Project Properties|Paths.

1 Click the Required Libraries tab.

2 Click the Add button.

3 Select the OpenTools SDK library.

4 Press OK to add it to your project.

5 Click Finish in the wizard or OK in the dialog (depending on which you used).

Now you have the project you need and the OpenTools SDK added as a required library. Time for code.

Create a new Java class

Use these values for this tutorial:

1 Select File|New Class.

2 Replace the default package with `example`.

3 Replace the class name with `HelloOpenTool`.

4 Click OK to create the new class definition and add it to your project.

Import referenced classes

During normal development you'd add the appropriate imports when you find a need for classes defined outside your package, but it just happens that we know what classes we will need ahead of time for this tutorial.

To compile the source code we'll create later, you will need to add the following set of import statements to your source code:

```
import com.borland.primetime.*;
import com.borland.jbuilder.*;
import com.borland.primetime.ide.*;

import javax.swing.*;
import java.awt.event.*;
```

The Swing and AWT classes are self-explanatory as part of the Java SDK, but the other packages are probably unfamiliar.

All packages under `com.borland.primetime` define a general-purpose framework for writing an IDE in Java, regardless of the target language. Packages under `com.borland.jbuilder` are specific to Java development. These two hierarchies contain many subpackages that will be introduced in later tutorials.

Generally you will not have to manually add imports. The ErrorInsight feature provides a mechanism to browse for a class being referenced so that the import can be generated. Also, when you cut/copy a block of code, the needed imports are discovered and are optionally inserted if required when you paste.

The name *PrimeTime* was the original code name for the all-Java IDE project, and it lives on in a class and package name in the OpenTools SDK.

Define an initOpenTool() method

Add the following method definition in the source for your `HelloOpenTool` class:

```
public static void initOpenTool(byte major, byte minor) {
    // Check OpenTools version number
    if (major != PrimeTime.CURRENT_MAJOR_VERSION) {
        return;
    }
}
```

This ensures that your OpenTool is being run with a compatible version of the OpenTools API before performing any initialization.

Create a Menu

Adding a menu to JBuilder is simple.

The first step is to create a static class that is a `BrowserAction`:

```
public static BrowserAction MyMenuAction =
    // A new action with short menu string, mnemonic, and long menu string
    new BrowserAction("My Menu", 'M',"Displays a simple confirm dialog") {
  public void actionPerformed(Browser browser) {
    JOptionPane.showConfirmDialog(null, "Menu Selected");
  }
};
```

If you aren't familiar with inner classes in general, and anonymous classes in specific, this code may look unusual to you. The code creates an instance of a subclass of `BrowserAction` and overrides an abstract method to define the behavior of the new action.

Next, we add a menu to the `Tools` menu by adding the following line to the `initOpenTool()` method:

```
JBuilderMenu.GROUP_Tools.add(MyMenuAction);
```

Our `initOpenTool()` method should now look like the following:

```
public static void initOpenTool(byte major, byte minor) {
    // Check OpenTools version number
    if (major != PrimeTime.CURRENT_MAJOR_VERSION) {
        return;
    }
    JBuilderMenu.GROUP_Tools.add(MyMenuAction);
}
```

A later tutorial will showcase JBuilder's menus and toolbars in more depth, but a quick look at the `JBuilderMenu` and `JBuilderToolBar` classes in the `com.borland.jbuilder` package will give you a starting point for experimentation.

Compile the OpenTool class

Select Project|Make Project to compile your project and ensure that there are no syntax errors in your code. The build system by default will automatically save modified files before compiling.

Create an OpenTools JAR

In order to create a deployable version of your OpenTool you need to create a JAR (Java Archive) file. The manifest of the JAR will help identify your OpenTool and tell JBuilder at what point in the startup cycle that it should be loaded.

The Archive Builder (File|New|Archive) helps you easily build this JAR and creates an empty manifest file for you to modify.

1. Select File|New|Archive.
2. Choose the OpenTool archive type.
3. Complete the wizard.

   You now have both an OpenTool node and a `classes.opentools` file which were added by the wizard to the project pane. The `classes.opentools` file provides the content for your JAR manifest.
4. Double-click `classes.opentools` in the project pane and add the following line to it:

   ```
   OpenTools-UI: example.HelloOpenTool
   ```

Note Manifest files must include a newline character at the end of every line, so be sure to press *Enter* when you're done!

Tip You can place as many fully-qualified OpenTools classes on a single line as you like, using spaces to separate the class names.

5. Select Project|Make Project.

   This makes the OpenTool node create the JAR, which you can deploy so that other users can have the benefit of your work.

Test your new OpenTool

There is a very convenient way to test your OpenTool without having first having to deploy your JAR or add your classes to JBuilder's classpath.

1. Select Run|Configurations.
2. Click the New button.
3. Set the type to OpenTool.
4. Press the OK button in both this dialog and the original dialog.

You can now run or debug your OpenTool in another copy of JBuilder from the information in this configuration.

Launch this copy of JBuilder using Run|Run Project, and look for the Help|Say Hello menu item. If it doesn't appear, check to make sure you've followed the steps above carefully.

Add an OpenTools JAR to JBuilder

JBuilder makes it easy to add OpenTools extensions. Just copy the `HelloOpenTool.jar` file you created in the last step into JBuilder's `lib/ext` directory, and you're done! The next time you start JBuilder it will automatically discover your OpenTool when it starts.

Just the beginning

You should begin to glimpse some of the flexibility afforded by the OpenTools mechanism. As you dig deeper into the OpenTools API, you'll see how you can

- Customize menu items and toolbars
- Define your own viewers, editors, and structure panes
- Extend the global properties system
- Associate property pages with individual files
- Create new run and debug systems
- Tweak the compilation process
- Enhance the context menus for individual files - and much, much more ...

Chapter

3

OpenTools Loader concepts

The IDE uses a dynamic discovery mechanism to load itself, rather than a hard-coded startup process. There are actually only two class files at the heart of Primetime: the `PrimeTime` class and the `Command` interface.

- The `PrimeTime` class provides a variety of methods for loading OpenTools and interpreting a command line. The actual process of discovering OpenTools is entirely automatic and is described in detail in the next section.
- The `Command` interface defines a self-describing command-line option. Each implementation of the interface provides a one-line description of the option, brief online help, and the actual command processor that handles requests.

There is no single class or interface that embodies the concept of an OpenTool. Each OpenTool can inherit from any class, though it must meet the following requirements:

- Be `public`
- Define a single method used to initialize the tool
- The OpenTool initialization method must have the following signature:

```
public static void initOpenTool(byte, byte)
```

This approach provides a very open-ended foundation for extensibility with minimal additional complexity. An extension to JBuilder need only be added to the classpath and it will automatically be initialized at the appropriate time.

OpenTool discovery

OpenTools are discovered by searching the current classpath for special manifest entries. An OpenTools entry in the JAR's manifest starts with the string `OpenTools-` and is followed by an OpenTools category, such as Core, Editor, Build, or UI. The value of the manifest entry must be a space-delimited list of fully-qualified class names, each of which must be a valid OpenTool.

The following manifest file describes a single OpenTool class in the "Core" category:

```
OpenTools-Core: com.borland.primetime.vfs.FileFilesystem
```

There is an alternative mechanism for finding OpenTools that is designed to be used only during development. This mechanism is described below under "Defining OpenTools during development" on page 20.

Using the initOpenTool() method

A valid OpenTool need only be a public class that implements a single `static` method matching the required signature:

```
public static void initOpenTool(byte, byte)
```

This method is invoked once when Primetime is loading, giving the OpenTool a chance to do whatever initialization is appropriate. The two parameters passed to the `initOpenTool()` method describe the version number of the OpenTools API implemented.

OpenTools API versions

The two parameters passed to an OpenTool are the major and minor version numbers. Generally, the major version number is incremented when the OpenTools API changes in a way that breaks compatibility with existing OpenTools, and the minor version number is incremented when features are added to the OpenTools API.

Each OpenTool must check the major version number before performing any registration tasks. The following template illustrates the standard implementation technique for this requirement:

```
public class examples.opentools.Sample {
  public static void initOpenTool(byte majorVersion,
      byte minorVersion) {

    // Make sure the OpenTools API is compatible
    if (majorVersion != PrimeTime.CURRENT_MAJOR_VERSION)
      return;

    // Perform OpenTool initialization
  }
}
```

For developers writing OpenTools for JBuilder, Primetime version numbers map to the following JBuilder version numbers:

Primetime	JBuilder
Primetime 4.5	JBuilder 8
Primetime 4.6	JBuilder 9
Primetime 4.7	JBuilder X
Primetime 4.8	JBuilder 2005
Primetime 4.9	JBuilder 2005, Upgrade 1

Writing and registering an OpenTool

The following is an example of a trivial OpenTool that does nothing but greet the user at load time:

```
public class examples.opentools.GreetUser {
  public static void initOpenTool(byte majorVersion,
      byte minorVersion) {

    // Make sure the OpenTools API is compatible
    if (majorVersion != PrimeTime.CURRENT_MAJOR_VERSION)
      return;
```

```
    // Perform OpenTool initialization
    String userName = System.getProperties().getProperty(
        "user.name");
    System.out.println("Greetings, " + userName + ".");
  }
}
```

OpenTools must be compiled and added to the classpath before starting, and the OpenTools discovery process needs to be able to find the class name at startup. To accomplish this, you'll need to create a manifest file that looks something like this:

```
OpenTools-Core: examples.opentools.GreetUser
```

Note that the manifest file must conform to the guidelines set forth by Sun:

- Entries are case sensitive.
- The manifest file cannot include blank lines.
- Each line must end with a line terminator.
- Only the last entry found with a given name is used.

To initialize more than one OpenTool in a single category using a single manifest file, you must use a space-delimited list of class names. The manifest file format will allow a single entry to span more than one line; however, each line continuation must begin with a space character *and this character does not count as a space between entries*. As a result, each new line with a class name should start with *two* spaces. The following example uses the text "<space>" to indicate the presence of a space character for clarity:

```
OpenTools-Core:<space>examples.opentools.OpenToolOne
<space><space>examples.opentools.OpenToolTwo
```

Suppressing existing OpenTools

There are times when you may wish to replace existing functionality by suppressing an existing OpenTool and providing a replacement that serves the same purpose. Any entry in the OpenTools list that starts with a minus character or exclamation point is treated as a request to suppress the OpenTool of the specified class name:

```
OpenTools-Core:<space>-examples.opentools.OpenToolOne
OpenTools-Core:<space>!examples.opentools.OpenToolOne
```

These are handled as follows:

- When a minus character is used, the category associated with the suppression request is ignored. The presence of the above entry in any manifest file on the classpath would prevent the OpenTool `examples.opentools.OpenToolOne` from ever being initialized.
- When an exclamation point is used, the OpenTool is only suppressed within the specified category. The same class listed in other categories may still allow the OpenTool to be initialized.

Registering command-line handlers

OpenTools in the "Core" category can extend the set of command line options recognized. The static method `registerCommand()` must be called once for each command line option.

Note It is important to remember that OpenTools in categories other than "Core" are typically loaded after the command line has been parsed; any commands they register will arrive too late to be recognized by the command line handler.

Each command registration requires two parameters:

- An associated option name in the form of a `String`
- An object that implements the `Command` interface

Note that the option name must be supplied without a leading hyphen, as shown in the following example:

```
PrimeTime.registerCommand("example", new ExampleCommand());
```

The preceding statement would allow the IDE to react to a command line that uses a leading hyphen to indicate the presence of an option. The remainder of the option name is case-sensitively matched to the appropriate command:

```
jbuilder -example
```

Each option on the command line results in a call to the associated Command instance's `invokeCommand()` method.

Commands that can accept one or more arguments should override `takesCommandArguments` and return `true`. When invoked, everything between the command line option and the following option is passed to these commands as a parameter.

In the following example, the handler for `example` will receive the three command arguments `one`, `two`, and `three`:

```
jbuilder -example one two three -example2 four
```

Additional methods defined by the Command interface require each command to return a one-line self-description from `getComandDescription`, and to print a more detailed description when `printCommandHelp` is called. The textual descriptions provided are available to the end user via the built-in `-help` command line option.

Default command-line handler

After invoking all other command-line handlers, JBuilder will always invoke the default command-line handler. This is the command most recently registered with a `null` option.

The OpenTools provided with JBuilder automatically register a default command handler when the `Core` OpenTools are initialized and start the full graphical IDE. Third-party OpenTools can take action during their `invokeCommand()` method if the normal IDE load process needs to be circumvented. The following statement is sufficient to prevent the graphical IDE from loading:

```
PrimeTime.registerCommand(null, null);
```

Defining OpenTools during development

Manifest entries can easily be included in JAR or ZIP archives, which is convenient for delivery but far from convenient during development. JBuilder also supports the notion of an "override" manifest which can be used in conjunction with a directory structure or an archive as an alternative to the true manifest. The override file has the same format as the normal manifest file, but JBuilder looks for it as an independent file in the same directory as each classpath entry. The override file is derived from the classpath entry by adding the suffix `.opentools`. For example:

`CLASSPATH` entry	Override manifest filename
`c:\classes\`	`c:\classes.opentools`
`c:\classes.jar`	`c:\classes.jar.opentools`

If found, the "override" manifest is used and the actual manifest file is ignored.

Debugging OpenTools

JBuilder has a special runtime type that can be used to test an OpenTool for a specific JBuilder release. You can access that runtime type by choosing Run|Configurations on the menu, then clicking the New button to create a runtime configuration, and selecting the OpenTool Run Type in the New Runtime Configurations dialog box.

The JBuilder startup process

JBuilder's `main()` method performs the following steps on startup:

1. Searches the classpath for OpenTools
2. Loads and initializes all OpenTools in the "Core" category
3. Parses the command line and invokes appropriate Command instances
4. Invokes the default Command instance

One of JBuilder's own core OpenTools registers a default command which, unless overridden while parsing and invoking commands from the command line, continues to load the remainder of the IDE as follows:

1. Displays the splash screen
2. Loads and initializes all OpenTools in the "UI" category
3. Initializes the Properties System
4. Creates and displays a Browser instance
5. Hides the splash screen

Debugging the startup process

JBuilder performs a special search for the command line switch `-verbose` before starting the OpenTools discovery process. If present, details of the search process and initialization of each OpenTool are printed to `System.out`. An abbreviated example of this output appears as follows:

```
Scanning manifest from X:\jbuilder\jdk1.2.2\lib\jpda.jar
Scanning manifest from X:\jbuilder\lib\AwtMotifPatch.jar
Scanning manifest from X:\jbuilder\lib\beandt3.1.jar
Scanning manifest from X:\jbuilder\lib\DBCSpatch.jar
Scanning manifest from X:\jbuilder\lib\dx3.1.jar
Scanning manifest from X:\jbuilder\lib\jbcl3.1.jar
Scanning manifest from X:\jbuilder\lib\jbuilder.jar
Scanning manifest from X:\jbuilder\lib\ext\PalmDeployer.jar
OpenTools discovered (110ms)
--- Initializing OpenTools-Core
OpenTool com.borland.jbuilder.JBuilderToolkit (170+701ms)
OpenTool com.borland.primetime.node.FolderNode (371+0ms)
OpenTool com.borland.primetime.node.ImageFileNode (20+20ms)
OpenTool com.borland.primetime.node.TextFileNode (0+0ms)
OpenTool com.borland.jbuilder.node.PropertiesFileNode (20+10ms)
OpenTool com.borland.jbuilder.node.ClassFileNode (10+0ms)
OpenTool com.borland.jbuilder.node.CPPFileNode (10+0ms)
OpenTool com.borland.jbuilder.node.HTMLFileNode (10+0ms)
--- OpenTools-Core initialized (1493ms total)
```

The first eight lines above describe the discovery process, detailing exactly which files are being used to gather a complete list of OpenTools available. The next line reports how long the complete discovery process took.

The next nine lines describe the process of loading each defined Core OpenTool in turn. The paired times shown in parentheses describe time to load the class and the time spent running the `initOpenTool()` method. These times should typically be less than 100ms combined, performing absolutely minimal initialization during startup. Time-consuming initialization should be deferred until the first time the feature is actually used.

The last line summarizes the combined time spent loading and initializing every OpenTool in a particular category. This value is the measured elapsed time rather than a sum of the times reported for individual OpenTools.

Defining additional OpenTools categories

Complex OpenTools may wish to provide hooks for other OpenTools to customize their behavior even further. The text editor in JBuilder is one such example, allowing OpenTools in the Editor category to register keymaps and other specialized behavior.

Why create a new category instead of just registering everything in the UI category? Because

- An OpenTool may need to know when all related extensions have been loaded
- The IDE will load faster and use less memory if large groups of OpenTools can be initialized only when needed

Choosing a category name

Borland reserves the right to use any category name that does not start with a package-like domain prefix. Third parties should use a category name based on one of their domain registrations to avoid name collisions. For example: Amazon.com might choose to use com.amazon.Book as a category.

Initializing a category

The static `initializeOpenTools()` method can be used to initialize all OpenTools belonging to a specified category. The method will return once each tool registered in the category has been initialized.

Chapter

4

IDE concepts

The IDE OpenTools API describes the application window with its menus and toolbars, as well as the panes that provide visibility into, control over, and feedback regarding your projects, files, and processes. The relevant concepts include:

- Browser
- Content Manager
- Project View
- Structure View
- Status View
- Message View

Browser concepts

`Browser` is the application IDE window. It constructs and establishes the relationships between all the other major UI elements:

- `BrowserMenuBar` (see `ActionMenuBar`)
- `BrowserToolBarPane` (see `ActionToolBarPane`)
- `ProjectView`
- `ContentManager`
- `StructureView`
- `MessageView`
- `StatusView`

You can obtain references to all these UI elements from a specific instance of `Browser` except for the menu bar and toolbar. To change the menu bar and toolbar, use the OpenTools API interfaces to modify the static `ActionGroup` objects for the application before the first `Browser` is instantiated. For more information about how OpenTools are loaded and initialized, see Chapter 3, “OpenTools Loader concepts.”

`Browser` fires events to each registered `BrowserListener` and each `ProjectGroupBrowserListener`. Such listeners can be registered for either a specific instance of `Browser` and statically for any `Browser` instance. You can use the convenient `BrowserAdapter` class to make implementing the `BrowserListener` interface easier.

Browser defines some `DelegateAction` objects that are usually tied to application toolbar buttons. The IDE pane that has the current focus can optionally provide its own context

sensitive implementation. There is a sample farther down in this document showing how to implement a delegated action.

These are the `Browser` delegate actions:

- `DELEGATE_ContextMenu`
- `DELEGATE_Copy`
- `DELEGATE_CustomEditorAction`
- `DELEGATE_Cut`
- `DELEGATE_Delete`
- `DELEGATE_GotoLineAction`
- `DELEGATE_Paste`
- `DELEGATE_Redo`
- `DELEGATE_Refresh`
- `DELEGATE_SearchAgain`
- `DELEGATE_SearchCombo`
- `DELEGATE_SearchFind`
- `DELEGATE_SearchIncremental`
- `DELEGATE_SearchReplace`
- `DELEGATE_SelectAll`
- `DELEGATE_Undo`

`Browser` and the other IDE components are not loaded during command line parsing. For example, JBuilder will parse the **-build** command line option, do the build, and terminate the application without ever creating a `Browser` instance.

Adding and removing menu bar groups

The `BrowserMenuBar` class is an extension of the `ActionMenuBar` class. It adds the registered menu `ActionGroup` objects to the browser menu bar. Each `ActionGroup` adds a new menu to the menu bar in the order it was registered through the OpenTools API. Most of the application native menus are registered at startup before the **OpenTools discovery** process begins. All should be registered before the “UI” OpenTools category is loaded.

You must add or remove a menu group within the OpenTool’s `initOpenTool()` method only. The following example adds a new menu called “MyClosers” to the browser’s menu bar. First it adds the desired close actions to the new menu group, then the new group is added to the browser menu bar with the call to `Browser.addMenuGroup()`:

```
public static void initOpenTool(byte majorVersion, byte minorVersion) {
  if (majorVersion == PrimeTime.CURRENT_MAJOR_VERSION) {
    GROUP_MyClosersGroup.add(ProjectView.ACTION_ProjectCloseActive);
    GROUP_MyClosersGroup.add(Browser.ACTION_NodeClose);
    GROUP_MyClosersGroup.add(Browser.ACTION_NodeCloseAll);
    GROUP_MyClosersGroup.add(Browser.ACTION_NodeRevert);

    GROUP_MyClosers.add(GROUP_MyClosersGroup);

    Browser.addMenuGroup(GROUP_MyClosers);
  }
}

public static ActionGroup GROUP_MyClosersGroup = new ActionGroup();
public static ActionGroup GROUP_MyClosers = new ActionGroup("MyClosers", 'm',
    true);
```

Here is the code that removes the native Edit menu of JBuilder:

```
Browser.removeMenuGroup(com.borland.jbuilder.JBuilderMenu.GROUP_Edit);
```

Adding and removing individual menu Actions or ActionGroups

Add a menu `Action` or `ActionGroup` only within your OpenTool's `initOpenTool()` method. This is the logic to add the Revert action to the New group of the native File menu of JBuilder:

```
JBuilderMenu.GROUP_FileNew.add(Browser.ACTION_NodeRevert);
```

This code removes the four New `Actions` from the native File menu of JBuilder:

```
JBuilderMenu.GROUP_File.remove(JBuilderMenu.GROUP_FileNew);
```

Adding and removing toolbar groups

The `BrowserToolBarPane` class is an extension of `ActionToolBarPane` class. It adds the registered toolbar `ActionGroup` objects. Each `ActionGroup` appends to the browser toolbar in the order it was registered through the OpenTools API. Each `ActionGroup` also appears in the toolbar popup menu to allow users to hide or reveal it just as they can the native toolbar groups. All the native toolbar groups of JBuilder are registered at application startup prior to the OpenTools discovery process.

The following example adds a new group to the browser's toolbar. First it adds the desired close actions to the new group, then the new group is added to the browser toolbar with the call to `Browser.addToolBarGroup()`:

```
public static void initOpenTool(byte majorVersion, byte minorVersion) {
  if (majorVersion == PrimeTime.CURRENT_MAJOR_VERSION) {
    GROUP_MyClosersGroup.add(ProjectView.ACTION_ProjectCloseActive);
    GROUP_MyClosersGroup.add(Browser.ACTION_NodeClose);
    GROUP_MyClosersGroup.add(Browser.ACTION_NodeCloseAll);
    GROUP_MyClosersGroup.add(Browser.ACTION_NodeRevert);

    GROUP_MyClosers.add(GROUP_MyClosersGroup);

    Browser.addToolBarGroup(GROUP_MyClosers);
  }
}

public static ActionGroup GROUP_MyClosersGroup = new ActionGroup();
public static ActionGroup GROUP_MyClosers = new ActionGroup("MyClosers", 'm', true);
```

This is the logic to remove the File group from the native JBuilder toolbar:

```
Browser.removeToolBarGroup(com.borland.jbuilder.JBuilderToolBar.GROUP_FileBar);
```

Hiding and revealing toolbar groups

This code removes the File toolbar group from all `Browser` instances:

```
Browser[] browsers = Browser.getBrowsers();
for (int i = 0; i < browsers.length; i++) {
  browsers[i].getToolBarPane().setToolBarVisible(JBuilderToolBar.GROUP_FileBar,
      false);
}
```

Adding and removing specific toolbar buttons

This logic adds a Revert button to the native JBuilder toolbar Edit group:

```
JBuilderToolBar.GROUP_EditBar.add(Browser.ACTION_NodeRevert);
```

This code removes the Close button from the native JBuilder toolbar:

```
JBuilderToolBar.GROUP_FileBar.remove(Browser.ACTION_NodeClose);
```

Delegated actions

`Browser` defines `DelegateAction` objects that are used to re-direct an action request to the UI component that has the current focus based on a timer event which occurs twice a second. By default, the action is disabled. To enable one of these actions, your class (typically this done by a `NodeViewer` instance) must implement the `DelegateHandler` interface and override its single `getAction()` method.

For example, this `getAction()` method handles the `DELEGATE_Copy` action:

```
public Action getAction(DelegateAction delegate) {
  if (delegate == com.borland.primetime.ide.Browser.DELEGATE_Copy) {
    return ACTION_CopyContent;
  }
  return null;
}

private static ClipboardOwner clipWatcher = new ClipboardObserver();
protected static class ClipboardObserver implements ClipboardOwner {
  public void lostOwnership(Clipboard clipboard, Transferable contents) {}
}

public UpdateAction ACTION_CopyContent =
  new UpdateAction("Copy", 'C', "Copy selected content") {
  public void update(Object source) {
    setEnabled(isSelection());
  }
  public void actionPerformed(ActionEvent e) {
    String str = getSelection();
    if (str != null) {
      Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
      clipboard.setContents(new StringSelection(str), clipWatcher);
    }
  }
};
```

Listening for Browser events

The `Browser` class maintains two listener lists. One list type is for listeners of events from any instance of `Browser` (methods `addStaticBrowserListener()` and `addStaticProjectGroupBrowserListener()`) while the other list type is for listeners of events from a particular `Browser` instance (methods `addBrowserListener()` and `addProjectGroupBrowserListener()`). Using `BrowserAdapter` provides a convenient way to listen for only those events you need.

For example,

```
Browser.addStaticBrowserListener(new BrowserAdapter() {
  public void browserNodeActivated(Browser browser, Node node) {
    System.out.println("Active node is " + node.getLongDisplayName());
  }
});
```

Content Manager concepts

The `ContentManager` class implements an IDE panel which hosts the presentation of open files for the currently active project manipulated using two sets of tabs. One set is used to select among the open `Node` instances, the other is to select which viewer to be

used to display or edit its content. Generally an open Node is an instance of FileNode or, less commonly, LightweightNode

There is one ContentView for each open Node. You can't replace the ContentView class with the OpenTools API. Each tab (located at either the top, right, or left depending on configuration) is for an instance of ContentView. That tab contains a single-click close icon, the Node's display name and optionally an icon based on the node type. It is possible using OpenTools to append your own menu options to the context popup that appears when right-clicking on this tab by registering a ContextActionProvider.

Each ContentView displays a set of tabs at the bottom of its pane. A tab is a viewer javax.swing.JComponent provided by an instance of NodeViewer. Each NodeViewer is created for that particular Node using an instance of NodeViewerFactory that registered with Browser at application startup.

Each NodeViewerFactory is associated with at least one particular Node class. There's no limit on the number of different factories that can be registered for the same Node class. If a file extension hasn't been registered, it's treated as if it were a TextFileNode.

Each NodeViewer is based upon either the AbstractNodeViewer or AbstractBufferNodeViewer class. Each NodeViewer when created is given an instance of Context to tell it which Browser it is associated and which Node it is providing a view. Using Browser methods, it is possible to obtain references to all open nodes and all viewers of any open Node.

The order of the ContentView tabs is largely determined by the order in which the NodeViewerFactory instances were loaded during OpenTools discovery since that determines when they can register themselves. There is an option during registration which can cause a NodeViewerFactory to be added to the front of the list, but that is not recommended. A factory can optionally implement NodeViewerFactoryFilter which permits it to re-order the factories which report they can provide viewers for that node prior to the UI being created.

Browser relationship

There is only one ContentManager instance for each Browser. You can access its functionality indirectly only through Browser methods. You can't replace the ContentManager class with the OpenTools API.

Browser methods provide access to the ContentManager and its component classes. For convenience, some of those methods are also available from ProjectView. The most commonly used Browser methods are getActiveNode() and setActiveNode(). The active Node is the file that has the currently select ContentView. If the given Node isn't currently open, calling setActiveNode() opens it.

The Browser also provides some BrowserListener events that are tied to NodeViewer activation and deactivation. If you are writing a NodeViewer, however, you generally should override the similar NodeViewer methods instead of using these notifications. (As an example of the problems you can run into, the browserActivated event only goes to the active NodeViewer which may have changed by a Wizard while the Browser is "deactivated" due to a modal dialog getting the focus. To ensure a NodeViewer is notified of the Browser state, browserActivated is always invoked prior to viewerActivated.)

When the project or application closes, Browser causes the open files to be recorded so that they can be automatically re-opened. For performance reasons, the viewerActivated method of each NodeViewer will only be called when actually activated. A NodeViewer should be designed for minimal initialization when its viewer component is requested.

NodeViewerFactory narrative

Here is an example of how this works.

`TextNodeViewerFactory` is a registered `NodeViewerFactory`. It creates a `TextNodeViewer` containing the application text editor for any `Node` which inherits from `TextFileNode`. (Any file type which is not registered will be treated as if it had a text file extension.)

When a `Node` is to be opened, all the registered factories are asked if they can provide a `NodeViewer` for it. The `TextNodeViewerFactory` replies it can when that `Node` is an instance of `TextFileNode`. Ultimately it will be asked to create a viewer. This factory creates an instance of `TextNodeViewer` which provides “Source” as the name on the tab.

If other registered factories respond, then additional tabs are created for those viewers. For instance, the “Design” tab is added if the `Node` is determined to be designable by the registered factory for the UI Designer.

Registering a new file type

You can register a new file type in two ways. The easiest way is to find an existing registered file type and make an association between it and your file extension so that they act identically. You can do this in JBuilder using the Tools|Preferences dialog box and using the Browser|File Types page.

You can accomplish the same thing with code in the `initOpenTool()` method that looks like this:

```
public static void initOpenTool(byte majorVersion, byte minorVersion) {
  if (majorVersion == PrimeTime.CURRENT_MAJOR_VERSION) {
    FileType ft = FileType.getFileType("mytxt");
    if (ft == null) {
      FileType.registerFileType("mytxt", FileType.getFileType("txt"));
    }
  }
}
```

If you do not want to associate your new type with an existing one, you must register your own `Node` in the `initOpenTool()` method. Here’s an example:

```
public static void initOpenTool(byte majorVersion, byte minorVersion) {
  if (majorVersion == PrimeTime.CURRENT_MAJOR_VERSION) {
    Icon ICON = BrowserIcons.ICON_FILEIMAGE;
    registerFileNodeClass("mine", "My source file", MyFileNode.class, ICON);
  }
}
```

Registering NodeViewerFactory OpenTools

Your `NodeViewerFactory` must provide the standard interface looked for by OpenTools discovery and register itself when loaded using one of the two `Browser` static `registerNodeViewerFactory()` methods. One of these methods has an `asFirst` boolean parameter that, if true, means that the `NodeViewerFactory` is requesting that it appear as the first `NodeViewer` for the file types it supports in the `ContentView`. If more than one `NodeViewerFactory` for the same file type makes the same request as all the OpenTools are discovered, only one can be honored, of course, so making the request doesn’t guarantee your `NodeViewerFactory` will have priority. Generally you should not take advantage of this option.

This code example registers a new `NodeViewerFactory` with `Browser`:

```
public static void initOpenTool(byte majorVersion, byte minorVersion) {
  if (majorVersion == PrimeTime.CURRENT_MAJOR_VERSION) {
    Browser.registerNodeViewerFactory(new MyNodeViewerFactory());
  }
}
```

If you specify the **-verbose** command-line option, the console display reports the registration of all `NodeViewerFactories` (along with all the other OpenTools being discovered and registered).

Implementing for FileNode and TextFileNode

If your file type isn't text or you choose to provide your own editor, you should extend `FileNode`. Your implementation would look similar to the following:

```
public class MyFileNode extends FileNode {
  public static final Icon ICON = BrowserIcons.ICON_FILEIMAGE;

  public MyFileNode(Project project, Node parent, Url url) throws
      DuplicateNodeException {
    super(project, parent, url);
  }

  public javax.swing.Icon getDisplayIcon() {
    return ICON;
  }
}
```

If the file contains text, you could base your implementation on `TextFileNode`. In that case, your implementation might be something like this:

```
public class MyFileNode extends TextFileNode {

  public MyFileNode(Project project, Node parent, Url url) throws
      DuplicateNodeException {
    super(project, parent, url);
  }

  public Class getEditorKitClass() {
    return MyEditorKit.class;
  }

  public Class getTextStructureClass() {
    return MyTextStructure.class;
  }
}
```

In this example, you would have to register the `MyEditorKit` class using the Open Tools API. See the Editor documentation for more details. (There is an existing `TextEditorKit`.)

Implementing NodeViewerFactory

`NodeViewerFactory` is a simple interface with only two methods to implement: `canDisplayNode()` and `createNodeViewer()`. When a file is opened, each registered

factory is checked to see if it can create a NodeViewer for that file type. Here's an implementation of a NodeViewerFactory:

```
public class MyNodeViewerFactory implements NodeViewerFactory {

  public boolean canDisplayNode(Node node) {
    return (node instanceof MyFileNode);
  }

  public NodeViewer createNodeViewer(Context context) {
    if (canDisplayNode(context.getNode())) {
      return new MyNodeViewer(context);
    }
    return null;
  }
}
```

Implementing NodeViewer

There are commonly two different types of NodeViewer: those that view a FileNode and those that view a TextFileNode. The TextFileNode viewer can use existing logic of the editor.

FileNode-based NodeViewer

Because of the AbstractNodeViewer class, implementing a FileNode-based NodeViewer is fairly easy. All you must do is provide a JComponent for the file content and an optional JComponent for the file structure to appear in the structure pane. You supply the name that appears on the ContentView tab, and the text that becomes the tab's tooltip. Here's an example:

```
public class MyNodeViewer extends AbstractNodeViewer {

  public MyNodeViewer(Context context) {
    super(context);
  }

  public String getViewerTitle() {
    return "MyView";
  }

  public String getViewerDescription() {
    return "My cool viewer";
  }

  public JComponent createViewerComponent() {
    return new MyViewerComponent(context);
  }

  public JComponent createStructureComponent() {
    return null;
  }
}
```

Another way to implement such a viewer is based on AbstractBufferNodeViewer. If your viewer needs to read and write its content through the Virtual File System and co-exist with other viewers registered for your file type, then this class greatly simplifies your work. It does this by not notifying you that the file buffer has changed unless your viewer is active, then using that saved state when the viewer is activated to request that the viewer reload using the latest version.

TextFileNode-based NodeViewer

When implementing a `TextFileNode`, you don't need to provide a `NodeViewer`. Instead you provide your customized implementations of the supporting classes used by the native editor `NodeViewer`. These supporting classes include `TextEditorKit`, and `AbstractScanner`. If you don't override any of these, you end up with the same `NodeViewer` as provided by default for a `TextFileNode`.

See the Editor documentation for more details.

Adding an Action to the context menu

You can append your own `UpdateAction` or `ActionGroup` to the `ContentManager` pop-up menu that appears when you right-click on the tab for an open `Node`. The following code demonstrates adding a `WizardAction`. Note that such wizards must be registered using the OpenTools API.

```
    public static void initOpenTool(byte majorVersion, byte minorVersion) {
      if (majorVersion == PrimeTime.CURRENT_MAJOR_VERSION) {
        ContentManager.registerContextActionProvider(new ContextActionProvider()
{
          public Action getContextAction(Browser browser, Node[] nodes) {
            if (browser.getActiveNode() != null) {
              return WIZARD_MyWizard;
            }
            return null;
          }
        });
      }
    }

    public static final WizardAction WIZARD_MyWizard = new WizardAction (
      "My wizard...", 'w', "My ContentManager wizard",
      BrowserIcons.ICON_BLANK,
      BrowserIcons.ICON_BLANK,
      false) {

      protected Wizard createWizard() {
        return new MyWizard();
      }
    };
```

For more information about writing and registering wizards with the OpenTools API, see Chapter 10, "Wizard concepts."

Project View concepts

The `ProjectView` is an IDE panel which can display `ProjectViewPage` instances created as needed using registered `ProjectViewPageFactory` instances. Information from the factory is used to automatically add an entry to the View|Panes menu for additional UI enabling hide/reveal of each page as a `View`. That state is part of the information recorded within a `Workspace`.

In general, each page should be project-related and is automatically refreshed as the active project is changed. Any number of projects can be open, but there can be only one active project at a time.

The "Project" page contains a toolbar and a tree which is a hierarchical display of the nodes that make up the currently active project plus those of any sibling projects if a member of a project group. There is one such page for each open project.

Each tree entry is one of the following node types or a subclass:

- `ProjectGroup`
- `Project`
- `FileNode`
- `PackageNode`
- `FolderNode`
- `LightweightNode`

The “Files” page contains a toolbar and tree which shows directories based on project context and additional directories which can be configured as “favorites”. These configurable directories are shared with the `UrlChooser` dialog which is used frequently in the product as a file or directory chooser. There is one such page for each open project.

The project pane context menu is configurable. You can register a `ContextActionProvider` which may be used to add an `ActionGroup` or `UpdateAction` to the context menu before it is displayed. However if the UI element selected is not a `Node` instance, the default behavior will be minimal.

Drag & drop behavior within the project pane is largely delegated to the `Node` being dragged, those being dragged over, and the `Node` which is dropped upon.

The project pane is both a `ProjectGroupListener` and `ProjectListener`. Changes to parent-child relationships within project groups and projects will cause an automatic refresh of the project pane. The `refreshTree()` method provides a way to force a complete refresh when needed. Individual pages can optionally be a `VFSListener` in order to update its display based on files being added, deleted, or renamed.

Just as the content pane has its “active node”, the project pane has its “selected nodes”. The selected nodes are highlighted and there are accessor methods to fetch them programmatically.

If no user project is open, then a default project which allows very limited functionality is active. If the default project is active, most wizards either disable themselves, or else they automatically invoke the New Project wizard to create a project before executing.

In general you should assume that `ProjectView` methods need to be invoked from the Swing event dispatching thread. Calling from the wrong execution thread can easily result in erratic behavior including causing the IDE to lock up just as it can in any other Swing-based application.

You can’t replace the `ProjectView` using the OpenTools API. Each `Browser` can have just one instance of `ProjectView`

Getting a ProjectView reference

Obtain a `ProjectView` reference from an instance of `Browser`. To be sure of getting the correct instance, you should attempt to use a `Browser` reference within your context. If one isn’t available, it’s usually safe to assume the currently active `Browser` if you have been launched from a menu or popup context menu:

```
ProjectView pv = Browser.getActiveBrowser().getProjectView();
```

Hiding and revealing the ProjectView

You can hide or reveal a `ProjectView` through its `Browser`:

```
Browser.getActiveBrowser().setProjectPaneVisible(true);
```

Opening and activating a project

Use `setActiveProject()` to open a project (if it isn't already) and make it the active project. The `getOpenProjects()` method of `ProjectView` returns a list of all open projects.

Adding a Node to the project

`ProjectView` has listeners that will detect a parent change and update the tree it displays without any additional help. The following examples use the active project for the parent, but you can use a `FolderNode` or any other implementor of `NodeContainer` in the project as the parent.

This example adds a `JavaFileNode` given only an absolute path and automatically causes the `ProjectView` true to be refreshed to show the new project child `Node`. This relationship will be recorded in the project file when the change is saved.

```
String path = "c:/myfile.java";
Project project = Browser.getActiveBrowser().getActiveUserProject();
if (project != null) {
  Url url = new Url(new File(path));
  Node node = project.getNode(url);
  node.setParent(project);
}
```

The following code demonstrates adding a JBuilder `PackageNode` to the project given only an absolute path and the root directory from the project source path.

```
Project project = Browser.getActiveBrowser().getActiveUserProject();

String projectSourcePath = "c:/myproject/src";
String packagePath = "c:/myproject/src/com/mypackage";
String name = packagePath.substring(projectSourcePath.length() + 1);
name = name.replace(File.separatorChar, '.');
// name is now in the format "com.mypackage"
if (project.findNodes(name).length == 0) {
  new PackageNode(project, project, name);
}
```

Removing a node from the project

To remove a node from the project that has a parent, first ensure it is closed, and then reset its parent reference to `null`. Here's an example:

```
Browser browser = Browser.getActiveBrowser();
Node node = browser.getProjectView().getSelectedNode();
if (node != null) {
  browser.closeNode(node);
  node.setParent(null);
}
```

Nodes which have been added to the `ProjectView` by the Automatic Source Package Discovery feature normally have a `null` parent reference. The `PackageNode` reports them as its children, however they do not have a reference to the `PackageNode`. This allows the `Node` to also appear as a child of the project or perhaps a `FolderNode`. It's not possible to remove a `PackageNode` which is added by Automatic Source Package Discovery, other than by disabling that feature on the Project|Project Properties| General page.

Getting the selected nodes from the ProjectView

The `ProjectView` tree allows the user to select multiple nodes. You can obtain an array that contains those selected nodes:

```
ProjectView pv = Browser.getActiveBrowser().getProjectView();
Node[] nodes = pv.getSelectedNodes();
for (int j = 0; j < nodes.length; j++) {
  System.out.println(nodes[j].getLongDisplayName());
}
```

Adding an action to the context menu

You can append your own actions to the `ProjectView` right-click pop-up menu. The following code demonstrates adding a `WizardAction`. Note that such wizards must be registered using the OpenTools API.

```
public static void initOpenTool(byte majorVersion, byte minorVersion) {
  if (majorVersion == PrimeTime.CURRENT_MAJOR_VERSION) {
    ProjectView.registerContextActionProvider(new ContextActionProvider() {
      public Action getContextAction(Browser browser, Node[] nodes) {
        if (browser.getActiveUserProject() != null) {
          return WIZARD_MyWizard;
        }
        return null;
      }
    });
  }
}

public static final WizardAction WIZARD_MyWizard = new WizardAction (
  "My wizard...", 'w', "My ProjectView wizard",
  BrowserIcons.ICON_BLANK,
  BrowserIcons.ICON_BLANK,
    false) {
  protected Wizard createWizard() {
    return new MyWizard();
  }
};
```

For more information about writing and registering wizards with the OpenTools API, see Chapter 10, “Wizard concepts.”

Ensuring a usable project is active

You can’t add files to the default project. Therefore, if you are writing a wizard that generates files and the default project is active, your wizard won’t be able to function correctly. You can solve this problem in a couple of ways. Your `BasicWizard` could either disable itself, or it could present the New Project wizard and let the user decide whether to create a new project.

The following code demonstrates how to handle the second situation for JBuilder. Your wizard remains enabled, but it invokes the Project wizard to create a new JBuilder project. If the user fails to create the project, the wizard terminates by returning `null`:

```
public WizardPage invokeWizard(WizardHost host) {
  setWizardTitle("My Wizard");

  Project project = host.getBrowser().getActiveUserProject();
```

```
    if (!(project instanceof JBProject)) {
      // If not a usable JBuilder project, launch Project Wizard
      WizardDialog dialog = new WizardDialog(host.getBrowser(),
        com.borland.jbuilder.wizard.newproject.ProjectWizard());
      dialog.show();
      project = host.getBrowser().getActiveUserProject();
    }

    // If not an active project we can use, abort
    if (!(project instanceof JBProject)) {
      return null;
    }

    step1 = new MyWizardPage1();
    addWizardPage(step1);

    return super.invokeWizard(host);
  }
```

Adding a new view

Adding a new view is relatively easy if you subclass `ProjectTree`. The code sample below shows how that could be done and points out a few methods where you might introduce behavior to customize it. In particular, `createRootNode()` provides a way to alter the tree root node to be something other than a Project or Project Group as the File Browser page does. Of course, your implementation does not have to be tree-based.

Those views which are visible are recorded in a workspace file. When you register a new `ProjectViewPage`, the default behavior is that it will not be visible. Use View|Panes to make it visible.

```
public class MyProjectViewPage extends ProjectTree {
  public final static String MY_VIEW_ID = "my_project_view_page";

  public static void initOpenTool(byte major, byte minor) {
    if (major == PrimeTime.CURRENT_MAJOR_VERSION) {
      ProjectViewPageFactory factory = new ProjectViewPageFactory(
          MY_VIEW_ID,
          "MyView",
          "MyView content",
          BrowserIcons.ICON_BLANK) {

        public ProjectViewPage createPage(
            ProjectViewPageHost pvh,
            RootNode rootNode) {

          return (new MyProjectViewPage(pvh, rootNode, this));
        }
      };
      ProjectView.registerProjectViewPageFactory(factory);
    }
  }

  public MyProjectViewPage(
    final ProjectViewPageHost pvh,
    RootNode rootNode,
    ProjectViewPageFactory factory) {
```

```
    super(pvh, rootNode, factory);
  }

  public void activate() {
    super.activate();
    // add listeners
  }

  public void deactivate() {
    super.deactivate();
    // remove listeners
  }

  protected ProjectTreeNode createRootNode() {
    return super.createRootNode();
  }

  public void releaseProject() {
    super.releaseProject();
    // free all resources, project closed
  }
}
```

Structure View concepts

The `StructureView` is an IDE panel that acts as a container for a `JComponent`, optionally provided by the currently active `NodeViewer`. The `StructureView` panel is referred to in JBuilder as the structure pane. By default, it appears in the lower left of the IDE underneath the project pane. Usually it uses a tree hierarchy to display structure and provide feedback on syntax errors.

You can hide, but not replace, the `StructureView` pane using the OpenTools API.

Registering a component for the StructureView

When you are defining a file type using the OpenTools API, the object you register with `FileNode` should extend `TextFileNode` if the file contains text and you want to customize the existing editor. You do this indirectly by overriding the `getTextStructureClass()` method of `TextFileNode`. For example,

```
public Class getTextStructureClass() {
  return MyTextStructure.class;
}
```

The `getTextStructureClass()` method returns your class which extends `TextStructure` in order to override its `setTree()` method. The `JTree` parameter of `setTree()` is the `JComponent` that is given to the `StructureView`.

If you do not extend `TextFileNode`, then the `createStructureComponent()` method of its `NodeViewer` supplies the Swing-based component for the `StructureView`. This would be done by something like the following:

```
public JComponent createStructureComponent() {
  return (new MyStructure(context));
}
```

For more information about defining your own file types, see “Implementing for FileNode and TextFileNode” on page 29.

Writing a component for the StructureView

Usually files that aren't text do not provide a structure view so there are no helper classes provided by the OpenTools API at this time. If you plan to use a sorted `JTree` to provide the structure view, however, you might find this section interesting.

For a `TextFileNode`, the component provided for the structure view is a `JTree` that is sorted and with the most commonly needed nodes automatically expanded. The key methods are `setTree()` and `updateStructure()`.

The `updateStructure()` method is called automatically whenever the source file is modified. Usually the nodes it adds to the tree contain references to positions in the file that are used by the `nodeSelected()` method, which scrolls to and points at a line in the source file, and the `nodeActivated()` method, which moves the focus to the source file line.

Example: simple implementation

The following is a skeleton of a simple implementation. The `mergeChildren()` method is a helper which tries not to collapse any expanded tree nodes as the tree is being updated.

```
public class MyTextStructure extends TextStructure {

  public MyTextStructure() {
    treeModel.setRoot(new MyStructureNode(null));
  }

  class MyStructureNode extends MergeTreeNode {
    public MyStructureNode(Object userObject) {
      super(userObject);
    }

    public void sortChildren() {
      MergeTreeNode[] array = getChildrenArray();
      if (array == null)
        return;
      Arrays.sort(array, new Comparator() {
        public int compare(Object o1, Object o2) {
          // Do comparison here between two tree nodes
          return 0;
        }
      });
      children = new Vector(Arrays.asList(array));
      sortDescendants();
    }

    public void sortDescendants() {
      if (children != null) {
        Enumeration e = children.elements();
        while (e.hasMoreElements()) {
          ((MyStructureNode)e.nextElement()).sortChildren();
        }
      }
    }
  }

  private void showProperties() {
    // Show properties here
```

```
  // Re-sort everything but the top level of the structure tree
  MyStructureNode root = (MyStructureNode)treeModel.getRoot();
  root.sortChildren();
  treeModel.nodeStructureChanged(root);
}

public void setTree(JTree tree) {
  super.setTree(tree);

  TreeNode root = (TreeNode)treeModel.getRoot();
  int count = root.getChildCount();
  for (int index = 0; index < count; index++) {
    TreeNode node = root.getChildAt(index);
    tree.expandPath(new TreePath(new Object[] {root, node}));
  }
}

public JPopupMenu getPopup() {
  JPopupMenu menu = new JPopupMenu();
  JMenuItem props = new JMenuItem("Properties...");
  props.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      showProperties();
    }
  });
  menu.add(props);
  return menu;
}

public void nodeSelected(DefaultMutableTreeNode node) {
  // handle selection of a tree node here
}

public void nodeActivated(DefaultMutableTreeNode node) {
  // handle activation of a tree node here
}

public Icon getStructureIcon(Object value) {
  // return Icon appropriate for Object instance
  return null;
}

public void updateStructure(Document doc) {

  final MyStructureNode newRoot = new MyStructureNode(null);

  try {

    // Build a new structure tree using newRoot here

    // Prepare an object that updates the model
    Runnable update =
      new Runnable() {
        public void run() {

          MyStructureNode root = (MyStructureNode)treeModel.getRoot();

          // Merge the new model into the old so that expansion paths can be
          // preserved
          root.mergeChildren(newRoot);
```

```
            // Sort everything including the top level of the structure tree
            root.sortChildren();

            // Update the display
            treeModel.nodeStructureChanged(root);
          }
        };

      // Update the model on the main swing thread...
      if (SwingUtilities.isEventDispatchThread())
        update.run();
      else
        SwingUtilities.invokeLater(update);

    }
    catch (java.io.IOException ex) {
    }
  }
}
```

Status View concepts

The `StatusView` is the status bar panel at the bottom of the IDE used to display a single-line text message. You can specify a predefined message type which controls the icon displayed and text color. Messages are maintained by type so that they either expire after a time interval or the arrival of another status message.

The `StatusView` supports transient messages, such as descriptive text for `ActionButton` and `ActionMenuItem`. For example, when the user moves the mouse pointer over a menu item, descriptive text about the menu item appears in the `StatusView`. The descriptive text disappears as the mouse pointer moves away from the menu item. To provide a way to display this temporary text, `StatusView` maintains a second internal string that holds this hint text. This second string is substituted for the usual text when it is needed.

You can't replace the `StatusView` with the OpenTools API. Access the `StatusView` only through the `Browser`.

Using the StatusView

There are three categories of messages supported: normal, warning, and error. There will always be a beep and a visual flash for warning and error messages. Each has a default color and icon associated with it when displayed. A different color can be optionally specified for a message.

Messages are displayed only for a configured interval before being removed. A different interval can be optionally specified for a message. You can write a message from a worker thread if desired. If there are multiple messages active, the text will alternate in one-second intervals.

```
StatusView sv = Browser.getActiveBrowser().getStatusView();
try {
  doSomething();
  sv.setText("Did something successfully");
}
catch (Exception ex) {
  sv.setText("Did something badly", StatusView.TYPE_ERROR);
}
```

Handling the hint text

The menuing subsystem automatically handles the hint text for the IDE menus and toolbar, but you might like to understand how this feature works. The hint text is set when the mouse pointer enters the menu item or the toolbar button, and then is reset when the mouse pointer moves away. This provides users additional information about that action before they actually use it. The code might look like this:

```
jMenuItem1.addMouseListener(new MouseAdapter() {
  public void mouseEntered(MouseEvent e) {
    Browser.getActiveBrowser().getStatusView().setHintText("I am a menu item");
  }
  public void mouseExited(MouseEvent e) {
    Browser.getActiveBrowser().getStatusView().setHintText(null);
  }
});
```

Message View concepts

The `MessageView` is an IDE panel that generally appears only as it is needed but may be hidden, revealed, and undocked through the user interface. Because there are multiple subsystems within the IDE that use it, a tabbed interface separates the output from each subsystem. There is one `MessageView` for each `Browser` instance. You can't replace a `MessageView` using the OpenTools API.

The `MessageCategory` class defines a `MessageView` tab. When we speak of a tab in the message pane, we are referring to the tab *and* the page it displays, as a unit. A `MessageCategory` routes `Message` objects to the correct tab as messages are added to the message pane. Each tab has a single-click Close icon. Using the `categoryWillPromptOnClose()` method, this icon will appear in either its "solid" (no prompt on close) or "broken" form. This method will be called based on a timer so it can be used to signal the completion of its output.

By default the tab is the parent of a `JComponent` that displays a tree, but you can provide a custom component. For example, the run/debug console is a custom component.

You can register an `UpdateAction` or `ActionGroup` with `MessageView` to customize the right-click context menu on the tab for a `MessageCategory`.

Each `Message` object appears using its defined text, background color, foreground color, font, icon, popup context action, and tooltip attributes. To add a message to the `MessageView`, call one of the several variants of the `addMessage()` method, specifying the appropriate `MessageCategory`.

Despite usually having the appearance of being a text control, the default `MessageView` tab actually contains a `javax.swing.JTree` allowing you to optionally generate a hierarchical view by specifying a parent `Message`. (This also means that you should not send extremely long blocks of text with embedded carriage control or tab characters, since they will not display as they would in a text control.)

The `Message` class accepts event notifications. A single-click on a message in the message pane calls the `selectAction()` method. A double-click calls the `messageAction()` method. If the user presses *F1* while a message is selected in the message pane, the `helpAction()` method is called. Implementing any of these events is optional.

Accessing the MessageView

To get access to the `MessageView` for a particular `Browser` window, call its `getMessageView()` method. The `Browser` also provides the `isMessagePaneVisible()` and `setMessagePaneVisible()` methods.

Creating a MessageView tab

A `MessageView` tab is defined from a `MessageCategory` object. You can choose from several possible constructors. The simplest one requires just the initial text for the tab, such as this example:

```
final static MessageCategory MY_MESSAGES = new MessageCategory("MyMessages");
```

A `MessageCategory` is always passed as a parameter in either one of the many variants of the `addMessage()` method or in the `addCustomTab()` method. If the passed `MessageCategory` describes a tab that doesn't yet exist, then a new tab is created before the message is added to it. For example, this code adds the new tab labeled "MyMessages" to the `MessageView` and adds the message that displays "This is a test.":

```
final static MessageCategory MY_MESSAGES = new MessageCategory("MyMessages");
Browser.getMessageView().addMessage(MY_MESSAGES, new Message("This is a
    test."));
```

If you have created a custom component to display the tab content, that component supplies all the UI inside that tab. Therefore, calling the `MessageView addMessage()` method would result in an exception. To create a custom tab, call the `MessageView addCustomTab()` method, passing to it the `MessageCategory` and the custom component. For example,

```
JComponent c = new MyCustomPanel();
Browser.getMessageView().addCustomTab(MY_MESSAGES, c);
```

Using a MessageView tab

Unless you're using a custom tab component, content consists of a `JTree` inside a `JScrollPane`. You can create a message hierarchy with code like this:

```
MessageView mv = Browser.getActiveBrowser().getMessageView();
Message msg = mv.addMessage(MY_MESSAGES, new Message("This is test."));
mv.addMessage(MY_MESSAGES, msg, new Message("This is a child of test."));
```

If messages are added to the `MessageView` with `addMessage()` calls that don't supply a parent `Message`, the tree appears flat like a text area.

`MessageView` provides several `UpdateAction` and `BrowserAction` objects that are tied to the user interface. Some apply against a particular `Message` object (expand, next, prior), a particular tab (clear, copy content, remove), and against all tabs (hide, remove all).

`ACTION_CopyContent` attempts to work against tab content in a way that might also work with a custom component. It assumes that within the component hierarchy there exists either a `JTree` or a `JTextArea`. When it is invoked with a keystroke or a click on a context menu, it tries to identify the key component and copies the selected portion (or if none is selected, then the entire content) to the clipboard.

`MessageView` calls the `MessageCategory` methods `categoryActivated()`, `categoryDeactivated()`, and `categoryClosing()` at the appropriate times. You might find them useful when you are implementing a custom tab. You can also use a `PropertyChangeListener` class for notification if the icon, tooltip, or title of the `MessageCategory` changes.

Customizing messages

Each `Message` object represents a single message line in the `MessageView`. Its text is displayed using any font, background color, and/or foreground color settings it defines.

If you are using a message hierarchy, you might want to take advantage of the `setLazyFetchChildren()` and `fetchChildren()` methods. They allow you to create a parent `Message` yet not supply its children until later, when the user decides to expand it. Here's an example that shows you how to do it:

```
public class ParentMessage extends Message {
  public ParentMessage() {
    setLazyFetchChildren(true);
  }
  public void fetchChildren(Browser browser) {
    MessageView mv = browser.getMessageView();
    mv.addMessage(MY_MESSAGES, this, new Message("This is my child."));
  }
}
```

If the user selects a `Message` by using the keyboard arrow keys to navigate through the message tree or by single-clicking the message, then either the `selectAction()` method or an `Action` specified by `setSelectAction()` method is called. If this is the type of interface you want, you should highlight the UI element in the `NodeViewer` that is associated with that `Message` (and also cause it to become the active viewer if not already).

If the user triggers a `Message` by using the *Enter* key or double-clicking, then either `messageAction()` or an `Action` specified by `setMessageAction()` is called. If this is the type of interface you want, you should move the focus to the UI element in the `NodeViewer` that is associated with that `Message`.

Example: using the message view

This example assumes that when the `Message` was created, it was associated with a known `TextFileNode` and a particular location in that file. (Refer to `LineMark` and `EditorPane` for more details about how this works.)

```
TextFileNode fileNode;
int line;
int column;
static final LineMark MARK = new HighlightMark();

public void selectAction(Browser browser) {
  displayResult(browser, false);
}

public void messageAction(Browser browser) {
  displayResult(browser, true);
}

private void displayResult(Browser browser, boolean requestFocus) {
  try {
    if (requestFocus || browser.isOpenNode(fileNode)) {
      browser.setActiveNode(fileNode, requestFocus);
      TextNodeViewer viewer = (TextNodeViewer)browser.getViewerOfType(fileNode,
          TextNodeViewer.class);
      browser.setActiveViewer(fileNode, viewer, requestFocus);
      EditorPane editor = viewer.getEditor();
      editor.gotoPosition(line, column, false, EditorPane.CENTER_IF_NEAR_EDGE);
```

```
        if (requestFocus) {
          editor.requestFocus();
        }
        else {
          editor.setTemporaryMark(line, MARK);
        }
      }
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }

  public static class HighlightMark extends LineMark {
    static Style highlightStyle;
    static {
      StyleContext context = EditorManager.getStyleContext();
      highlightStyle = context.getStyle(BasicStyleMap.LINE_HIGHLIGHT_KEY);
    }

    public HighlightMark() {
      super(highlightStyle);
    }
  }
```

If the user presses *F1* when the `Message` has the focus, its `helpAction()` method is called if no `Action` was supplied by `setHelpAction()`.

Each `Message` can supply an `UpdateAction` or `ActionGroup` through the `getContextAction()` method, so that a pop-up menu appears if the user right-clicks it.

Tip If you want a `Message` to appear as something other than a text string and are not using a custom component, specify a `TreeCellRenderer` using the `setCellRenderer()` method.

Chapter

5

VFS concepts

The Virtual File System provides seamless access to a number of different storage techniques, each defined by an implementation of the `Filesystem` interface. The contents of entries in ZIP/JAR files, physical files, and New “untitled” documents are provided as three `Filesystem` implementations in the core VFS implementation.

Features and subsystems

Much as `java.io.File` instances are used in Java to represent filenames, logical files in the VFS are identified using instances of a special `Url` class, one that’s different from the `java.net.URL` class. All virtual file manipulation is done through static methods on the `VFS` class and passing one or more `Url` instances as parameters.

The VFS also supports an in-memory representation of a file as an instance of the `Buffer` class. Buffers can be hold changes before being discarded or written to disk, and while a buffer exists all VFS operations treat the contents of the buffer as the “current” state of the virtual file.

The VFS also interacts with an internal disk caching mechanism for improved performance.

The VFS provides notifications of files being deleted, created and renamed to listeners via the `VFSListener` interface. Primetime UI components listen to VFS events, and may update the UI based on the type of VFS event.

For the above two reasons, it is important that OpenTools do not bypass the VFS, e.g., by creating/deleting files directly with the `java.io.*` package. Doing so risks making the caches invalid, and also does not notify `VFSListener`s about the creation and deletion of files.

However, it is sometimes unavoidable to bypass the VFS. For example, your OpenTool may run some external program that creates, deletes, and/or modifies files. In that case, you must notify the VFS that the files were created, deleted, and/or modified after the external program has been run.

Key VFS Classes

These are the classes you generally need to know:

- **com.borland.Primetime.vfs.VFS**

  This class is responsible for mapping an `Url` to a buffer representing its contents. The VFS serves as a global registry of `Buffer` instances, mapping `Url` descriptions to existing in-memory buffers when possible, and creating new buffers as necessary. All attempts to manipulate an `Url` through the VFS are directed to the filesystem associated with the `Url`. Certain VFS operations may use a cache. The usage or non-usage of the cache is transparent.

- **com.borland.primetime.vfs.Url**

  The `Url` class defines immutable resource references similar to the JDK URL class. Use the `VFS` class to manipulate `Url`s.

- **com.borland.Primetime.vfs.Buffer**

  A `Buffer` is an in-memory representation of the resource referred to by an `Url`, and as such may represent changes to the resource that have not yet been saved to permanent storage. A file open in the Primetime editor displays the contents of the `Buffer` for that file.

- **com.borland.Primetime.vfs.VFSListener**

  `VFSListener` is an interface for listening to VFS events, such as when a file is created or deleted. The `VFS` class has methods to add and remove a `VFSListener`.

- **com.borland.Primetime.vfs.Filesystem**

  The Filesystem interface defines a means of interacting with a particular persistent storage mechanism. The same interface is used for both read/write and read-only file systems, with exceptions thrown as appropriate to indicate situations when writing is not allowed. There are three implementations that ship with Primetime:

  - `NewFilesystem` — Used for providing an artificial Url for new files. Cannot be written to or read.
  - `FileFilesystem` — Use for accessing the contents of physical files. This is a read-write implementation.
  - `ZipFilesystem` — Used for accessing the contents of ZIP files. This is a read-only implementation.

- **com.borland.Primetime.vfs.VFSUtil**

  The `VFSUtil` class provides some useful utility methods for manipulating trees of `Url`s.

The FileFilesystem

There are various ways to create an `Url` associated with a physical file. For example, to create an `Url` for a file on Windows, `c:\abc\def.ghi`:

```
Url firstWay = new Url(new java.io.File("c:\\abc\\def.ghi"));
Url secondWay = new Url(FileFilesystem.PROTOCOL, "c:/abc/def.ghi");
Url thirdWay = new Url("file:///c:/abc/def.ghi");
```

For a Linux/Unix file, `/home/maria/def.ghi`:

```
import java.io.File;
import com.borland.Primetime.vfs.*;
...
Url firstWay = new Url(new java.io.File("/home/maria/def.ghi"));
Url secondWay = new Url(FileFilesystem.PROTOCOL, "/home/maria/def.ghi");
Url thirdWay = new Url("file:////home/maria/def.ghi");
...
```

Checking existence

Note that just like with a `java.io.File`, an instance of an `Url` does not mean the resource associated with the `Url` necessarily exists. To see if the associated resource exists, call `VFS.exists()`:

```
Url url = new Url(FileFilesystem.PROTOCOL, "c:/abc/def.ghi");
System.out.println(url.getName() + " exists: " + VFS.exists(url));
```

The ZipFilesystem

To create an `Url` in the `ZipFilesystem`, you need to enclose the filename with brackets:

```
Url zipUrl = new Url(ZipFilesystem.PROTOCOL, "[/JBuilderX/lib/Primetime.jar]");
```

Note that this is different than creating an `Url` for the same JAR in the `FileFilesystem`:

```
Url fileUrl = new Url(FileFilesystem.PROTOCOL, "/JBuilderX/lib/Primetime.jar");
```

With `zipUrl` you will be able to retrieve the children of the JAR, and with `fileUrl` you will not. `VFS.isDirectory(zipUrl)` will return `true`, while `VFS.isDirectory(fileUrl)` will return `false`.

The `ZipFilesystem` is read-only. Additionally, as of Primetime 4.7, the `ZipFilesystem` does not support nested ZIPs—it cannot read the children of a ZIP that is inside another ZIP.

Working with the NewFilesystem

The `NewFilesystem` provides an artificial `Url` source for new files. No actual information can be read from or written to the new filesystem. To read and write before a physical storage location has been decided, you need to get a `Buffer`, and read/write to that. When ready to save, you will need an `Url` from the `FileFilesystem`.

```
Url newUrl = new Url(NewFilesystem.PROTOCOL, "untitled1.txt");
VFS.getBuffer(newUrl).setContent("Hello".getBytes());
Url diskUrl = new Url(FileFilesystem.PROTOCOL, "c:/mydir/realfile.txt");
VFS.rename(newUrl, diskUrl); //After rename(), buffers and persistent
    state match.
VFS.getBuffer(diskUrl).save(); // Since NewFilesystem has no persistent state,
    save the buffer to persist to FileFilesystem.
```

The above code is a simplified example. If you know from the beginning that you want to create the file `c:/tmp/realfile.txt`, there is no reason to first create an `Url` in the `NewFilesystem`. However, if you don't initially know the name of the file that you will end up persisting, as it may be something you prompt the user for, or if you may end up not persisting at all, then it makes sense to create an `Url` in the `NewFilesystem` first.

New files are more typically generated using the `NewFilesystem` by requesting a node via `Project.createFileNode()` until the file is actually saved, at which time a real name should be chosen and the contents copied to a buffer representing real physical storage. The following code accomplishes the same as the above snippet using `Project` to create an untitled `Url`, and using `FileNode.saveAs()` to persist:

```
FileNode fileNode = project.createFileNode("txt"); //Creates a FileNode with a
    unique Url
System.out.println(fileNode.getUrl().getFile()); //Prints something like
    "1/Untitled1.txt"
fileNode.getBuffer().setContent("Hello".getBytes());
Url realUrl = new Url(FileFilesystem.PROTOCOL, "c:/mydir/realfile.txt");
fileNode.saveAs(realUrl);
```

VFS and Caching

Primetime has an internal file caching mechanism to improve performance for disk operations. Only certain directories may be cached, depending on the type of project

that is open. An OpenTool has no way of knowing whether a particular directory is cached not. VFS operations will transparently use the cache as appropriate.

The cache is kept up-to-date when all file manipulation goes through VFS. If files are manipulated outside VFS, then the cache will be invalid. Therefore, you should always use VFS to manipulate files, and the cache will always be correct. There are certain circumstances, however, where it is not possible to use VFS to delete or create file. For example, you might run some external program that generates files.

In such a case, you need to notify VFS about the changed files. If you know the individual files that were modified, created, and/or deleted, then there are three corresponding methods you can invoke:

- `VFS.fileCreated(Url createdUrl);`
- `VFS.fileDeleted(Url deleteUrl);`
- `VFS.fileModified(Url modifiedUrl);`

If you know that a directory or set of directories was modified, there are various methods to invoke:

- `VFS.checkCache(Url dirUrl); //Checks the cache, if there is one, for the specified directory only`
- `VFS.checkCaches(Url urlDir); //Recursively checks the caches, if there is one, for the specified directory`
- `VFS.checkCaches(); // Checks all caches.`

You should invoke the method that requires the least amount of checking possible. `VFS.checkCaches()`, without a parameter, is the most expensive operation, and should be avoided unless absolutely necessary.

VFSListener

The `VFSListener` is an interface for VFS events. The Primetime UI listens to VFS, and may update the UI based on VFS events. For example, if the Project Pane is displaying dynamically discovered content for a directory or a Java package, and some of the content is deleted, the UI in the Project Pane is updated to no longer display the deleted content.

This is an additional reason to invoke the same VFS methods discussed in “VFS and Caching” on page 47. If, for example, you delete a file without using VFS, the Project Pane UI will not correctly update, until a refresh is done.

VFSListener2

The `VFSListener2` extends the `VFSListener2` interface. This is a new interface as of Primetime 4.8 (in JBuilder 2005). A new interface was introduced instead of modifying the `VFSListener` interface in order to not break the OpenTools API. Use the `VFS.addVFSListener(VFSListener)` method to add both `VFSListener` and `VFSListener2` instances.

Getting the children of an Url

To get a directory, `c:\mydir`:

```
Url myDirUrl = new Url(FileFilesystem.PROTOCOL, "c:/mydir");
Url [] children = VFS.getChildren(myDirUrl, Filesystem.TYPE_BOTH);
```

To get a zip file, `/home/foo.zip`:

```
Url myDirUrl = new Url(ZipFilesystem.PROTOCOL, "[/home/foo.zip]");
Url [] children = VFS.getChildren(myDirUrl, Filesystem.TYPE_BOTH);
```

Working with directories

```
Url zipUrl = new Url(ZipFilesystem.PROTOCOL, "[/home/maria/foo.zip]");
Url dirUrl = new Url(FileFilesystem.PROTOCOL, "/home/maria/foo.contents");
Url copyDirUrl = new Url(FileFilesystem.PROTOCOL, "/home/maria/foo.again");
VFSUtil.copyTree(zipUrl, dirUrl); //Copy contents from the zip to disk
VFSUtil.copyTree(dirUrl, copyDirUrl); //Copy from disk to another place on disk
VFSUtil.removeTree(copyDirUrl); //Delete new directory recursively - use with
    caution!
```

Retrieving the contents of a file

```
Url url = new Url(FileFilesystem.PROTOCOL, "c:/mydir/foo.txt");
java.io.InputStream inputStream = VFS.getInputStream(url);
```

Note that if `c:/mydir/foo.txt` is open in the editor and has been modified, the `inputStream` will be a stream of the modified contents—of the `Buffer`. If `c:/mydir/foo.txt` is not modified, then the `inputStream` will be a stream of the file contents. The `VFS.getInputStream()` method handles that automatically. This makes it possible, for example, for a compiler to a compile something that has not been saved to disk—as long as the compiler reads from an `InputStream` that it retrieved via VFS.

Creating a new file on disk

This code will create a new `c:/mydir/newfile.txt` if it does not already exist, or overwrite it if it does already exist. If it does exist, and there is also a `Buffer` for it, then the `Buffer` will be updated.

```
Url url = new Url(FileFilesystem.PROTOCOL, "c:/mydir/newfile.txt");
java.io.OutputStream outputStream = VFS.getOutputStream(url);
outputStream.write("Hello".getBytes());
outputStream.close();
```

Relative Urls

The `Url` class has a method, `getRelativeUrl()`, which makes it easy to instantiate new `Url`s relative to an existing `Url`:

```
Url dirUrl = new Url(FileFilesystem.PROTOCOL, "c:/mydir/subdir/anothersubdir");
...
Url hardWay = new Url(FileFilesystem.PROTOCOL, "c:/mydir/subdir/anothersubdir/
    somefile.txt");
Url easyWay = dirUrl.getRelativeUrl("somefile.txt");
```

Relative Paths

The `Url` class has several methods for retrieving relative paths. This makes it easy to persist an `Url` relative to something. An `Url` is often persisted relative to a Project file or the project file's directory. That way, if the Project is copied to another location, or pulled to a different location from a Version Control System, it will be possible to reconstruct an `Url` that will work regardless of location.

```
Url projectDir = project.getProjectPath();
Url myUrl = <an url somewhere underneath projectDir>;
String relativePath = projectDir.getRelativePath(myUrl);
// Save relativePath to some file

...
// Later, in another routine, after having quit and restarted:
String relativePath = <read value from some file>;
Url myUrl = projectDir.getRelativeUrl(relativePath);
```

Chapter

6

Build system concepts

When a build is launched in Primetime, a `BuildProcess` is instantiated. The `BuildProcess`'s constructor takes either a node or an array of nodes.

The `BuildProcess` then gets an array of registered `Builder`s from the `BuilderManager` and passes each node, one at a time, to each `Builder`. It then recursively repeats those operations for all the children of the nodes.

Every time a `Builder` is passed a node, it determines if the node is of interest to the `Builder`; if so, it constructs one or more `BuildTask`s that will build the node. The `Builder` will also schedule where in the `BuildProcess` the `BuildTask` will execute.

After all nodes have been passed to all `Builders`, the `build()` method on `BuildProcess` is invoked, specifying which target(s) to execute. All `BuildTask`s that are ultimately dependencies of the specified target(s) will execute.

To write your own build OpenTool, you will typically need to define two classes:

1. One that extends `Builder`. An instance of this class will register itself with the `BuilderManager` and create one or more `BuildTask`s as necessary.
2. One that extends `BuildTask`. Instances of this class will do the actual building.

For a sample on how to use `Builder`s, see the `Obfuscator` sample in the JBuilder `samples/OpenToolsAPI/Build` directory.

Features and subsystems

Terminology

The Primetime build system is based on Ant, a Java-based build tool maintained by the Jakarta Project. Although it is not necessary to know anything about Ant to write build OpenTools, Primetime uses some of the same terminology as does Ant:

Build task A task that gets executed as part of the build process. Build tasks cannot exist standalone; they must belong to a target. If more than one build task belongs to a target, then, when the target is executed, the build tasks are executed in their order within the target. Build tasks are created by `Builder`s.

Target A collection of zero or more build tasks. Targets can have dependencies between them. For example, an EAR target will have an EJB target as a dependency—the EJB has to build before the EAR can build. Targets are created by `Builder`s.

Phase An existing target that JBuilder creates every time a `BuildProcess` is created. Phases serve as high-level markers for the build process, providing a broad framework into which `Builder`-created targets can be made dependencies. Although a phase could contain a build task, build tasks are typically created in their own targets, and those targets are made dependencies of the phases.

Phases

Primetime always creates the following phases:

- Clean
- Pre-compile
- Compile
- Post-compile
- Package
- Deploy

These phases have no dependencies on each other. This allows, for example, the Compile phase to be executed by itself, without having to execute the Pre-compile phase.

There are two phases, also always created by Primetime, that tie the above phases together via dependencies:

- Make: has Pre-compile, Compile, Post-compile, Package, and Deploy as its dependencies.
- Rebuild: has Clean and Make as its dependencies.

The build process

A build is started by, for example, the user selecting Make Project on the JBuilder menu. In that case, the following occurs:

1 A `BuildProcess` is instantiated, passing the project as a parameter:

```
BuildProcess buildProcess = new BuildProcess(projectNode);
```

2 The `BuildProcess` invokes `beginUpdateBuildProcess()` on all registered `Builder`s:

```
Builder [] builderArray = BuilderManager.getBuilders();
for (int i = 0; i < builderArray.length; i++) {
  builderArray[i].beginUpdateBuildProcess(this);
}
```

3 The `BuildProcess` invokes `updateBuildProcess` for every registered `Builder`, passing the project node. The children of the project are then passed to every registered `Builder`, and this continues recursively until there are no more children:

```
...
  addNode(builderArray, projectNode);
...
private void addNode(Builder[] builderArray, Node node) {
  for (int i = 0; i < builderArray.length; i++) {
    builderArray[i].updateBuildProcess(this, node);
  }
```

```
        Node[] subnodes = getBuildChildren(node);
        for (int index = 0; index < subnodes.length; index++) {
          addNode(builderArray, subnodes[index]);
        }
      }
```

4 The `BuildProcess` invokes `endUpdateBuildProcess()` on all registered `Builder`s:

```
Builder [] builderArray = BuilderManager.getBuilders();
for (int i = 0; i < builderArray.length; i++) {
  builderArray[i].endUpdateBuildProcess(this);
}
```

5 `BuildProcess.build()` is executed, passing a `true` to indicate that the process should occur in the background and passing the target named `make`:

```
buildProcess.build(true, "make");
```

As another example, if the user right-clicks a node in the project pane and selects Rebuild, the same process as above occurs with two exceptions:

1 The node the user clicked on is passed to the `BuildProcess` constructor, instead of the project node.

2 `rebuild` is passed as a parameter to the `build()` method, instead of `make`.

Registering a Builder

A `Builder` must register itself with the `BuilderManager`, in its `initOpenTool()` method:

```
public static void initOpenTool(byte majorVersion, byte minorVersion) {
  BuilderManager.registerBuilder(new MyBuilder());
}
```

`Builder`s must be listed in the OpenTool Build category.

Instantiating a build task

If a `Builder` creates a build task, it should do so in its `updateBuildProcess()` method. The `Builder` determines if the node parameter is of interest, and if so, it instantiates a build task using `BuildProcess.createTask()`. If the node is of no interest to the `Builder`, then the `Builder` does nothing.

This code determines if the node is an instance of `MyNode`, and if it is, instantiates a `MyBuildTask` in the target `MyBuildTarget`. If the target `MyBuildTarget` does not exist, it's created, and a new instance of `MyBuildTask` is added to the target. If the target already exists, a new instance of `MyBuildTask` is added to the existing target. The `Builder` then passes the node to `myBuildTask`. The build task will query the node as needed to perform its job.

```
public void updateBuildProcess(BuildProcess buildProcess, Node node) {
  if ( isMakeable(node) ) {
    MyBuildTask myBuildTask = (MyBuildTask)
buildProcess.create(MyBuildTask.class,
        "MyBuildTarget");
    myBuildTask.setNode((MyNode)node);
  }
}
public boolean isMakeable(Node node) {
  return node instanceof MyNode;
}
```

We have not yet established when, in the `BuildProcess`, that `MyBuildTask` will execute. See "Establishing dependencies" on page 55.

Instead of creating one `BuildTask` for every `Node` that a `Builder` will build, the `Builder` can alternatively create one `BuildTask` for all applicable nodes. For example, if you have 20 Java files in your project, JBuilder's JavaBuilder does not create 20 `BuildTask`s to compile one file at a time; it creates one `BuildTask` that compiles all 20 files at once. Here's one way we could modify the above code:

```
public void updateBuildProcess(BuildProcess buildProcess, Node node) {
  if ( isMakeable(node) ) {
    MyBuildTask myBuildTask = buildProcess.getFirstBuildTask("MyBuildTarget",
        MyBuildTask.class);
    if ( myBuildTask == null ) {
      myBuildTask = (MyBuildTask) buildProcess.create(MyBuildTask.class,
        "MyBuildTarget");
    }
    myBuildTask.addNode((MyNode)node);
  }
}
```

In the above code, the `buildProcess.getFirstBuildTask()` method call looks for the first instance of `MyBuildTask.class` inside the `MyBuildTarget` target. It either returns the instance, or `null` if there is no `MyBuildTask` instance, in which case we go ahead and create an instance.

Another way of accomplishing this is to store a reference to the `MyBuildTask` instance as a member variable of the `Builder`. If you take this approach, you must remember to release the reference to the `BuildTask` in the `endUpdateBuildProcess()` method. If you do not do so, Java will not be able to garbage collect the `BuildTask`, because the `Builder` instance is in memory for the whole session. See "Uses for beginUpdateBuildProcess() and endUpdateBuildProcess()" on page 56 for more details.

Writing a build task

As the Primetime build system is based on Ant, a build task has the same requirements as an Ant build task:

- It must have a parameterless constructor.
- It must have the following two methods:

```
public void setProject(org.apache.tools.ant.Project project);
public void execute() throws org.apache.tools.ant.BuildException;
```

The easiest way to write a build task is by extending `com.borland.primetime.build.BuildTask`. It already takes care of implementing the above methods and you only need implement the `build()` method. The `build()` method is passed a `BuildProcess` parameter. The `BuildTask` implementation should report any context-specific errors or warnings by invoking `BuildProcess.fireBuildProblem`. You can instead, or also, return `false` in the `build()` method, which will cause a generic error message to be reported.

Use `BuildProcess.fireBuildStatus` to report the progress of your `BuildTask`, in particular if your `BuildTask` takes a long time to execute. Use `BuildProcess.fireBuildMessage` to report informational messages that are neither errors nor warnings.

A build task typically has one or more setter methods. The `Builder` that creates the build task sets values on the build task by calling those setters. The build task then uses those values when it executes. In the previous section, the `Builder` simply passed the whole node to the build task. In other cases, the `Builder` may need to pass more values to the build task.

Establishing dependencies

As described earlier, build tasks are instantiated in their own targets in the `updateBuildProcess()` method. It is expected that a `Builder` will establish any necessary dependencies for the targets it created in `endUpdateBuildProcess()`. At that point all `Builder`s will have created all build tasks in their appropriate targets, and you can only establish dependencies between existing targets. `BuildProcess` has an `addDependency()` method to set up dependencies.

As a minimum, you want to make your target a dependency of a phase:

```
public void endUpdateBuildProcess(BuildProcess buildProcess) {
  ...
  buildProcess.addDependency("MyBuildTarget", Phase.POST_COMPILE_PHASE);
  ...
}
```

This means that when it is time for the Post-compile phase to execute, `MyBuildTarget` will be executed first, since it is a dependency of the Post-compile phase. There may be several other targets that are dependencies of the Post-compile phase. If we leave it at this, we do not know when this will execute in relation to other dependencies of the Post-compile phase. Depending on what `MyBuildTask` does, this may be enough. You may only care that `MyBuildTarget` is executed after the Compile phase. But let's say that you want `MyBuildTarget` to execute **after** the JNI target, which is also a dependency of the Post-compile phase. In that case, you want the JNI target to be a dependency of `MyBuildTarget`:

```
buildProcess.addDependency(BuildTargets.JNI, "MyBuildTarget");
```

You do not need to verify that there is an existing JNI target for the above code to work. If there isn't a JNI target, one will be created, and it will be made a dependency of `MyBuildTarget`.

Using DependencyBuilder to add a dependency to a phase

There is another way to make a target a dependency of a phase—by invoking `DependencyBuilder.registerTarget()`. Register it in your `Builder`'s `initOpenTool()` method:

```
DependencyBuilder.registerTarget("MyBuildTarget", Phase.POST_COMPILE_PHASE);
```

In addition to no longer needing to add the dependency in `endUpdateBuildProcess()`, another class, `DependencyPanel`, uses registered targets to present a list of choices of targets that the user can select to be dependencies via the IDE's UI. The Properties page of an External Build Task uses the `DependencyPanel` component, and you can see that, when you click on the toolbar buttons, a list of targets that are dependencies of that phase are presented.

isMakeable() and isCleanable()

When you right-click a node in the project pane, there are 3 different groups of build menu choices that can be displayed on the context menu:

1. No build menu choices are displayed. The node is not buildable; a `.txt` file, for example.
2. A Make menu choice is displayed. The node can be made, but there is nothing that cleans its output; an External Build Task, for example.
3. Clean, Make, and Rebuild menu choices are displayed. The node can be made or cleaned, or both (rebuilt); a `.java` file, for example.

The group of menu choices displayed is determined by the `BuildProcess` calling the `isCleanable()` and `isMakeable()` methods on registered `Builders`, until either there is an `isCleanable()` method that has returned `true` **and** an `isMakeable()` method that has returned `true`, or until all `Builders` have been queried.

Since these methods are invoked when a context menu is being constructed, these methods should be very quick to execute. For example:

```
public boolean isMakeable(Node node) {
  return node instanceof MyNode;
}
```

To ensure that `updateBuildProcess()` builds what you report as makeable, and doesn't build what you say is not makeable, you should typically invoke `isMakeable()` from inside your `updateBuildProcess()` method:

```
public void updateBuildProcess(BuildProcess buildProcess, Node node) {
  if ( isMakeable(node) ) {
    // Construct build task
  }
}
```

The same applies to `isCleanable()`, if it returns a different value than `isMakeable()`.

Clean

If your `Builder` creates a build task that generates some build output, you may want the `Builder` to also create a task that cleans, or deletes, that build output.

When your `Builder`'s methods are invoked, the `Builder` does not know what target (`clean`, `make`, `rebuild`, etc.) is going to be executed. Therefore, the `Builder` must schedule all necessary tasks without knowing which one(s), if any, will actually be executed. In other words, if applicable, the `Builder.updateBuildProcess()` method should create a build task to build, and it should create another build task to clean, and the build should work if neither, either, or both of the tasks are actually executed in the build process.

Depending on what sort of build output your build task generates, you may be able to use `CleanBuildTask`, which has various methods to delete different types of output. By calling `CleanBuilder.getCleanBuildTask()`, you can get a reference to the single instance of a `CleanBuildTask` that normally exists in a `BuildProcess`. If the `CleanBuildTask` does not have a method to schedule the deletion of your output, define a new build task, and make it a dependency of the Clean phase.

If your `Builder` does schedule a clean task, then its `isCleanable()` method should return `true`. See "isMakeable() and isCleanable()" on page 55 for more details.

Uses for beginUpdateBuildProcess() and endUpdateBuildProcess()

As discussed earlier, the `endUpdateBuildProcess()` method is where a `Builder` should set up its dependencies. Another usage for `endUpdateBuildProcess()` method, possibly in combination with `beginUpdateBuildProcess()`, is to initialize and/or clear variables. `Builder`s, once instantiated, are in memory for the duration of a session. You should free up any references to objects in the `endUpdateBuildProcess()` method so that the referenced objects can be garbage collected.

getMappableTargets()

The IDE user has the option of configuring which build targets appear on the Project menu and on the build toolbar drop-down menu, via Project Properties|Build|Menu Items. The user can also specify which build target is executed before running, debugging, and optimizing.

In all of the above cases, the build targets that the user can choose from are determined by `Builder.getMappableTargets`. The UI invokes `BuildProcess.getMappableTargets(projectNode)`, which recursively iterates through the project and all of its children, passing each node, one at a time, to `Builder.getMappableTargets` of all the registered `Builder`s. For example, among other `BuildAction`s, Primetime itself has a `Builder` that returns `BuildAction`s for Project|Make, and Project|Rebuild. This is how those menu choices and toolbar buttons appear.

The `BuildAction` class extends `BrowserAction`. It has three abstract methods:

- `getNodes` — the node(s) which invoking this `BuildAction` will build
- `getTargets` — the name(s) of the target(s) to invoke to force the node to be built
- `getKey` — returns a `String` that can used to recreate the `BuildAction`. This key may be written out to a properties file.

  For example, menu configurations that you save via Project Properties|Build|Menu Items are stored as keys returned by `BuildAction`s in your project's `.local` file. Because it is written out to a properties file, it must not contain any `\n` or `\r` characters (and this is why object serialization is not used).

Example: BuildAction

```
public BuildAction [] getMappableTargets(Node node) {
  if ( node instanceof MyNode ) {
    return new BuildAction [] { new MyBuildAction((MyNode)node) };
  }
  return super.getMappableTargets(node); //Default implementation returns an
      empty array
}
...
public class MyBuildAction extends BuildAction {
  private MyNode myNode;

  public MyBuildAction(MyNode myNode) {
    super(myNode.getDisplayName(), // a name for the action
        '%', // mnemonic, in this case none
        myNode.getDisplayName(), // text that appears on menu
        myIcon, // a javax.swing.Icon
        myNode.getDescription())); // the long description
    this.myNode = myNode;
  }

  public Node [] getNodes() {
    // Return the node that will get built when this action is invoked
    return new Node [] {myNode};
  }

  public String [] getTargets() {
    // We specify the make phase, because the build task will
    // ultimately be a dependency on that phase.
    return new String [] {Phase.MAKE_PHASE};
  }
```

```
  /**
   * The getKey() method returns a String with two values separated by a
       semi-colon:
   * 1. The name of the class containing a static getBuildAction() method
   * 2. An arbitrary parameter that the getBuildAction() method uses to
   *    to create the BuildAction.
   */
  public String getKey() {
    StringBuffer sb = new StringBuffer();
    sb.append(getClass().getName()); // This class contains getBuildAction()
    sb.append(";");
    sb.append(myNode.getDisplayName()); // Save the name of the node
    return sb.toString();
  }

  /**
   * The getBuildAction() method recreates a BuildAction based on a key
   * that had previously been generated by MyBuildAction.getKey().  In this
   * case, instances of MyNode can only be children of the Project node,
   * and all have unique names.
   * If neither of above two cases were true, this code, and probably
   * getKey() as well, would have to be changed accordingly.
   */
  public static BuildAction getBuildAction(Project  project,  String param) {
    Node [] children =  project.getChildren();
    for ( int i = 0; i < children.length;  i++ ) {
      if ( children[i] instanceof MyNode &&
          ((MyNode)children[i]).getDisplayName.equals(param)) ) {
        return new MyBuildAction((MyNode)node);
      }
    }
    return null;
  }
}
```

You can programmatically execute a build using a `BuildAction`:

```
BuildProcess buildProcess = new BuildProcess(buildAction.getNodes());
buildProcess.build(true, buildAction.getTargets());
```

The BuildAware interface

The `BuildAware` interface is implemented by some nodes. In the opening paragraphs of this document and in “Registering a Builder” on page 53, we said that the children of each node are recursively passed to all the `Builder`s. In “Registering a Builder” on page 53, there is a code snippet that calls `BuildProcess.getBuildChildren()` to get the children of a node. The implementation of `BuildProcess.getBuildChildren()` is:

```
private static Node [] getBuildChildren(Node node) {
  Node [] children;
  if ( node instanceof BuildAware ) {
    children = ((BuildAware)node).getBuildChildren();
  }
  else {
    children = node.getChildren();
  }
  return children;
}
```

If you create your own node type and don’t want the `BuildProcess` to build what is returned by the node’s `getChildren()` method, the node type must implement `BuildAware`.

Communication between build tasks

If one build task needs to communicate with another, it can do this with these `BuildProcess` methods:

- `getProperty()`
- `setProperty()`
- `removeProperty()`

For example, `Task1` invokes `BuildProcess.setProperty("task1key", "task1value")`, and `Task2` can query the property with `BuildProcess.getProperty("task1key")`.

Another approach is for a build task or `Builder` to get a reference to another build task. Sometimes a `Builder` might manipulate two or more different types of build tasks. For example, JBuilder's Java2IIOP Builder sets up a Java2IIOP build task, which will produce `.java` files. Those `.java` files need to be compiled by a Java compile task. The `Builder` gets an existing Java compilation build task, or creates one if one does not exist. The exact names of the `.java` files produced by Java2IIOP are not known until build time. At that point, the Java2IIOP build task gets a reference to the Java compilation build task, which has not yet executed, and adds those just-generated `.java` files to the list of files the Java compilation build task must compile. This approach will only work if the Java2IIOP build task executes *before* the Java compilation build task.

`BuildProcess` has the following methods to help you find an existing task, if you know the build task's class and in what target it exists:

```
public Object getFirstBuildTask(String targetName, Class clazz);
public Object [] getBuildTasks(String targetName, Class clazz);
```

Performance

A `Builder` can instantiate a build task that ends up not getting executed. This can legitimately occur, based on the build target that ends up being executed. Because of this, a `Builder` should execute as fast as possible, and defer as much work as possible to the build task. Pass the minimum amount of information necessary to the build task, and let the build task manipulate that information. Also, see "isMakeable() and isCleanable()" on page 55 for more performance considerations.

Using an existing Ant build task

It is possible to use existing Ant build tasks within the Primetime build process. You will still need to write your own `Builder`. The `Builder` will instantiate the Ant task via the `BuildProcess.createTask()` method. Since there is no XML file, you must programmatically set the Ant task's properties.

Cancellation

While building inside the IDE, a status dialog with a Cancel button is displayed. Once the user presses the Cancel button, the `BuildProcess.isCancelled()` method returns `true`. The build process checks that value between the execution of each target, and cancels the build.

If your build task takes a long time to execute, the build task should poll the `BuildProcess.isCancelled()` method while it is executing, and if that method returns `true`, it should stop executing.

Errors

If any errors occur in executing your build task, you should generally use `BuildProcess.fireBuildProblem()` to report them. Optionally, your `BuildTask.build()` method can return `false` when there is an error. Returning `false` causes the `BuildTask` base class to invoke the `BuildProcess.fireBuildProblem()` method with a generic error message.

Because you can probably provide more context-specific error messages from inside your `BuildTask`, you will probably always want to return `true` from the `BuildTask.build()` method, even if there are errors, and report errors yourself from within your `BuildTask` implementation.

Note that invoking the `BuildProcess.fireBuildProblem()` method with its error parameter set to `true` does not necessarily cancel the build process. This will only happen if the `BuildProcess.isAutoCancelled()` method returns `true`. The value that method returns is controlled via the IDE's UI, in Project|Project Properties|Build, by the Cancel Build On Error check box.

Build extensions

The `Builder` class has two methods that are used by Primetime to determine which files to display underneath automatic package nodes:

```
public String [] getBuildExtensions(Project project);
public Map getBuildExceptions(Project project);
```

When JBuilder does automatic package discovery for a project, it queries all `Builder`s for an array of file extensions that the `Builder`s build. All files with matching file extensions will then be displayed underneath automatic package nodes. For example, the `JavaBuilder`, which is the `Builder` for `.java` files, returns `new String [] {".java"}`, and the `SqljBuilder`, which is the `Builder` for `.sqlj` files, returns `new String [] {".sqlj"}`.

`Builder` has a convenience method, `getRegisteredExtensions(Class nodeType)`, which returns an array of extensions registered to a particular type of node. The actual `JavaBuilder` implementation of `getBuildExtensions()` is to invoke `getRegisteredExtensions()`, passing the JavaFileNode class:

```
public String [] getBuildExtensions(Project project) {
  return getRegisteredExtensions(JavaFileNode.class);
}
```

This method is particularly useful for `Node`s that have multiple file extensions registered.

You can fine-tune which files are displayed underneath automatic package nodes by implementing `getBuildExceptions()`. This function returns a `Map`, where the keys in the `Map` are `com.borland.primetime.vfs.Url`s, and the values are `Boolean`s. An example of a `Builder` that does this is the `ResourceBuilder`, which schedules build tasks to copy resources from the source directory to the output directory.

For example, by default, all image files in the project's source paths are copied. However, there is UI the user can use to override both at a project level, as well as at an individual file level, which of those image files are copied. The pseudo-code for the `ResourceBuilder` to honor the override of individual files would look something like this:

```
public String [] getBuildExtensions(Project project) {
  // Note: this is not the complete list of resource extensions
  // supported by Primetime, and the list also varies depending
  // on project settings.  This is a simplified version of the
  // ResourceBuilder code that demonstrates a concept.
  return getRegisteredExtensions(ImageFileNode.class);
}
```

```
public Map getBuildExceptions(Project project) {
  // Note: Again, this is just demonstrating a concept, and is a
  // very simplified version of what the ResourceBuilder actually does.
  Map map = new Hashmap();
  ...
  // Suppress a particular URL:
  map.put(<Url of "somepath/foo.gif">, Boolean.FALSE);
  return map;
}
```

With the above code, all image files (`.gif`, `.jpg`, `.jpeg`, etc.) in the sourcepath will appear under automatic packages, except for `somepath/foo.gif`.

Excluded packages

`Builder`s that process `com.borland.jbuilder.node.PackageNode`s should not process excluded package nodes. Excluded packages are, by definition, excluded from the build process. If your `Builder` processes `PackageNode`s, use the method `com.borland.jbuilder.build.BuildUtil.isExcludedPackageNode()` to determine if a package node is excluded:

```
public boolean isMakeable(Node node) {
  return node instanceof PackageNode && !BuildUtil.isExcludedPackageNode(node);
}
```

Command-line builds

JBuilder allows projects to be built from the command line. For example,

```
jbuilder -build myproject.jpx
```

When a command-line build executes, the GUI portion of JBuilder is not loaded. This means your `Builder` and `BuildTask` implementations should not access any GUI, such as the `Browser` class. If they do so, a command-line build will most likely fail.

From your `BuildTask` implementation, always report build progress, warnings, status, errors, etc. using the `fireBuildxxx()` methods provided in `BuildProcess`.

Tracking output

Your build OpenTool may need to track the output of the build system. For example, if your tool fixes up the `.class` files produced by a build, you need to know which `.class` files were created.

As of Primetime 4.5 (which shipped with JBuilder 8), a new method was added to the `BuildListener` class for tracking the creation and deletion of build output:

```
/**
 * Reports that output has been created or deleted by the build process. The
 * buildOutputEvent contains the details of the event.
 *
 * @param buildProcess the build process in question
 * @param buildOutputEvent the event that describes the creation or deletion
 * of build output
 * @since Primetime 4.5
 */
public void buildOutputEvent(BuildProcess buildProcess,
    BuildOutputEvent buildOutputEvent) {}
```

For example, the following code prints out all `BuildOutputEvent`'s that occur in a build:

```
buildProcess.addBuildListener(new BuildListener() {
  public void buildOutputEvent(BuildProcess buildProcess,
      BuildOutputEvent buildOutputEvent) {
    System.out.println("Url " + buildOutputEvent.getOutputUrl() + " was " +
        (buildOutputEvent.isCreated() ? "created" : "deleted"));
  }
});
```

`BuildOutputEvent`s are typically subclassed. You can use `instanceof` to focus on a particular type of output. For example, if you are interested only in the creation and deletion of archives:

```
buildProcess.addBuildListener(new BuildListener() {
  public void buildOutputEvent(BuildProcess buildProcess,
      BuildOutputEvent buildOutputEvent) {
    if (buildOutputEvent instanceof ArchiveOutputEvent) {
      // Process here
      ...
    }
  }
});
```

If your build OpenTool needs to inform the build process of output that your OpenTool has created and/or deleted, there is similarly a new method in `BuildProcess`:

```
/**
 * Fires a build output event notification.  All registered build process
 * listeners will have their buildOutputEvent method executed.
 *
 * The event contains the details of what output was either deleted or
 * created.
 *
 * @param buildOutputEvent an event that contains details about the  output
 */
public void fireBuildOutputEvent(BuildOutputEvent buildOutputEvent);
```

Your build OpenTool should invoke the above `BuildProcess.fireBuildOutputEvent()` method as appropriate while it is executing. For example, assuming your build task creates an `Url`, then you can notify the build system like this:

```
public boolean build(BuildProcess buildProcess) {
  Url myUrl;
  ...
  buildProcess.fireBuildOutputEvent(new BuildOutputEvent(this, myUrl, true));
}
```

It is not a requirement that every build task invoke the `BuildProcess.fireBuildOutputEvent()` method to notify the build system about generated output. Because of this, your `BuildListener` will not necessarily be notified of all output generated by the build system. It will only be notified of output generated by those build tasks that inform the build system.

As of JBuilder 8, the following JBuilder build tasks notify the build system about their output:

- Java compilation
- Clean
- Archiving
- EJB
- EAR

Build tasks that invoke `fireBuildOutputEvent()` may subclass `BuildOutputEvent`. For example, when JBuilder compiles a `.java` file or deletes a `.class` file, `fireBuildOutputEvent()` is invoked with a `ClassOutputEvent` subclass of `BuildOutputEvent`. You can use this to listen for particular types of output. Using the `ClassOutputEvent` example, if you are only interested in the creation of `.class` files by the build system, you can ignore all `buildOutputEvent` invocations that are not passed an instance of `ClassOutputEvent`.

Additional JBuilder build tasks will notify the build system of their output over time.

Building Build Output

A build can produce output where the output in turn needs to be built. In other words, the build output might be source that needs to be built. For example, if a project has an IDL file, depending on what type of project is open in the IDE, the building of the IDL file might yield `.java`, `.c`, etc. source files. Those source files need, in turn, to be compiled into `.class`, `.obj`, etc. files.

In Primetime 4.7 (which shipped with JBuilder X), a new method was introduced to the `Builder` class to handle this case:

```
/**
 * ...
 * @param buildProcess the build process
 * @param node a node that was created or deleted
 * @param targetName the name of the target
 * @param buildTask the build task that created
 * @param created whether this node was created or deleted
 * @since Primetime 4.7
 */
public void updateBuildProcess(BuildProcess buildProcess, Node node,
    String targetName, Object buildTask, boolean created) {}
```

This method is fired during a build. When it is fired, it indicates that a build task has created or deleted `node`. If the node was created, the `Builder` can either schedule a build task to build the node, or it can add the node to an already scheduled build task. If the node was deleted, the `Builder`, depending on the particular build task in question, may need to remove the node from the build task (some build tasks may not care if the node actually exists, and some may).

If the `Builder` schedules a new build task, it must schedule it in a target that already exists. Just before a build starts executing, all dependencies between targets are resolved—it is too late to add any new targets to the build process. Additionally, the build task needs to be scheduled in a target that has not yet been executed. If the build task is scheduled in a target that has already executed, the build task will not execute, as the target will not be executed a second time. To determine whether a target can still be executed while a build is running, invoke the `BuildProcess.canStillExecuteTarget()` method.

The `updateBuildProcess(BuildProcess buildProcess, Node node, String targetName, Object buildTask, boolean created)` method is fired on all `Builder`s when a build task fires a `BuildProcess.fireBuildOutputEvent()` method. Thus, individual build tasks that want their output processed for further building are responsible for invoking `BuildProcess.fireBuildOutputEvent()` as appropriate.

Executing a Build

You can programmatically execute a build by instantiating a build process for the node or nodes that you want to build, and by then invoking the build process' `build()` method:

```
public void buildNode(Node node) {
  BuildProcess buildProcess = new BuildProcess(node);
  buildProcess.build(false, Phase.MAKE_PHASE);
}
```

If you want to build multiple nodes, then pass an array of nodes to the `BuildProcess` constructor.

The first parameter of the `build()` method indicates whether to build in the background or in the foreground. If the parameter is `false`, then the build is executed in the current thread. If the parameter is `true`, then the build is launched in a new thread. In the case of a background build, when program flow returns from the call to the `build()` method, there is no guarantee that the build has finished executing, as the build was spun off in a separate thread. If the background parameter is set to `false`, then it is guaranteed that the build will have finished executing when program flow returns from the call to `build()`.

You will typically want to add a `BuildListener`. It is the only way to know if an error occurred during the build. If you have launched a background build, it is the only way to know that the build has finished:

```
public void buildNode(Node node) {
  BuildProcess buildProcess = new BuildProcess(node);
  buildProcess.addBuildListener(new BuildListener() {
    public void buildProblem(BuildProcess process, Url url,
        boolean error, String message, int line,
        int column, String helpTopic) {
      if (error) {
        System.out.println("An error occurred");
      }
      else {
        System.out.println("A warning occurred");
      }
    }

    public void buildFinish(BuildProcess process) {
      System.out.println("Build finished!");
    }
  });
  buildProcess.build(true, Phase.MAKE_PHASE);
}
```

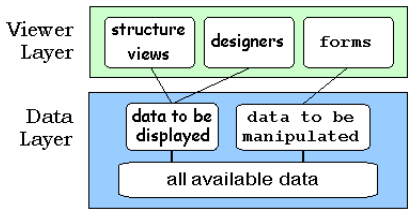
Chapter 7

XMT concepts

The XMT (eXtensible Modelling Toolkit) is a framework for creating and assembling two-way tools on the Primetime platform, using an MVC (Model-View-Controller) pattern. The XMT unifies the data modelling concepts into one coherent, scalable, extensible format.

Purpose and structure

The XMT provides a structured, consistent way for multiple viewers and forms to reference the same data. This way, designers, forms, and structure views using the same set of data can maintain synchronization and can significantly reduce loading time when switching from one view to another.



Abstract classes and interfaces constitute the XMT's core. The programmer is expected to extend these by using implementation base classes where they're available. The implementation base classes provide commonly-needed "handles", usually in the form of classes with "Default" or "Abstract" prefixes, simplifying the process of extending and implementing the different aspects of this framework.

Purpose

The philosophy behind the design/development of the XMT is to keep the code:

- Streamlined
  - Reduces or eliminates subclassing wherever possible, which mitigates class explosion relating to the inherent problems of binding data to different models.
  - Extracts functionality expected to be used by fewer clients into helper classes, which minimizes the number of required methods exposed in external interfaces.
  - Where appropriate, separates default and helper implementations into `.impl` subpackages.

- Easy to maintain
  - Defines interfaces to be used by OpenTools clients which are backed by abstract classes. These abstract classes are extended in order to implement the interface.
  - Where feasible, optimized and/or strategized default or base implementations reduce the amount and complexity of code required by XMT clients.
- Easy to use
  - Separates individual services to keep clients from having to use the XMT on an all-or-nothing basis.
  - Depends as little as possible on external packages, to maximize applicability.
  - Names classes and interfaces that define the XMT using the prefix “Xmt”.

Structure

The model

The core of XMT is a generic, hierarchical data model which allows maintenance of a tree of unlimited breadth and depth.

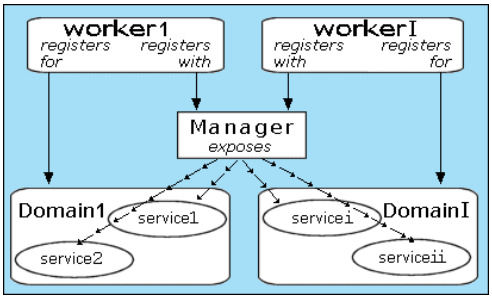
The tree is a composite structure of `XmtNode`s which can contain an arbitrary collection of `XmtProperty`s and child `XmtNode`s of client-defined types. The model:

- Stores, navigates, and looks up data via a pluggable set of optimized component collections.

  A *component*, in this case, is a node or property.
- Provides unique, immutable identifiers for nodes, either automatically-generated or client-specified.
- Automatically tracks component parentage and modification.
- Types nodes hierarchically, supporting inheritance of common data ancestry.
- Optionally, tracks and enforces component mutability.
- Optionally, provides observation of models via static or instance-based listeners, managed by a pluggable set of optimized listener collections.
- Optionally, automatically propagates change events, simplifying model observation.

The management

Most of the XMT framework is centered around providing services for working with an XMT model. These services must be extensible at runtime. Therefore, functionality not specifically related to data model management is in assistant subsystems, which are used by individual XMT clients. These subsystems are arranged in a Manager/Worker pattern where client-implemented workers (i.e. `XmtReader`s) are registered with a manager (i.e. `XmtReaderManager`). In order to avoid polling all workers for every request, a worker can be registered for a specific `XmtDomain`.



Each manager exposes a set of services (e.g. `XmtReaderManager.read()`) which are implemented at runtime by delegating requests to the registered workers. Workers implement the service methods exposed by the manager and implement a method for determining whether a worker can handle a specific request (e.g. `XmtReader.canRead()`).

A service must give a worker contextual information so the worker knows if it can determine and service a request. To do so, the service requires a parameter list object which is a descendant of `XmtContext` or `XmtWorkerContext`. This allows for the contextual information to be extended on either a client or request basis, without having to change the API. Each `XmtWorkerContext` contains a reference to an `XmtMessageReporter` which can be used as a standard method for reporting messages (status, problem, validation, etc.) to the user, even when running in command-line mode.

Note For more specific implementation details regarding the XMT, please refer to the Javadoc for individual classes, especially the manager classes.

Features and subsystems

Basically, the subsystems are designed to be used in this order:

1. Create nodes and properties (`xmt.model`, `xmt.creator`, `xmt.configurator`)
2. Specify how the data will appear to the user (`xmt.view.*`)
3. Manipulate the data at runtime (`xmt.visitor`, `xmt.validator`)
4. Read and write the data between models and viewers (`xmt.reader`, `xmt.writer`, `xmt.resolver`)

The subsystems available for implementation are:

- **`Model`**
  The starting point. Manages identification of, inheritance of, and changes to, data components.
- **`Creator`**

  Implementation of the factory pattern for creating `XmtNode` and `XmtProperty` instances.
- **`View`**

  Used to load, save, and refresh components in context. Extended by the subpackages below.
  - **`Form`**

    Used to create form-based views (such as the DD Editor) for an XMT model. This includes Swing widgets which can automatically synchronize themselves with an XMT model.
  - **`Structure`**

    Used to create structure pane content which visualizes the components of an XMT model, including support for runtime extension of structure content and context actions.
- **`Visitor`**

  Used to implement the visitor pattern against the components of an XMT model.
  - **`Validator`**

    A specialized implementation of Visitor which is used for validating the data contained in a model.
  - **`Configurator`**

    A specialized implementation of Visitor which is used for configuring `XmtNode` and `XmtProperty` instances.

- **Reader**

  Used to populate an XMT model from one or more data sources (i.e. Java sources, XML files, databases, etc.).

  - **Resolver**

    Primarily used when merging the models returned by individual readers to determine if there is an equivalent component in a target model for a given component from a source model.

- **Writer**

  Used to persist an XMT model to one or more data destinations (Java sources, XML files, databases, etc.)

XMT has compile-time and run-time dependencies on some other Primetime subsystems in order to provide the proper context or tie-in required by its subsystems. The most notable are:

- `xmt.XmtContext` has a compile-time dependency on `ide.Browser` and `node.Node`, but neither is required at run-time.
- `xmt.XmtMessage` extends `ide.Message` in order to richly support display of messages in the Browser's message view.
- `xmt.view` subsystems have additional dependencies on `ide.NodeViewer` and its associated helper classes and lifecycle.
- `xmt.view.XmtViewContext` extends `xmt.XmtContext` to have a run-time dependency on `ide.Browser` and `node.Node`.

Creator

`com.borland.primetime.xmt.creator`

Used as an implementation of the factory pattern for creating `XmtNode` and `XmtProperty` instances. For instance, this can be used to change the component implementation which is instantiated or to customize properties of the created component, such as whether or not property and/or child node collections should be ordered.

`XmtDefaultNodeCreator` is an `XmtNodeCreator` implementation which can be used to customize the creation of `XmtDefaultNode` instances according to an `XmtNodeType` filter without having to implement and register your own `XmtNodeCreator`. Specifically, this class allows you to change the `XmtNodeIdGenerator`, `XmtNodeCollection`, `XmtPropertyCollection`, and `XmtNodeListenerCollection` that will be used by the created node.

`XmtDefaultPropertyCreator` is an `XmtPropertyCreator` implementation which can be used to customize the creation of `XmtDefaultProperty` instances according to a property key filter without having to implement and register your own `XmtPropertyCreator`. Specifically, this class allows you to change the `XmtPropertyListenerCollection` and default value that will be used by the created property.

View

`com.borland.primetime.xmt.view`

`View` is the basis for the subpackages which provide different ways of displaying data at runtime. These subpackages include `view.form` and `view.structure`.

Form

`com.borland.primetime.xmt.view.form`

Used to create form-based views (such as the DD Editor) for an XMT model. This includes a set of reusable Swing widgets which can automatically synchronize themselves with an XMT model, thereby mostly reducing `XmtForm` creation to regular Swing development.

`XmtForm` defines an interface for creating user interfaces which display and/or modify some subset of an XMT model. A Form implementation is expected to be a regular Swing component which has been extended to implement the `XmtForm` interface or a subclass of `XmtAbstractForm`, with the user interface implemented via a composition of `XmtWidget`s that are to be managed automatically.

`XmtFormsViewer` is a `JTabbedPane`-based `NodeViewer` viewer component implementation which displays one or more `XmtForm`s for a specific `XmtNode`.

`XmtValueListResolver` defines a simple interface for use by components which are expected to display a set of values the user can choose from (i.e. `JXmtComboBox`, `JXmtCheckList`, etc.), including support for display values being different from the values which are stored in the model. `XmtStaticValueListResolver` is an `XmtValueListResolver` which is backed by a static list of values and, optionally, a matching list of display values.

`XmtWidget` implementations for common Swing components can be found in the `com.borland.primetime.xmt.view.form.widget` and `com.borland.jbuilder.xmt.view.form.widget` packages.

Structure

`com.borland.primetime.xmt.view.structure`

Used to create structure pane content which visualizes the components of an XMT model. This includes support for runtime extension of structure content and context actions.

`XmtTreeStructure` is a `JTree`-based `NodeViewer` structure component implementation whose content (`XmtTreeStructureNode`s) and context actions can be implemented/ extended via implementations of `XmtTreeStructureNodeProvider` and `XmtTreeStructureContextActionProvider` respectively.

`XmtNodeTypeTreeStructureNode.createStructureNode()` can be used to easily created a collector node with a child structure node representing each child model node of a specified `XmtNodeType`.

Visitor

`com.borland.primetime.xmt.visitor`

Used to implement the visitor pattern against the components of an XMT model. For instance, the `Visitor` subsystem is used as the basis for the Configurator and Validator subsystems.

Additional visitation actions can be implemented by creating a new instance of `XmtVisitorAction` and passing it to `XmtVisitorManager.visit*()`.

Configurator

`com.borland.primetime.xmt.configurator`

A specialized implementation of `Visitor` which is used for configuring `XmtNode` and `XmtProperty` instances. For instance, configurators can be implemented to initialize default property values and child nodes whenever a model component is created because of a user action.

`XmtNamePropertyNodeConfigurator` is an `XmtNodeConfigurator` implementation which sets a specified property to a `String` containing a specified prefix, followed by a number

which is unique among the siblings of the specified type. In order to use this functionality, a client merely needs to register an `XmtNamePropertyInfoSet`.

Note If an `XmtNode` is expected to have exactly one child of a specific type, this relationship should be initialized by the reader and by a configurator for the parent node. This allows views to depend on the child node being there, instead of having to modify the model in inappropriate or invalid circumstances.

Validator

`com.borland.primetime.xmt.validator`

A specialized implementation of `Visitor` which is used for validating the data contained in a model. For instance, you can use it to check if required properties are specified and contain appropriate values.

Reader

`com.borland.primetime.xmt.reader`

Used to populate an XMT model from data sources (i.e. Java sources, XML files, databases, etc.). Multiple readers can be registered to service a specific context, in which case the models are merged.

`XmtAbstractUrlsReader` is a base class for implementing `XmtReader`s that build a model from one or more `Url`s. It provides an implementation of `XmtReader.needsRead()` based on when each `Url` was last modified.

Resolver

`com.borland.primetime.xmt.resolver`

Primarily used to determine if there is an equivalent component in a target model for a given component from a source model.

`XmtKeyPropertySetResolver` is an `XmtNodeResolver` implementation that compares a set of properties, specified via a registered instance of `XmtKeyPropertySet`, which are expected to define a unique key for nodes of the specified `XmtNodeType`.

Writer

`com.borland.primetime.xmt.writer`

Used to persist an XMT model to one or more data destinations (i.e. Java sources, XML files, databases, etc.). Multiple writers can be registered to service a specific context where each would write some subset of the model.

`XmtAbstractUrlsWriter` is a base class for implementing `XmtWriter`s which save a model to one or more `Url`s. It implements everything except the actual content generation, which is expected to be done via implementations of `BufferUpdater`.

Implementation examples

See also

- The project `XMTExample` in the samples/OpenToolsAPI subdirectory `XmtPropertiesExample`.

Example: XmtForm

Objective Display and allow modification of some properties in an XMT model.

```
public class ExampleXmtForm
    extends XmtAbstractForm {

  private static final XmtNodeType EXAMPLE_NODE_TYPE = new
      XmtNodeType("ExampleNodeType");

  private static final XmtDomain EXAMPLE_DOMAIN = new XmtDomain("ExampleDomain");

  private JLabel label = new JLabel();
  private JXmtTextField textField = new JXmtTextField(this, "examplePropertyKey");

  private void jbInit() {
    label.setText("Name:");
    label.setLabelFor(textField);
    add(label);

    textField.setColumns(10);
    add(textField);
  }

  public ExampleXmtForm(XmtFormContext formContext) {
    super(formContext, true);

    jbInit();
  }

  public String getDisplayName() {
    return "Example Form";
  }

  public void updateControls() {
    // Honor the read-only status of the XmtFormContext
    boolean isEnabled = (getFormContext().getReadOnly() == false);

    label.setEnabled(isEnabled);
    textField.setEnabled(isEnabled);
  }

  public static void initOpenTool(byte majorVersion,
                                  byte minorVersion) {
    // Only provide forms for a specific XmtDomain
    XmtFormProviderManager.registerFormProvider(new Provider(), EXAMPLE_DOMAIN);
  }

  public static class Provider
      extends XmtAbstractFormProvider {

    public boolean canProvideForm(XmtFormProviderContext context) {
      // Only provide forms for a specific XmtNodeType
      return (context.getFormContext().getModelNode().getType().equals(EXAMPLE_NODE_TYPE)
          == true);
    }

    public XmtForm provideForm(XmtFormProviderContext context) {
      return new ExampleXmtForm(context.getFormContext());
    }

  }

}
```

Example: XmtNodeConfigurator

Objective Set the default value of a specific property whenever a certain type of `XmtNode` is created.

```
public class ExampleXmtNodeConfigurator
    extends XmtAbstractNodeConfiguratorVisitor {

  private static final XmtNodeType EXAMPLE_NODE_TYPE = new
      XmtNodeType("ExampleNodeType");

  private static final String EXAMPLE_PROPERTY_KEY = "PropertyWhichNeedsDefaultValue";

  private static final XmtDomain EXAMPLE_DOMAIN = new XmtDomain("ExampleDomain");

  public boolean canConfigureNode(XmtNodeConfiguratorVisitorContext context) {
    // Only configure nodes of a specific XmtNodeType
    return (context.getTargetNode().getType().equals(EXAMPLE_NODE_TYPE) == true);
  }

  public XmtVisitorStatus configureNode(XmtNodeConfiguratorVisitorContext context) {
    // Set a default value for a specific property
    context.getTargetNode().setPropertyValue(EXAMPLE_PROPERTY_KEY, "defaultValue");

    return XmtVisitorStatus.STATUS_CONTINUE;
  }

  public static void initOpenTool(byte majorVersion,
                                  byte minorVersion) {
    // Only configure nodes for a specific XmtDomain
    XmtVisitorManager.registerNodeVisitor(new ExampleXmtNodeConfigurator(),
        EXAMPLE_DOMAIN);
  }

}
```

Example: XmtNodeResolver

Objective Define a new method for determining when two `XmtNode`s of a certain type are equivalent.

```
public class ExampleXmtNodeResolver
    extends XmtAbstractNodeResolver {

  private static final XmtNodeType EXAMPLE_NODE_TYPE = new
      XmtNodeType("ExampleNodeType");

  private static final XmtDomain EXAMPLE_DOMAIN = new XmtDomain("ExampleDomain");

  public boolean canResolveNode(XmtNodeResolverContext context) {
            // Only try resolution for source nodes with a specific XmtNodeType
    return ((context.getSourceNode().getType().equals(EXAMPLE_NODE_TYPE) == true) &&
            // For this resolver, the target node must have the same XmtNodeType
            (context.getTargetParentNode().hasChildNodes(EXAMPLE_NODE_TYPE) == true));
  }

  public XmtNode resolveNode(XmtNodeResolverContext context) {
    XmtNode resolvedNode = null;

    for (Iterator iter =
        context.getTargetParentNode().getChildNodes(EXAMPLE_NODE_TYPE).iterator();
        iter.hasNext(); ) {
      resolvedNode = (XmtNode)iter.next();
```

```
      // Consider source and target node equivalent if they have the same number of
          properties and child nodes
      if ((resolvedNode.getPropertyCount() ==
          context.getSourceNode().getPropertyCount()) &&
          (resolvedNode.getChildNodeCount() ==
              context.getSourceNode().getChildNodeCount())) {
        break;
      }
      else {
        resolvedNode = null;
      }
    }

    return resolvedNode;
  }

  public static void initOpenTool(byte majorVersion,
                                  byte minorVersion) {
    // Only resolve nodes for a specific XmtDomain
    XmtResolverManager.registerNodeResolver(new ExampleXmtNodeResolver(),
        EXAMPLE_DOMAIN);
  }

}
```

Example: XmtNodeValidator

Objective Check that the properties in an XMT model are in a valid state.

```
public class ExampleXmtNodeValidator
    extends XmtAbstractNodeValidatorVisitor {

  private static final XmtNodeType EXAMPLE_NODE_TYPE = new
      XmtNodeType("ExampleNodeType");

  private static final String EXAMPLE_PROPERTY_KEY = "PropertyWhichMustHaveAValue";

  private static final XmtDomain EXAMPLE_DOMAIN = new XmtDomain("ExampleDomain");

  public boolean canValidateNode(XmtNodeValidatorVisitorContext context) {
    // Only validate nodes with a specific XmtNodeType
    return (context.getTargetNode().getType().equals(EXAMPLE_NODE_TYPE) == true);
  }

  public XmtVisitorStatus validateNode(XmtNodeValidatorVisitorContext context) {
    // Check whether the value of the property is valid
    if ((context.getTargetNode().getPropertyValue(EXAMPLE_PROPERTY_KEY) == null) ||
        (context.getTargetNode().getPropertyValue(EXAMPLE_PROPERTY_KEY).equals("") ==
            true)) {
      // If not, report a validation message
      XmtValidatorHelper.reportValidationMessage("Property must have a value",
                                                 XmtMessageSeverity.SEVERITY_ERROR,
                                                 context);
    }

    return XmtVisitorStatus.STATUS_CONTINUE;
  }

  public static void initOpenTool(byte majorVersion,
                                  byte minorVersion) {
    // Only configure nodes for a specific XmtDomain
    XmtVisitorManager.registerNodeVisitor(new ExampleXmtNodeValidator(),
        EXAMPLE_DOMAIN);
  }

}
```

Example: XmtReader

Objective Read an XMT model from a `properties` file.

```
public class ExampleXmtReader
    extends XmtAbstractUrlsReader {

  private static final XmtNodeType EXAMPLE_NODE_TYPE = new
      XmtNodeType("ExampleNodeType");

  private static final XmtDomain EXAMPLE_DOMAIN = new XmtDomain("ExampleDomain");

  public boolean canRead(XmtReaderContext context) {
    // Only read for a specific type of Primetime node
    return (context.getNode() instanceof PropertiesFileNode);
  }

  public List resolveUrls(XmtReaderContext context) {
    return Collections.singletonList(((PropertiesFileNode)context.getNode()).getUrl());
  }

  public XmtNode doRead(XmtReaderContext context)
      throws XmtReaderException {
    XmtNode rootNode;
    Properties props = new Properties();
    Enumeration propNames;
    String propName;

    // Create a root node of the appropriate type
    rootNode = XmtCreatorManager.createNode(context, null, EXAMPLE_NODE_TYPE);

    // Set properties on the root node from the properties file
    try {
      props.load(((PropertiesFileNode)context.getNode()).getInputStream());
    }
    catch (IOException ixcn) {
      XmtHelper.reportProblemMessage(("Unable to load document from file (" +
                                      ((PropertiesFileNode)context.getNode()).getUrl() +
                                      ")"),
                                     XmtMessageSeverity.SEVERITY_ERROR,
                                     context);

      XmtHelper.reportProblemMessage(ixcn.getLocalizedMessage(),
                                     XmtMessageSeverity.SEVERITY_ERROR,
                                     context);

      throw new XmtReaderException("Unable to load document from input stream", ixcn);
    }

    propNames = props.propertyNames();
    while (propNames.hasMoreElements() == true) {
      propName = (String)propNames.nextElement();

      rootNode.setPropertyValue(propName, props.getProperty(propName));
    }

    return rootNode;
  }

  public static void initOpenTool(byte majorVersion,
                                  byte minorVersion) {
    // Only read for a specific XmtDomain
    XmtReaderManager.registerReader(new ExampleXmtReader(), EXAMPLE_DOMAIN);
  }

}
```

Example: XmtTreeStructureContextActionProvider

Objective Provide a new context action for the structure pane of an XMT-based `NodeViewer` which displays the number of properties stored on the `XmtNode` of an `XmtModelNodeTreeStructureNode`.

```
public class ExampleXmtTreeStructureContextActionProvider
    extends XmtAbstractTreeStructureContextActionProvider {

  private static final XmtDomain EXAMPLE_DOMAIN = new XmtDomain("ExampleDomain");

  public boolean canProvideContextAction(XmtTreeStructureContextActionProviderContext
      context) {
    // Only provide context action for a single and specific type of
        XmtTreeStructureNode
    return ((context.getStructureNodes().size() == 1) &&
            (context.getStructureNodes().get(0) instanceof
             XmtModelNodeTreeStructureNode));
  }

  public Action provideContextAction(XmtTreeStructureContextActionProviderContext
      context) {
    return new ExampleXmtTreeStructureContextAction(context);
  }

  private static class ExampleXmtTreeStructureContextAction
      extends XmtAbstractTreeStructureContextAction {

    public
       ExampleXmtTreeStructureContextAction(XmtTreeStructureContextActionProviderContext
       context) {
      super(context, "Show property count");
    }

    public void actionPerformed(Browser browser) {
      XmtModelNodeTreeStructureNode structureNode;

      structureNode =
          (XmtModelNodeTreeStructureNode)getContext().getStructureNodes().get(0);

      JOptionPane.showMessageDialog(browser,
              ("Property count is " + structureNode.getModelNode().getPropertyCount()));
    }

  }

  public static void initOpenTool(byte majorVersion,
                                  byte minorVersion) {
    // Only provide context actions for a specific XmtDomain
    XmtTreeStructureContextActionManager.registerContextActionProvider(new
        ExampleXmtTreeStructureContextActionProvider(), EXAMPLE_DOMAIN);
  }

}
```

Example: XmtTreeStructureNodeProvider

Objective Provide a new root node for the structure pane of an XMT-based `NodeViewer`.

```
public class ExampleXmtTreeStructureNodeProvider
    extends XmtAbstractTreeStructureNodeProvider {

  private static final XmtDomain EXAMPLE_DOMAIN = new XmtDomain("ExampleDomain");

  public boolean canProvideStructureNodes(XmtTreeStructureNodeProviderContext context) {
    // Only provide a node when there is no target parent (e.g. a root structure node)
    return (context.getTargetParentStructureNode() == null);
  }
```

```
  public List provideStructureNodes(XmtTreeStructureNodeProviderContext context) {
    return Collections.singletonList(new XmtDefaultTreeStructureNode("Example Tree
        Structure Node"));
  }

  public static void initOpenTool(byte majorVersion,
                                  byte minorVersion) {
    // Only provide nodes for a specific XmtDomain
    XmtTreeStructureNodeManager.registerStructureNodeProvider(new
        ExampleXmtTreeStructureNodeProvider(), EXAMPLE_DOMAIN);
  }

}
```

Example: XmtWriter

Objective Write a XMT model to a properties file.

```
public class ExampleXmtWriter
    extends XmtAbstractUrlsWriter {

  private static final XmtNodeType EXAMPLE_NODE_TYPE = new
      XmtNodeType("ExampleNodeType");

  private static final XmtDomain EXAMPLE_DOMAIN = new XmtDomain("ExampleDomain");

  public boolean canWrite(XmtWriterContext context) {
           // Only write for a specific type of Primetime node
    return ((context.getNode() instanceof PropertiesFileNode) ||
           // and the root node of the model is of a specific XmtNodeType
           (context.getRootNode().getType().equals(EXAMPLE_NODE_TYPE) == true));
  }

  public List resolveUrls(XmtWriterContext context) {
    return Collections.singletonList(((PropertiesFileNode)context.getNode()).getUrl());
  }

  public BufferUpdater resolveBufferUpdater(XmtWriterContext context,
                                            Url url) {
    return new ExampleBufferUpdater(context);
  }

  public static class ExampleBufferUpdater
      implements BufferUpdater {

    XmtWriterContext context;

    public ExampleBufferUpdater(XmtWriterContext context) {
      super();

      this.context = context;
    }

    public byte[] getBufferContent(Buffer buffer) {
      Properties props = new Properties();
      XmtProperty prop;
      ByteArrayOutputStream bstm;

      for (Iterator iter = context.getRootNode().getProperties().iterator();
          iter.hasNext(); ) {
        prop = (XmtProperty)iter.next();

        props.setProperty(prop.getKey(), String.valueOf(prop.getValue()));
      }

      bstm = new ByteArrayOutputStream();
      try {
        props.store(bstm, null);
      }
```

```
      catch (IOException ixcn) {
        ixcn.printStackTrace();
      }

      return bstm.toByteArray();
    }

  }

  public static void initOpenTool(byte majorVersion,
                                  byte minorVersion) {
    // Only write for a specific XmtDomain
    XmtWriterManager.registerWriter(new ExampleXmtWriter(), EXAMPLE_DOMAIN);
  }

}
```

Chapter 8

Editor concepts

This concept document describes how the editor works in JBuilder. The editor provides a way to view and modify text documents, and it also gives other tools access to the text. Some of the tools interfacing with the editor are CodeInsight and the Structure View. The user communicates through keyboard and mouse input; editor keybindings, as defined in keymaps, translate the keyboard input into meaningful editor actions that manipulate the text in the editor.

See also

- "Structure View concepts" on page 36

Features and subsystems

- Editor manager
- Editor pane
- Editor actions
- Text utilities

Editor manager

There is a single instance of the `EditorManager` class in JBuilder, and its task is to keep track of most of the editor specific details. The `EditorManager` instance is responsible for creating instances of the `EditorPane` class, and it will also fire events to those `EditorPanes` to tell them about global changes that they should act upon, such as changes to the tab size, the font, the keymap, and more. See `EditorManager` class for the complete list of attributes, also called properties, such as `fontAttribute` and `tabSizeAttribute`. You can recognize them because they all end with "Attribute".

The `EditorManager` is also responsible for creating and managing `keyMaps`, and you can read all about that in Chapter 9, "Keymap concepts."

Of course, `EditorPanes` are not the only classes who would want to know about global editor changes, so the `EditorManager` has general support for adding and removing property change listeners. You can either add a class as the listener of a single property change (`EditorManager.addPropertyChangeListener(String, PropertyChangeListener)`), or add a class as the listener of all property changes

(`EditorManager.addPropertyChangeListener(PropertyChangeListener)`). The class that is added as the listener should implement the `java/Beans/PropertyChangeListener` interface, which basically means it should have the function:
`void propertyChange(PropertyChangeEvent evt);`

Let's practice and write code for a class so it will know when the tab size has changed:

```
public class KnowsTabSize implements PropertyChangeListener {

  int tabSize;

  public KnowsTabSize() {
    tabSize = 0;
  }

  public static void initOpenTool(byte majorVersion,
                                  byte minorVersion) {

    // Make sure the OpenTools API is compatible
    if (majorVersion != PrimeTime.CURRENT_MAJOR_VERSION)
      return;

    // Create our class and add it as a listener
    //  of the tabSize EditorManager changes
    KnowsTabSize knowItAll = new KnowsTabSize();
    EditorManager.addPropertyChangeListener(EditorManager.tabSizeAttribute,
                                            knowItAll);
  }

  // Implement to satisfy the PropertyChangeListener interface
  // The EditorManager will call this function anytime it fires a
  //  tab size property change

  public void propertyChange(PropertyChangeEvent e) {
    // Update our own tab size
    tabSize = EditorManager.getTabSize();
  }
}
```

Editor pane

When JBuilder opens a text-based file, such as files with `.java` or `.html` extensions, the browser creates a new node, which adds a new tab to the content pane, and the file is opened in a new editor window. This editor window is of type `EditorPane`, and at any time you can get a handle to the current focused editor window from `EditorAction` as follows:

```
EditorPane editor = EditorAction.getFocusedEditor();
```

Of course, open nodes in the browser that don't have the focus might have an `EditorPane` associated with them, in which case you can use the `getEditor()` function from `EditorManager`:

```
Node[] nodes = Browser.getActiveBrowser().getOpenNodes();
for (int i = 0; i < nodes.length; i++) {
  EditorPane editor = EditorManager.getEditor(nodes[i]);
  if (editor != null) {
    // Do something with the editor
  }
}
```

Many methods that deal with column and caret position have overloaded versions that adjust for expanded tabs.

```
int line = 5;
int column = 15;
int caretIndex;

// Move to the 15th character on line 5
// (assumes tab characters are one column wide)
caretIndex = editor.calcCaretPosition(line, column);
editor.setCaretPosition(caretIndex); // may not move
    //to column 15 if there are tabs on the line

// Move to column 15 on line 5
// (tab characters are expanded and may be several columns wide)
caretIndex = editor.calcCaretPosition(line, column, true);
editor.setCaretPosition(caretIndex);
```

Editor pane document

Each `EditorPane` has its text stored in an `EditorDocument`, and you can get to that document with a call to `EditorPane.getDocument()`. Once you have an `EditorDocument` object, you can get to the actual text of the document. The text is stored in objects of type `javax/swing/text/Element`, and each `Element` object basically represents one line. At the same time, each character of the document has an index, and you can go from an `Element` object to its starting and ending index, and from a character index to the line `Element` that contains that character. Let's try this out with some code:

```
// First get the document
EditorPane editor = EditorAction.getFocusedEditor();
Document doc = editor.getDocument();

// Get the base of all Elements
Element baseElement = doc.getDefaultRootElement();

// Just for fun, how many lines are there in the document
int totalLines = baseElement.getElementCount();

// Where is my caret now
int caretIndex = editor.getCaretPosition();

// On what line is my caret and get the Element of that line
int lineIndex = baseElement.getElementIndex(caretIndex);
Element lineElement = baseElement.getElement(lineIndex);

// Get the boundaries of that line
int startingIndex = lineElement.getStartOffset();
int endingIndex = lineElement.getEndOffset();

//Get the actual text of the line
String lineText = doc.getText(startingIndex, endingIndex - startingIndex);
```

Editor pane caret

Each editor pane has a caret, and that caret will be at a certain index into the document. The caret also holds information about the starting and ending index of a highlight, if there is one. Note that the editor has functions about the selection highlight

as well. Many selection related methods can be found in `EditorCaret`. The following code shows how to use the caret:

```
// Get the editor and the caret
EditorPane editor = EditorAction.getFocusedEditor();
Caret caret = editor.getCaret();

// Where is the caret?
int caretPosition = editor.getCaretPosition();

// Does the caret say there is a highlight?
// Note: this always works for normal (exclusive) selections but
//  sometimes not for block and exclusive selections:
int dot = caret.getDot();   // get the caret position
int mark = caret.getMark(); // get the selection end mark
if (dot != mark) {
  // There probably is a highlighted selection
}

// We cast to EditorCaret to access more methods.
// The caret is normally of the class EditorCaret unless
//  an IME is active (e.g. for Japanese input of Kanji).
if (caret instanceof EditorCaret) {
  EditorCaret ecaret = (EditorCaret) caret;
  // this is a better way to determine if there is a selection
  if (ecaret.isSelected()) {
    // There definitely is a highlighted selection.
    // This catches line and exclusive selections when
    //  the dot and mark are same.
    // This avoids block selections where
    //  the dot and mark are different,
    //  but on the same column of different lines
    //  and there really isn't a selection.

    // Determine the type of selection
    int type = ecaret.getSelectionType();
    switch (type) {
      case EditorCaret.SELECTION_BLOCK:
      case EditorCaret.SELECTION_BLOCK_INC: // used by brief keymap
        // Do something special for block selections
        break;
      case EditorCaret.SELECTION_LINE:
      case EditorCaret.SELECTION_INCLUSIVE: // used by brief keymap
      case EditorCaret.SELECTION_EXCLUSIVE: // normal selection
        // Do something else for other selection types
        break;
      }
    }
    // Get all info about the selection at once
    EditorCaret.Selection sel = ecaret.getSelection();

    // Do some work here that moves the caret,
    //  thus removing the selection.

    // Restore the selection
    ecaret.setSelection(sel);
  }
}
```

```
// Alternate way to find out about selections
if (editor.isSelected()) {

  // Alternate way to get the selection type
  int type = editor.getSelectionType();

  // Alternate way to determine if the selection is a block
  if (editor.isSelectionBlock()) {
  }

  // Get a list of character offsets for all segments of the selection
  // (there will only be one segment unless it is a block selection)
  SelectionOffsets so = editor.getSelectionOffsets();
}
```

Editor code folding

If code folding is enabled and fold block are collapsed, there are methods to determine this and help adjust for the change in screen position caused by the folded blocks. See the API documentation for `primetime.editor.folding.Folder`. The following code shows how to use the fold blocks:

```
// Get the list of fold blocks from the document
FoldBlock[] foldBlocks = ((EditorDocument)getDocument()).getFoldBlocks();
int lineNumber = 5; // some arbitrary document line number

// Determine if a line number is in a collapsed fold block
if (!isInCollapsedFoldBlock(foldBlocks, lineNumber)) {

  // Do action on visible (non-collapsed) line

  // Get the view location for the line
  int lineIndex = Folder.getPhysicalLineNumber(lineNumber, foldBlocks) - 1;
  Point p = new Point(0, lineIndex * editor.getFontHeight());
}

// Determine the document line for the second visible line in the view
if (getParent() instanceof JViewport) {
  JViewport parent = (JViewport)getParent();
  Point p = parent.getViewPosition();
  if (p.y >= 0) {
    // compute a one-based line number and add one more for the second line
    int pos = (p.y / editor.getFontHeight()) + 2;
    // given the physical (screen) line, get the document line
    lineNumber = Folder.getLogicalLineNumber(pos, foldBlocks);
  }
}
```

Editor actions

There is a whole set of actions available that target the editor. These actions all live in `EditorActions`. One of the best ways to learn about these actions is to study the Keymap examples in the OpenTools samples. There are samples that show how to create a CUA keybinding, an Emacs keybinding, and a Brief keybinding. The keymaps map key strokes to actions that often are one of the actions defined in `EditorActions`.

When a new action is written, it is important that it's hooked up to a keystroke. In addition, the action should register itself so it will be displayed in the keymap editor. There are several steps involved in this process, and these steps can be studied in the

samples/OpenToolsAPI/LineCommentHandler example which shows how to create a new action and register it.

Here is some of that code:

```
public static void initOpenTool(byte majorVersion, byte minorVersion) {
  // Register our LineCommentHandler action so it will show up
  // in the keymap editor.
  // We register it as an Editor action because this action only
  // makes sense if the focus is in the editor.
  EditorActions.addBindableEditorAction(ACTION_LineCommentHandler);
}

public static EditorAction ACTION_LineCommentHandler =
  // The "line_comment_handler" name is used in the
  //  keymap editor to name our action.
  new LineCommentHandlerAction("line_comment_handler");

public LineCommentHandlerAction(String nm) {
  super(nm);
  // The long description property is used in the keymap editor
  // to fill up the description memo.
  this.putValue(Action.LONG_DESCRIPTION,
  "Adds //DBG at the start of the line if it's not already there, otherwise
        //deletes it."
  );
  // The ActionGroup property is used to put this action
  // in the specified group of actions.
  this.putValue("ActionGroup", "Miscellaneous");
}
```

Text utilities

Another useful class to look at is the TextUtilities class. Here you can find functions that will go from a document index to a line, or will find the beginning and end of words. If the text has hardcoded tab characters, there are some functions that will compute the width of a text, or determine the index of a character in a text, taking tabs into account.

Chapter 9

Keymap concepts

This concept document describes how keymaps work in JBuilder, and especially how keymaps can be created, changed and extended.

A keymap provides a structured way to bind keystrokes to actions; it allows a keystroke to be registered with a corresponding action, and the keymap will ensure that the action will get fired when the user enters the keystroke. At least, that will happen when the keymap is the currently registered keymap in the component that has focus.

There are some JDK classes that should be studied to understand how keymaps are implemented in Swing:

- The Keymap class is located in `javax/Swing/text/Keymap.java`
- The KeyStroke class is located in `javax/Swing/KeyStroke.java`
- The Action class is located in `javax/Swing/Action.java`
- The Event class is located in `java/awt/Event.java`

JBuilder installs several keymaps, such as CUA, Emacs, and Brief. To change to another keymap, choose Tools|Preferences on the menu and select Keymaps. From there, you can edit keymaps to add and remove keybindings as well as create entirely new keymaps. This method of keymap modification is preferable to the solutions listed below.

Changing and extending keymaps

Basic keymap information

The `EditorManager` class does most basic keymap handling. The current keymap is returned by the `getKeymap()` function, and the name of the current keymap by the `getKeymapName()` method.

An instance of a specific keymap can be retrieved by name with the `getKeymap(String)` method, using as argument, for instance, `CUA` or `Emacs`. A new keymap can be installed, either by name or with a keymap instance, using the methods `setKeymapName(String)` and `setKeymap(Keymap)`.

Notification of keymap changes

The `EditorManager` class will also fire a change event when a new keymap is installed as the current keymap. The code to install a class as a change event listener of the `EditorManager` class is as follows:

```
EditorManager.addPropertyChangeListener(myClass);
```

Where `myClass` is an instance of a class that implements the `java.beans.PropertyChangeListener` interface, meaning the `propertyChange()` method should be implemented. Here is a simplified code sequence:

```
public class ModifyKeyBinding implements PropertyChangeListener {

  public ModifyKeyBinding() {
  }

  public static void initOpenTool(byte majorVersion, byte minorVersion) {
    // Make sure the OpenTools API is compatible
        if (majorVersion != PrimeTime.CURRENT_MAJOR_VERSION)
          return;
    // Create our class and add it as a listener of any EditorManager changes
    // This will allow us to catch keymap changes
    ModifyKeyBinding m = new ModifyKeyBinding();
    EditorManager.addPropertyChangeListener(m);
  }

  // The EditorManager will call this function anytime it fires a
  // property change
  public void propertyChange(PropertyChangeEvent e) {
    String propertyName = e.getPropertyName();

    // We are only interested in keymap changes
    if (propertyName.equals(EditorManager.keymapAttribute)) {

      // We need a keymap to change
      Keymap keymap = EditorManager.getKeymap();
      if (keymap == null)
        return;

      // Change the keymap......
  }
}
```

Changing the keymap

When a new keymap is installed, it's easy to insert new keybindings, or modify existing ones. Below is a code example that changes one of the keybindings:

```
keymap.addActionForKeyStroke(KeyStroke.getKeyStroke(KeyEvent.VK_0,
    Event.CTRL_MASK|Event.SHIFT_MASK),
                              EditorActions.ACTION_MatchBrace);
```

Now is the time to study the `Keymap`, `KeyStroke`, `KeyEvent`, and `Event` classes in the JDK. What we are doing here is binding the *Ctrll+Shift+0* (zero) key event (on English keyboards this will be *Ctrl+)*) to the editor action that will find the matching brace when the cursor is positioned at a brace (see the `EditorActions.MatchBraceAction` class).

In the `EditorActions` class is a list of static instances of all the actions the JBuilder editor supports. The names of all those actions start with `ACTION_`. For instance, `ACTION_Backward`.

In the `KeyEvent` class of the JDK, you will find the keycodes for all recognized keys. These keycodes all start with `VK_`, and most of them are easy to remember. The `Event` class of the JDK lists all the modifiers of a key event, such as *Control*, *Alt*, `Meta`, and *Shift*.

We know enough now to tie any key combination to any `EditorActions` action.

Managing keystrokes with multiple key events

The JDK fires multiple key events for each keystroke. Normally this is of no concern when you modify key bindings with `Keymap.addActionForKeyStroke`, because things will work fine. If, however, you try to change the binding of a keystroke with no modifiers, you might run into trouble. Say you want to tie the *Y* key to the "cut" action, so you have this statement:

```
keymap.addActionForKeyStroke(KeyStroke.getKeyStroke(KeyEvent.VK_Y, 0),
                             EditorActions.ACTION_Cut);
```

When you run your program, you'll find that the "cut" action is indeed executed, but you also still get the *Y* character added to your document, which is not what you wanted. This is caused by the fact that JDK fires three events for each key pressed, a "pressed" event, a "typed" event, and a "released" event. When you use any of the `VK_` keycodes in your `addActionForKeyStroke()` method call, you are hooking into the "pressed" event, and you are overriding the default action for the "pressed" event. The "typed" event only affects keystrokes that translate into printable characters, and the default action for "typed" events is to emit the character into the document. The "released" event is normally of no importance.

Why can we add actions for most keystrokes, and things work fine? It's because we normally want to add an action to a keystroke with the *Alt*, *Ctrl*, or `Meta` modifier, and those key events are typically not printable, so they don't generate a "typed" event. We simply override the "pressed" event, and we're done. For printable keystrokes, things work better if we hook into the "typed" event with code like:

```
keymap.addActionForKeyStroke(KeyStroke.getKeyStroke('y'),
                             EditorActions.ACTION_Cut);
```

The code above should still call the "cut" action, but not emit the *Y* character into the document.

Writing your own action

The editor actions JBuilder supplies are, of course, not always enough. It's easy to write your own action class and bind it to any key event.

First of all, each new `Action` class should be derived from the appropriate `Action` class. There is the `EditorAction` class, which can function as the parent class of most new actions. There are also the `BriefAction` class and the `EmacsAction` class, and you would derive from one of these two classes if you are writing a Brief or Emacs specific action.

This is what a basic action class should look like:

```
class MyActions {
  public static class DoSomethingAction extends EditorAction {

    // Every action should be initialized with an appropriate name
    public DoSomethingAction(String nm) {
      super(nm);
    }
```

```
    // This is called when the JDK fires a key event
    public void actionPerformed(ActionEvent e) {
      // Action specific code, for instance:
      EditorPane target = getEditorTarget(e);
      if (target != null) {
        // Do something with the editor pane
      }
    }

  }

  // Add a static instance so anybody can use it
  public static EditorAction ACTION_DoSomething =
    new DoSomethingAction("do_something");
}
```

And now you can put it to use with the following code:

```
keymap.addActionForKeyStroke(KeyStroke.getKeyStroke(KeyEvent.VK_Y,
    Event.CTRL_MASK),MyActions.ACTION_DoSomething);
```

Making the keymap editor recognize your new action

If you want the keymap editor to add your new action to the list of actions that can be bound to keystrokes, you need to add code similar to one of the following sequences:

```
public static void initOpenTool(byte majorVersion, byte minorVersion) {
    // Register our action so it will show up in the keymap editor.
    // We can register it as an Editor action, which is an action that
    // makes sense if the focus is in the editor, or as an IDE action
    // which makes sense no matter where the focus is.
    //
    // This is how to add an action as an Editor action
    EditorActions.addBindableEditorAction(ACTION_DoSomething);

    // If you want to add your action as an IDE action, you would
    // use the following code:
    //
    //  EditorActions.addBindableIDEAction(ACTION_DoSomething);

    // If you want to add an action as an Editor action, but have
    // it only show up in a particular keymap, use the following
    // code which makes the action only show up in the Brief keymap
    //
    // EditorActions.addBindableEditorAction(ACTION_DoSomething, "Brief");

    // Similar if it is an IDE action for the CUA keymap
    //
    //  EditorActions.addBindableIDEAction(ACTION_DoSomething, "CUA");
}

public static class DoSomethingAction extends EditorAction {

  // Every action should be initialized with an appropriate name
  public DoSomethingAction(String nm) {
    super(nm);
```

```
    // The long description property is used in the keymap editor
    // to fill up the description memo.
    this.putValue(Action.LONG_DESCRIPTION,
                  "This is a function that does something.");
    // The ActionGroup property is used to put this action
    // in the specified group of actions.
    this.putValue("ActionGroup", "Miscellaneous");
  }

  // This is called when the JDK fires a key event
  public void actionPerformed(ActionEvent e) {
    // Action specific code, for instance:
    EditorPane target = getEditorTarget(e);
    if (target != null) {
      // Do something with the editor pane
    }
  }

}

// Add a static instance so anybody can use it
public static EditorAction ACTION_DoSomething =
  new DoSomethingAction("do_something");
```

Using advanced keymap functions

The JDK Keymap class has some more interesting functions:

- `public Action getAction(KeyStroke key)`

  This is useful if you want to save the current `Action` bound to a keystroke before you bind a new `Action` with `addActionForKeyStroke()`. Later on you can restore the original action.

- `public Action getDefaultAction()` and `public void setDefaultAction(Action a)`

  The default action is called when there is no binding for the current keystroke. Handy if you want to handle most keystrokes in one event.

- `public void removeKeyStrokeBinding(KeyStroke keys)`

  Use this if you want to remove a binding you added through addActionForKeyStroke.

- `public void removeBindings()`

  Gets rid of all the bindings in the current keymap. Handy when you switch keymaps.

Removing key bindings

As we saw previously, key bindings can be removed from a keymap, either individually or *en masse*. Removing a keybinding does not mean, though, that a key event doesn't have an action associated with it anymore. It only means that there's no action associated with the key in the current keymap. Keymaps are stacked on top of each other, and the JDK travels the stack of keymaps trying to find an action that is bound to a certain keystroke. For instance, by default the JDK might bind an action to the *Page Up* key, and JBuilder overrides this with the `PageUpAction`. If JBuilder removes that binding to `PageUpAction`, the *Page Up* key will trigger the default JDK action again.

In case you really want to “delete” a keybinding, meaning you want the key not to fire any event, you might tie it to a `doNothing` action, such as:

```
public static class DoNothingAction extends EditorAction {

  // Every action should be initialized with an appropriate name
  public DoNothingAction(String nm) {
    super(nm);
  }

  // This is called when the JDK fires a key event
  public void actionPerformed(ActionEvent e) {
  }

}
```

Creating your own keymap

Starting with JBuilder 2005, keymaps are stored in XML format and discovered dynamically at startup. Previously written OpenTools which add new keymaps will still work, but it’s now much simpler to add new keymaps via the Tools|Preferences| Keymaps dialog. The New button will create an entirely fresh keymap which only includes keybindings that are common to all of the default keymaps. If you wish to create a new keymap that contains many of the same keybindings as another keymap, select the keymap and use the Copy button.

Using improved JDK 1.3 keymaps

Keymaps changed in JDK 1.3. The old scheme consisted of keymaps that contain the bindings between keystrokes and actions. The new scheme has the keymap split into two parts: an input map that holds the keystrokes, and an action map that contains the actions. Note, however, that most keymap behavior won’t change for developers, since they will wrap the old behavior around the new behavior. If you want, you can go one level deeper and start using the new input and action maps. For a detailed description, see the official JavaSoft `kestrel/keybindings` report.

Chapter 10

Wizard concepts

A wizard is a standardized modal dialog box that has one or more panels that walk a user through an otherwise complicated or tedious task.

Wizards usually appear in the IDE either in the object gallery (for those wizards that construct source code for objects) or under the Edit|Wizards menu (also available via the *Ctrl+Shift+W* keyboard shortcut) on the menu bar. Other possible locations (such as on the toolbar and on pop-up context menus) require different registration methods. To display the object gallery, choose File|New.

To write a new wizard, generally you perform these steps:

1. Register a `WizardAction` with the `WizardManager`.
2. Override methods in the `BasicWizard` class.
3. Provide one or more `BasicWizardPage` panel objects used to gather user input.

You can optionally also provide a `SummaryPage`. This is intended to provide feedback to the user about what was accomplished and perhaps could be used to configure something to be done either in the page `summaryAcknowledged()` method or else when your wizard `wizardCompleted()` method is called immediately after.

Like other OpenTools, wizards are introduced at application startup during the OpenTools discovery process. Each wizard provides a static `initOpenTools()` method that registers its static `WizardAction` method with the `WizardManager`.

A `WizardAction` provides information that helps integrate that wizard with the application and acts as a factory when an instance of that wizard is needed.

Note The entire IDE is based on the Swing component architecture. All wizard UI components must be Swing-based.

Features and subsystems

- OpenTools registration
- Wizard flow control
- Wizard steps
- Advanced features
- Testing your new wizard

OpenTools registration

The new wizard must provide the standard interface looked for by the OpenTools discovery process and use it to register its `WizardAction`. The most common way to integrate a wizard is to register it with the `WizardManager` in the `initOpenTool()` method. Here is an example:

```
public static void initOpenTool(byte majorVersion, byte minorVersion) {
  if (majorVersion == PrimeTime.CURRENT_MAJOR_VERSION) {
    WizardManager.registerWizardAction(myWizard);
  }
}
```

A `WizardAction` is essentially a way of describing the wizard. Use one of its several constructors which provide ever increasing overrides of default behavior. This example uses the constructor that takes six parameters: `shortText`, `mnemonic`, `longText`, `smallIcon`, `largeIcon`, and `galleryWizard`:

```
public static final WizardAction myWizard = new WizardAction (
  "My Wizard...", 'm', "My wizard description",
  BrowserIcons.ICON_BLANK,
  BrowserIcons.ICON_BLANK,
  false) {

  protected Wizard createWizard() {
    return new MyWizard();
  }
};
```

Because the final `galleryWizard` parameter is false, this wizard doesn't appear in the object gallery, but on the Edit|Wizards menu. If the wizard appears in the object gallery, the specified large icon is used and the `longText` parameter value is the description that appears as the wizards tool tip. If the wizard appears on the Edit|Wizards menu, the small icon is used and the description appears on the status bar when the mouse is over the wizard menu item.

The `category` parameter is used to specify the name of an object gallery page where the wizard is to appear. If the name does not already exist, a new object gallery page will be created. To specify a subpage, use semi-colons to separate the page names. For instance, `Enterprise;CORBA` to appear on the page where the CORBA wizards are presented.

If you want your wizard to appear both in the object gallery and in the Edit|Wizards menu, you could implement your two actions as subclasses of a class similar to this:

```
class MyBaseWizardAction extends WizardAction {
  public MyBaseWizardAction(
    String shortName, char mnemonic, boolean galleryWizard, String category) {

    super(shortName,
          mnemonic,
          "My descriptive text",
          Icons.getIcon(MyWizard.class, "image/icon16x16.gif"),
          Icons.getIcon(MyWizard.class, "image/icon32x32.gif"),
          galleryWizard,
          category);
  }

  protected Wizard createWizard() {
    return new MyWizard();
  }
};
```

Wizard flow control

The `BasicWizard` class controls the flow of the wizard by defining which pages will appear and in what order. When you extend this class, you must override the `invokeWizard()` method to call `setWizardTitle()`, and then call `addWizardPage()` for each `BasicWizardPage` you defined. Here's an example:

```
MyWizardPage1 page1 = new MyWizardPage1();
public WizardPage invokeWizard(WizardHost host) {
  setWizardTitle("My Wizard");
  addWizardPage(page1);
  return super.invokeWizard(host);
}
```

The `BasicWizard` class also supplies the `finish()` method that is called when the user clicks the OK or Finish button in the wizard, indicating the wizard should perform its assigned tasks. In this example, you would put the primary logic of the wizard within the `doIt()` method:

```
protected void finish() throws VetoException {
  doIt();
}
```

If you encounter an error during `finish()`, at the minimum you will want to throw a `VetoException`. This will not close the wizard allowing the user to try again. You might also want on error to provide feedback to the user as to the problem.

Returning from `finish()` without throwing an exception will normally cause the dialog to be closed and a call to the `wizardCompleted()` method.

An alternative way to terminate a wizard is to provide `BasicWizard` a `SummaryPage` instance using the provided setter method. A normal return from `finish()` will then by default cause that summary page to be displayed in the dialog with just an "OK" button to acknowledge it. The `SummaryPage` does not have to be a step already on the wizard page list.

An alternative to the above in handling the Finish button is to override the `finish(WizardPage currentPage, WizardHost host)` method which calls `checkPage()` for the current step, calls `finish()`, and then returns any `SummaryPage` which has been set. If this method returns any `WizardPage`, then the wizard dialog will not close and it will make that returned page the active step. This is intended to return the wizard back to a step where an error can be corrected. Thanks to the `checkPage()` method on each step, this situation probably should never come up because it can be detected earlier.

Wizard steps

Each `BasicWizardPage` class provides the UI for a single panel (sometimes referred to as a step) of the wizard.

There aren't any methods that you absolutely must override in the class. Usually the content consists of Swing components that form the user interface panel.

If the default large icon on the left of the page is in the way, you can switch to a smaller icon by calling the `setPageStyle(STYLE_COMPLEX)` method. Depending on which page style you choose and if you want to supply a different icon than the default, you can also call `setSmallIcon()` or `setLargeIcon()`.

You might find the `activated()` method of the BasicWizardPage useful. It provides a way of initializing fields based on the input from prior pages.

If you want the wizard to validate the user input before the user advances to the next page or clicks the OK or Finish button, you can override the `checkPage()` method,

supply the validation logic, and then throw a `VetoException` if any validation errors exist. Here's an example:

```
public void checkPage() throws VetoException {
  if (checkForError()) {
    JOptionPane.showMessageDialog(wizardHost.getDialogParent(),
                                  "We have a problem.",
                                  "Error",
                                  JOptionPane.ERROR_MESSAGE,
                                  null);
    throw new VetoException();
  }
}
```

Sometimes you add listeners in the `activated()` method and want to remove them when they are no longer needed. This may require placing the remove listener logic in both the `deactivated()` and `checkPage()` methods. This is necessary with a multiple step wizard because `checkPage()` will not be called on the Back button, but `deactivated()` will be.

Advanced features

Enabling your WizardAction

A `WizardAction` can optionally enable itself depending on the current state of the environment. You can do this by overriding the `update()` method and using the supplied context to determine if conditions are suited for the wizard. For instance, you could have `update()` check if there is a project open, test the file type of the active node if there is one, and/or check whether a needed class can be found on the current classpath.

```
public static final WizardAction myWizard = new WizardAction (
  "My Wizard...", 'm', "My wizard description",
  BrowserIcons.ICON_BLANK,
  BrowserIcons.ICON_BLANK,
  false) {

  public void update(Object source) {
    Browser browser = Browser.findBrowser(source);
    Node node = browser.getActiveNode();
    setEnabled(node instanceof JavaFileNode);
  }
  protected Wizard createWizard() {
    return new MyWizard();
  }
};
```

Dynamically changing wizard pages

You can have a `BasicWizard` dynamically change the flow of the wizard pages by overriding the `checkPage()` method. Usually `checkPage()` is called on the currently visible `BasicWizardPage` to validate its user input. Instead, for example, you can include code that, based on the input on the first page, dynamically alters the flow for the rest of the wizard. (Do not remove or add pages before the current page since internally an index to the current page is being kept!) Here's an example:

```
protected void checkPage(WizardPage page) throws VetoException {
  super.checkPage(page);

  if (page == step1) {
    removeWizardPage(step2);
    removeWizardPage(step3);
```

```
      removeWizardPage(step4);
      if (step1.getChoice()) {
        addWizardPage(step2);
        addWizardPage(step4);
      }
      else {
        addWizardPage(step3);
      }
    }
  }
```

Providing error feedback

Occasionally you might want to have the wizard validate input entered into a text field and dynamically alter the state of the OK/Finish or Next button accordingly as the user is typing.

One approach do that is have your implemented `BasicWizardPage` class extend `javax.swing.event.DocumentListener` and add itself as a listener for that field:

```
jTextField1.getDocument().addDocumentListener(this);
```

Then the implementation of the interface would be similar to this:

```
public void insertUpdate(DocumentEvent e) {update();}
public void removeUpdate(DocumentEvent e) {update();}
public void changedUpdate(DocumentEvent e) {update();}

private void update() {
  String text = jTextField1.getText().trim();
  if (wizardHost ! = null){
    boolean valid = (text.length() > 0);
    wizardHost.setNextEnabled(valid);
    wizardHost.setFinishEnabled(valid);
  }
}
```

The disadvantage of this approach is that the error feedback is a bit subtle. Many people might not realize at what point the error condition was triggered, therefore it might need to be accompanied by some other error indication.

Maintaining state information

In a multiple step wizard, the user may use the Back button to revisit a prior step. It is also possible that you might want to enable the OK/Finish button before the last step, thereby allowing the user to accept all the defaults of steps not visited. It then makes sense to architect such a wizard with a class that initializes all the default settings. A reference to that class would be passed when creating each wizard page, and each would initialize by accessing that class when activated and update it when deactivated. The actual logic invoked by Finish could reside in that class as well.

Using a Personality

It is possible to associate your wizard with an existing `Personality` implementation. All wizards are normally part of a global default personality and so are not affected by a configuration change.

This feature causes your wizard to either appear or disappear automatically depending on the project settings. For instance, you could specify that your wizard is associated with the Team Development feature. If the user disables this in Project|Project Properties|Personality, then your wizard (and others which are associated only with that personality) would not appear the next time the object gallery or Edit|Wizards menu is displayed.

Override this `WizardAction` superclass `UpdateAction` method to make an association with one or more existing personalities:

```
public Personality[] getPersonalities() {
  return com.borland.primetime.personality.TeamdevPersonality.teamdevPersonalities;
}
```

Testing your new wizard

The heart of each `BasicWizardPage` implementation is the user interface it presents. To ease future localization, you should make the layout as flexible as possible to accommodate labels and text that might greatly decrease or increase in width once translated. When the wizard frame is stretched larger, the layout for each page should be designed to stretch with it so the user can adjust for any components that might have been clipped despite your best efforts. When there is extra vertical space, components should move upwards rather than centering, or have the extra vertical space go controls which can make use of it.

You should assign keyboard accelerators to all input fields. Avoid using the same accelerator keys as are used by the Back, Next, and Finish buttons if you are implementing a multi-page wizard.

Check the order in which focus moves between components with the Tab key to ensure movement is in the order the user of that page would expect.

You might want to add tooltips to any controls where the user could get confused about what is intended.

Using Tools|Preferences|Browser, switch Look And Feel settings and verify that the new wizard still has an acceptable layout and colorization in each.

Check that your UI does not require a screen resolution beyond 1024 x 768 pixels to be usable. In particular, try controls such as JComboBox with very long strings to ensure that a wizard page does not resize itself to fill the screen to accommodate the increased “preferred” size of a control.

Compare your wizard to the native application wizards and attempt to present a similar user interface. For instance, JBuilder wizards capitalize only the first word in a label and end the text with a colon. When you put a group box around controls, use the following to make the box appear similar to the native wizards:

```
JPanel myGroup = new JPanel();

myGroup.setBorder(com.borland.primetime.ui.Util.createDefaultTitledBorder("My
    group:"));
```

The “browse” button with the ellipsis for native wizards can be created this way:

```
JButton myButton = com.borland.primetime.ui.Util.createBrowseButton();
```

Depending on your wizard, you may be able to do some useful early testing without having to run as part of JBuilder by adding the following code block to your class. You will be able then to right-click on your file node in either the project pane or on its content pane tab, and select the “Run” option.

```
public static void main(String[] args) {
  try {
    //
UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
    UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
    //
UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
  }
  catch (Exception ex) {}

  WizardDialog dlg = new WizardDialog(null, new MyWizard());
  dlg.show();
```

Chapter 11

UI package concepts

The com.borland.primetime.ui package contains many support classes that not only can be useful in building your user interfaces, but also help you to duplicate the look and feel of JBuilder within your own wizards. The majority of these classes are public, yet are not fully documented as part of the OpenTools API. The classes are all Swing-based.

Features and subsystems

The classes discussed in this document are:

- ButtonStrip
- ButtonStripConstrained
- CheckTree
- CheckTreeNode
- ColorCombo
- ColorPanel
- CompositeIcon
- DefaultDialog
- DialogValidator
- EdgeBorder
- ImageListIcon
- LazyTreeNode
- ListPanel
- MergeTreeNode
- MessageLabel
- SearchTree
- VerticalFlowLayout

ButtonStrip and ButtonStripConstrained

ButtonStrip extends JPanel and uses a FlowLayout to display a single row of buttons either vertically or horizontally. The default layout is horizontal with right-alignment and a horizontal/vertical gap of 5 pixels, but this may be changed by constructor

parameters or by using either the `setAlignment()` or `setOrientation()` methods after construction.

```
ButtonStrip buttonStrip = new ButtonStrip(FlowLayout.RIGHT, 5);
buttonStrip.add(okButton);
buttonStrip.add(cancelButton);
```

The one drawback of `ButtonStrip` is that its `FlowLayout` will not display one or more buttons when there is insufficient space to draw them completely. For that reason, there is the alternative `ButtonStripConstrained` class which uses `GridBagLayout` (although it only supports a horizontal layout with right-alignment of the components at this time). This layout makes more efficient use of the available space. It will clip buttons that do not fit completely.

```
ButtonStripConstrained buttonStrip = new ButtonStripConstrained();
buttonStrip.add(okButton);
buttonStrip.add(cancelButton);
```

CheckTree and CheckTreeNode

These classes (along with support classes `CheckTreeCellEditor`, `CheckTreeCellRenderer`, `CheckCellRenderer`, and `CheckBoxCellRenderer`) provide an implementation of a `JTree` where each `CheckTreeNode` includes a checkbox. An `ItemListener` event is fired when a checkbox is checked or unchecked by the user.

```
public class MyPropertyPanel extends JPanel  {
  public class OptionTreeNode extends CheckTreeNode {
    public OptionTreeNode(String label, boolean isCheckable, OptionTreeNode
        parent) {
      super(label, isCheckable);
      setText(label);
      if (parent != null) {
        parent.add(this);
      }
    }
  }
  JTree optionTree = new CheckTree();
  OptionTreeNode root = new OptionTreeNode("Root", false, null);
  OptionTreeNode optionsGroup = new OptionTreeNode("Options", false, root);
  OptionTreeNode showErrorsOption = new OptionTreeNode("Show error messages",
      true, optionsGroup);
  DefaultTreeModel model = new DefaultTreeModel(root);

  public MyPropertyPanel() {
    try  {
      jbInit();
    }
    catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  public void writeProperties() {
    optionTree.stopEditing();
    System.out.println("Show error messages " + showErrorsOption.isChecked());
  }

  public void readProperties() {
    optionTree.stopEditing();
    showErrorsOption.setChecked(false);
  }
```

```
  void jbInit() throws Exception {
    // build UI

    optionTree.setModel(model);
  }
}
```

ColorCombo and ColorPanel

`ColorPanel` extends `JPanel`. It displays sixteen colors in a grid on the left, and eight custom colors (optionally passed as parameters) in a grid to the right. The number of displayed rows defaults to 2.

`ColorCombo` extends `JComboBox`. It displays the currently selected color in its "edit" field and displays a `ColorPanel` as its dropdown when clicked. The control fires `ActionListener` events when a value is selected. It does not fire `ItemListener` events.

```
ColorCombo cc = new ColorCombo(null, 2, ColorCombo.RIGHT);
cc.setSelectedColor(Color.red);
cc.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    Color color = cc.getSelectedColor();
  }
});
```

CompositeIcon

`CompositeIcon` takes an array of icons and displays them in a single row. The row height will be that of the tallest icon in the array and the icons will be centered vertically in the row.

DefaultDialog and DialogValidator

`DefaultDialog` extends `JDialog`. It provides some convenient mechanisms to support your dialog design. These include auto-centering, frame finding, default button handling, and minimum size enforcement. When the dialog is closed via the OK button and you had provided a component (optional) using the `DialogValidator` interface in your `showModalDialog` call, the `DefaultDialog` will invoke that interface, permitting it to decide if the dialog can be closed.

`DefaultDialog` can provide a shell to display your panel:

```
MyPanel panel = new MyPanel();
if (DefaultDialog.showModalDialog(Browser.getActiveBrowser(), "My Title",
    panel, null, null)) {
  System.out.println("pressed OK");
}
```

Or you can extend it for more convenient access to its features:

```
MyDialog dlg = new MyDialog(Browser.getActiveBrowser(), "My Title", true);
dlg.show();

public class MyDialog extends DefaultDialog {
  public SelectWindowDialog(Component owner, String title, boolean modal) {
    super(owner, title, modal);
    setAutoCenter(true);
  }
}
```

EdgeBorder

This class implements the `Border` interface, providing a two-pixel inset where three-dimensional shadow and highlight effects are drawn.

```
statusPanel.setBorder(new EdgeBorder(EdgeBorder.EDGE_TOP));
```

ImageListIcon

`ImageListIcon` allows you to extract a single `Icon` from a horizontal strip. Each bitmap is square and each has the same dimensions.

```
public static Image IMAGE_ACTIONS = ImageLoader.loadFromResource("icons16x16.gif",
    BrowserIcons.class);
public static ImageListIcon ICON_CUT   = new ImageListIcon(IMAGE_ACTIONS, 16, 0);
public static ImageListIcon ICON_COPY  = new ImageListIcon(IMAGE_ACTIONS, 16, 1);
public static ImageListIcon ICON_PASTE = new ImageListIcon(IMAGE_ACTIONS, 16, 2);
```

LazyTreeNode

This class extends `javax.swing.tree.TreeNode`. You should use `LazyTreeNode` if, for performance or other reasons, you only want to provide access to children when the user asks for them. When that happens, your implementation of the `createNodes()` method will be called to produce them.

```
setRoot(new LazyTreeNode() {
  public TreeNode[] createNodes() {
    if (myList == null) {
      return EMPTY_NODES;
    }

    TreeNode[] nodes = new TreeNode[myList.size()];
    for (int index = 0; index < nodes.length; index++) {
      nodes[index] = (TreeNode)myList.get(index);
    }
    return nodes;
  }
});
```

ListPanel

This is an abstract class and extends `JPanel`. It provides a scrolling `JList` next to five buttons which are, by default, labeled “Add...”, “Edit...”, “Remove”, “Move Up”, and “Move Down”. The first three labels can be changed by using the appropriate constructor. If you override the `getListCellRendererComponent()` method, you can do custom rendering of the list elements, such as supplying a different color for particular elements.

```
public class MyPanel extends ListPanel {

  public MyPanel() {
  }

  public Component getListCellRendererComponent(JList list, Object value,
      int index, boolean isSelected, boolean cellHasFocus) {
    defaultListCellRenderer.getListCellRendererComponent(list,
        getElementName(value), index, isSelected, cellHasFocus);
    if (value instanceof JPanel) {
      defaultListCellRenderer.setForeground(Color.gray);
    }
    return defaultListCellRenderer;
  }
```

```
  // Called on Add...
  protected Object promptForElement() {
    return null;
  }

  // Called on Edit...
  protected Object editElement(Object listElement) {
    return null;
  }
}
```

MergeTreeNode

This class extends `javax.swing.tree.DefaultMutableTreeNode`. Use `MergeTreeNode` if you have a `JTree` that you need to update while still preserving any node expansions. Typically, this is used in implementations providing Structure Pane content. You would first build a second instance of the tree and then invoke the `mergeChildren()` method to update the original tree.

```
public class MyTextStructure extends TextStructure {

  public MyTextStructure() {
    treeModel.setRoot(new MyMergeTreeNode(null));
  }

  class MyMergeTreeNode extends MergeTreeNode {
    public MyMergeTreeNode(Object userObject) {
      super(userObject);
    }

    public void sortChildren() {
      MergeTreeNode[] array = getChildrenArray();
      if (array == null)
        return;
      Arrays.sort(array, new Comparator() {
        public int compare(Object o1, Object o2) {
          // Do comparison here between two tree nodes
          return 0;
        }
      });
      children = new Vector(Arrays.asList(array));
      sortDescendants();
    }

    public void sortDescendants() {
      if (children != null) {
        Enumeration e = children.elements();
        while (e.hasMoreElements()) {
          ((MyMergeTreeNode)e.nextElement()).sortChildren();
        }
      }
    }
  }

  public void updateStructure(Document doc) {

    final MyMergeTreeNode newRoot = new MyMergeTreeNode(null);
```

```
      try {

        // Build a new structure tree using newRoot here

        // Prepare an object that updates the model
        Runnable update =
          new Runnable() {
            public void run() {

              MyMergeTreeNode root = (MyMergeTreeNode)treeModel.getRoot();

              // Merge the new model into the old so that expansion paths can be
              // preserved
              root.mergeChildren(newRoot);

              // Sort everything including the top level of the structure tree
              root.sortChildren();

              // Update the display
              treeModel.nodeStructureChanged(root);
            }
          };

        // Update the model on the main swing thread...
        if (SwingUtilities.isEventDispatchThread())
          update.run();
        else
          SwingUtilities.invokeLater(update);

      }
      catch (java.io.IOException ex) {
      }
    }
  }
```

MessageLabel

`MessageLabel` is a component that derives from `JTextArea`, yet behaves like a multiline label. Use in places where a label that wraps text is desired. Create the label before setting its text. Below are two examples of ways to customize the `MessageLabel`.

To specify a label of a fixed pixel width, use the constructor:

```
MessageLabel(int fontStyle, int maxWidth)
```

To specify a label that wraps at wordbreaks and has a specific `fontStyle` (such as `Font.PLAIN`), use the constructor:

```
MessageLabel(boolean lineWrap, int fontStyle)
```

SearchTree

`SearchTree` extends `JTree`. When you enter a number, letter, or one of the supported regular expression characters, it displays a `JTextField` with the text “Search for:” followed by the input. With each character entered, it attempts to match the entire string with a visible node, causing the first matched node to be selected. If the selected node has children, entering a period while depressing *Ctrl* causes the node to expand. The *Enter* or *Esc* key cancels the window. In order for this to work, each `TreeNode` must supply the same text in its `toString()` method as it displays in the tree.

```
public class MyPanel extends JPanel {
  BorderLayout layout = new BorderLayout();
  JScrollPane scroller = new JScrollPane();
  SearchTree tree = new SearchTree() {
    public void updateUI() {
      super.updateUI();
      unregisterKeyboardAction(KeyStroke.getKeyStroke(KeyEvent.VK_A,
          Event.CTRL_MASK));
    }
  };

  public MyPanel() {
    try {
      jbInit();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  private void jbInit() throws Exception {
    this.setLayout(layout);
    this.add(scroller, BorderLayout.CENTER);
    tree.setPreserveExpansion(true);
    scroller.getViewport().add(tree, null);
  }
}
```

VerticalFlowLayout

This class extends and mimics `FlowLayout`, but does its layout vertically rather than horizontally.

```
int hGap = 5;
int vGap = 5;
int hFill = true;
int vFill = false;
setLayout(new VerticalFlowLayout(VerticalFlowLayout.TOP, hGap, vGap, hFill,
    vFill));
```

Chapter

12

Properties system concepts

The properties system provides a general framework for storing and retrieving user settings. These settings are either associated with a particular project and therefore stored in that project file, or are associated with a particular user and saved either in the `user.properties` file or a `<project>.local` file.

Each user setting, which is usually referred to as a property, has several characteristics:

- A category string used to prevent namespace collisions. (Third parties should use category names that are prefixed in the same way as Java packages are.)
- A name string used to distinguish between properties of the same category.
- A pair of setter/getter methods that provide the means to change or access the current value of that property.
- A default value, which can be implied or can be explicitly specified, to provide the initial value of the property.

The user sees neither the category nor name strings. These strings act as keys for filing and locating the information.

The behavior defined by the base `Property` class is deliberately minimal. Each subclass defines its own data type and storage mechanism, and provides appropriate getter and setter implementations. Subclasses should follow these general guidelines:

- The initial value of a property is set to its default value.
- When you set property values, delete those values that match the property's default value, to minimize the property's storage size.

Most property subtypes can notify listeners of changes to the underlying property value. These listeners aren't notified when the property value is originally read, but only when a subsequent change to the value occurs. In practice, this facility is of interest only to subsystems that must be notified of changes to properties of another subsystem.

Features and subsystems

A subsystem usually manages its own properties using the `PropertyGroup` interface. A `PropertyGroup` implementation can:

- Supply a `PropertyPage` to appear in the appropriate properties dialog box.
- Provide notification when global properties are loaded.
- Provide a convenient way to organize related Property constants.

These are the `Property` subclasses used to provide node and global property support:

- `NodeProperty`
- `NodeArrayProperty`
- `NodeArrayPropertyDefault`
- `MultipleNodeProperty`
- `ProjectAutoProperty`
- `ProjectAutoArrayProperty`
- `ProjectAutoArrayPropertyDefault`
- `GlobalProperty`
- `GlobalArrayProperty`
- `GlobalColorProperty`
- `GlobalIntegerProperty`
- `GlobalBooleanProperty`
- `GlobalFloatProperty`

Because global properties aren't read until after the command-line parsing process, your OpenTools shouldn't attempt to get or set these values in command-line handlers.

Node-specific properties

Node-specific properties fall into two subtypes: Node and ProjectAuto properties. The ProjectAuto properties are those considered to be user-specific and are recorded in a separate `.local` file associated with the project.

NodeProperty and NodeArrayProperty

Properties you set for a particular `Node` instance and will cause its project to be "dirtied". These new settings are retained only after the project is closed and if the user elected to save the project.

You typically define a project property by creating a `public` constant to represent the property, and to provide a default setting when appropriate.

```
public static final String CATEGORY = "my_node_properties";
public static final NodeProperty PROPKEY_ENABLED =
   new NodeProperty(CATEGORY, "enabled", "0");
```

You can then read or write the property value using this property accessor:

```
System.out.println("enabled=" + PROPKEY_ENABLED.getValue(node).equals("1"));
```

As an alternative, you can use the property accessors built into the `Node` class, which produce identical results:

```
System.out.println("enabled=" + node.getProperty(CATEGORY, "enabled",
    "0").equals("1"));
```

If you attempt to set a `NodeProperty` on a `Node` which does not have a parent, then an "Additional Settings" folder will be added to the project so that the property setting can be persisted. You can avoid that behavior by using the variant of the `setValue()` method with the `checkPersistence` parameter set to `false`. However, if you do that and fail to later provide a parent, the property setting will ultimately be lost when the application closes.

ProjectAutoProperty and ProjectAutoArrayProperty

Properties you set for a particular `Node` instance will not cause its project to be “dirtied”, because they are recorded instead in a `.local` file in the same directory as the project. It is saved automatically without user interaction, but not necessarily immediately, as each auto property is changed. This file will be ignored by the Team Development subsystem since it is considered to be user-specific.

Global properties

Global properties are user-specific settings that aren’t tied to a particular project. These settings are preserved in a file named `user.properties`. The IDE reads `user.properties` when it starts up, and it writes to this file whenever the user closes the properties dialog box without canceling. The IDE also writes to `user.properties` just before it shuts down.

Your subsystem that defines a property usually creates a `static public` constant to represent the property:

```
public static final String CATEGORY = "my_global_properties";
public static final GlobalProperty ENABLED =
    new GlobalProperty(CATEGORY, "enabled", "0");
```

You read or write the property value using this `static` property instance:

```
System.out.println("enabled value is " + ENABLED.getValue());
```

Managing sets of properties with PropertyManager

Implementations of the `PropertyGroup` interface are typically registered by OpenTools in the UI category. This is done by calling the `static registerPropertyGroup()` method. When all UI OpenTools are registered, the IDE loads global property settings and notifies each property group with its `initializeProperties()` method.

```
public class FavoritesPropertyGroup implements PropertyGroup {
  public static final Object FAVORITES_TOPIC = new Object();
  public static final String CATEGORY = "favorites";
  public static GlobalProperty FAVORITE_FOOD =
    new GlobalProperty(CATEGORY, "food", "cookie");
  public static GlobalProperty FAVORITE_DRINK =
    new GlobalProperty(CATEGORY, "drink", "milk");

  public static void initOpenTool(byte majorVersion, byte minorVersion) {
    if (majorVersion == PrimeTime.CURRENT_MAJOR_VERSION) {
      PropertyManager.registerPropertyGroup(new FavoritesPropertyGroup());
    }
  }

  public PropertyPageFactory getPageFactory(final Object topic) {
    if (topic == FAVORITES_TOPIC) {
      return new PropertyPageFactory("Name", "Descriptive text") {
        public PropertyPage createPropertyPage() {
          return new FavoritesPropertyPage();
        }
      };
    }
    return null;
  }

  public void initializeProperties() {}
}
```

Before the PropertyManager displays a properties dialog using the showPropertyDialog() method, every registered PropertyGroup is queried to see if it has an appropriate page for the given topic. A null topic is used for the global user properties (recorded in the user.properties file), and a Project instance is used for the Project Properties dialog. You can define a custom topic which will be recognized only by your factories.

The PropertyManager adds one page to the dialog box for each group that returns a non-null PropertyPageFactory when the PropertyGroup's getPageFactory() is invoked. You can override the PropertyPageFactory.getNestedFactories() method to create subpages in the dialog navigation tree.

```
public static BrowserAction ACTION_EditFavorites =
  new BrowserAction("Favorites options...",
    'f',
    "Edit settings for favorite things",
    BrowserIcons.ICON_BLANK) {

  public void actionPerformed(Browser browser) {
    PropertyManager.showPropertyDialog(browser,
      "Favorites",
      FavoritesPropertyGroup.FAVORITES_TOPIC,
      PropertyDialog.getLastSelectedPage());
  }
};
```

The PropertyPage instance created by the factory has three primary responsibilities:

- It presents a user interface for manipulating property values.
- It transfers the current property settings into the user interface and saves changes to those settings.
- It validates the current set of values entered into the user interface.

Example: PropertyPage implementation

Your PropertyPage implementation should look much like the following:

```
public class FavoritesPropertyPage extends PropertyPage {
  private JTextField editFood = new JTextField();
  private JTextField editDrink = new JTextField();

  public FavoritesPropertyPage() {
    try  {
      jbInit();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  private void jbInit() throws Exception {
    // build the UI
  }

  private void cleanup() {
    // remove any listeners, etc
  }

  public boolean isPageValid() {
    boolean valid = true;
    if (editFood.getText().trim().equals("candy")) {
      reportValidationError(editFood, "An invalid food has been entered");
```

```
      valid = false;
    }
    return valid;
  }

  public void writeProperties() {
    FavoritesPropertyGroup.FAVORITE_FOOD.setValue(editFood.getText().trim());
    FavoritesPropertyGroup.FAVORITE_DRINK.setValue(editDrink.getText().trim());
  }

  // This method will also be called by the Reset button in a Property dialog
  public void readProperties() {
    editFood.setText(FavoritesPropertyGroup.FAVORITE_FOOD.getValue());
    editDrink.setText(FavoritesPropertyGroup.FAVORITE_DRINK.getValue());
  }

  public void pageCanceled() {
    cleanup();
  }

  public void removeNotify() {
    cleanup();
    super.removeNotify();
  }
}
```

Sometimes it's useful to know whether one or more page factories exist for a given topic. The method `getPageFactories()` accepts a topic and returns an array of `PropertyPageFactory` instances. If you ask the factory to create a `PropertyPage` which you do not display, you need to invokes its `pageCanceled()` method so it can do any needed cleanup.

Locating user settings files

There are some `static` convenience methods associated with the `PropertyManager` class for finding settings files:

- `getInstallRootUrl()` returns a reference to the directory where JBuilder is installed.
- `getSettingsRootUrl()` returns a reference to the directory used to store settings for the current user.
- `getSettingsUrl()` takes a setting file name and returns a reference to the appropriate file in the settings root directory.

These methods all return an `Url`.

Here is an example of finding files relative to where the application is installed:

```
Url libExt = PropertyManager.getInstallRootUrl().getRelativeUrl("../lib/ext");
Url[] urls = VFS.getChildren(libExt, Filesystem.TYPE_FILE);
for (int i = 0; i < urls.length; i++) {
  System.out.println(urls[i].getLongDisplayName());
}
```

Here is an example of creating a properties file in the same directory as the `user.properties` file:

```
Url url = PropertyManager.getSettingsUrl("test.properties");
if (!VFS.exists(url)) {
  try {
    OutputStream os = VFS.getOutputStream(url);
```

```
      os.close();
    }
    catch (Exception ex) {
      System.err.println("Unable to create " + url);
    }
  }
```

Chapter 13

Version Control System (VCS) concepts

Primetime includes a general framework to integrate any version control system (VCS) and manage projects with this VCS directly from the IDE. The API has been designed to be small and simple, thus making the job of interfacing the VCS with the IDE as easy as possible. This document outlines what you must do to add support to Primetime for a given VCS.

A complete implementation of the VCS API is presented in the `SampleVCS` OpenTools sample. The code can be found in the `samples` directory and can be used as a guideline to integrate any VCS, taking advantage of the History pane and the VCS Commit Dialog. This document uses snippets of code from the `SampleVCS` class to illustrate examples of implementation of the VCS API.

Significant changes were made in Primetime 4.8, which JBuilder 2005 shipped on.

Features and subsystems

All the classes involved in Version Management are in the packages included in the `com.borland.primetime.teamdev` tree. They are:

- `vcs` — The VCS framework.
- `frontend` — The VCS UI framework.
- `cvsimp` — CVS backend implementation.
- `vssimp` — Visual Source Safe backend implementation.
- `clearcase` — ClearCase backend implementation.
- `svn` — Subversion backend implementation.
- `starteam` — StarTeam backend implementation.

To integrate a VCS into JBuilder, you must create a subclass of the abstract class `com.borland.primetime.teamdev.vcs.VCS`. This class defines the API required to register and integrate the VCS into JBuilder.

Some of these methods are used to define a series of actions that will be accessible with the Team menu. The scope of these actions is totally under your control and there

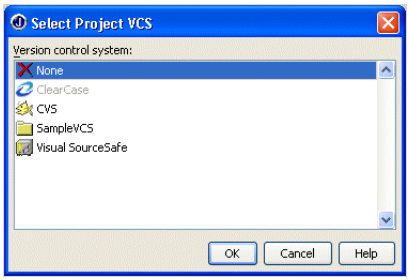
are no specific constraints or hierarchies to follow. In other words, if the implementer of a VCS back-end must display a dialog box (to administer access rights, for example) the action can define any UI element and execute any action without having to fit into a specific framework. The usual OpenTools and Swing guidelines apply. You can access any feature of the selected VCS without having to adhere to a minimum common denominator.

Configuration of the VCS

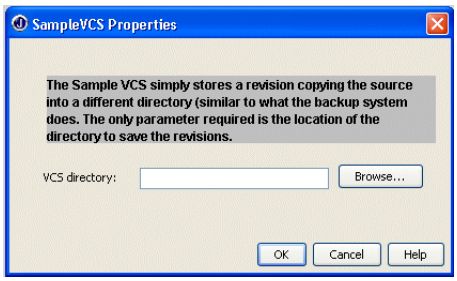
To make a VCS available in JBuilder, the class must register itself with the VCS Factory (`com.borland.primetime.vcs.VCSFactory`). For example:

```
package com.borland.samples.samplevcs;

import com.borland.primetime.teamdev.vcs.*;

public static void initOpenTool( byte major, byte minor ) {
  // VCS support was added starting from JBuilder 4
  if ( major < 5) {
    return;
  }
  VCSFactory.addVCS(new SampleVCS());
}

...

public String getName() {
  return "SampleVCS";
}
```

This makes the VCS selectable in the Select Project VCS dialog, invoked from the Team|Select Project VCS menu. Note that the `getName()` method returns the name that will be associated with the VCS. This is the result in the Team property page:



The SampleVCS is added to the list of registered VCS's. Here is the result of calling `VCS.getProjectConfigPageNew(Project)` and `VCS.getProjectConfigPage(JBProject)`:



To provide a configuration panel to the user, define a subclass of `com.borland.primetime.properties.PropertyPage` (basically a `JPanel`). In our example, the page has only a text field to enter a directory name and a button to set this value using JBuilder's `com.borland.primetime.vfs.ui.UrlChooser`.

JBuilder obtains this page:

- In JBuilder 5, 6, 7, and 8, by calling the `VCS.getProjectConfigPage(JBProject project)` method.
- For JBuilder versions past JBuilder 8, by calling `VCS.getProjectConfigPageNew(Project project)` instead.

Therefore, it's important that both these methods be implemented in your VCS implementation, in order to ensure that your VCS implementation is supported in JBuilder 5 and greater.

Saving project settings

Save VCS settings together with the project to make them persistent. You can do this using the `com.borland.jbuilder.node.JBProject.setAutoProperty()` method. The method accepts three parameters:

1. The category for the property.

   It should always be `VCS.CATEGORY`.

2. The name of the property.

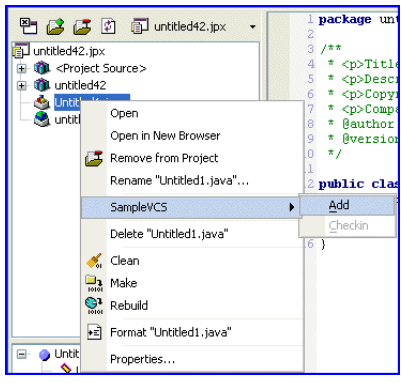
   This name should be unique so that it doesn't conflict with other VCSs. It's a good idea to use the name of the VCS to prefix the property name (for example, `SampleVCS_path`).

3. The property's value.

Here is how `SampleVCS.ConfigPage` saves and restores its properties:

```
public void writeProperties() {
  project.setAutoProperty(VCS.CATEGORY, SampleVCS.PROP_PATH,
      pnlConfig.getPath());
}

public HelpTopic getHelpTopic() {
  return null;
}

public void readProperties() {
  String path = project.getAutoProperty(VCS.CATEGORY, SampleVCS.PROP_PATH);
  if (path != null) {
    pnlConfig.setPath(path);
  }
}
```

Context menus



The `getVCSContextMenuGroup()` method is called by JBuilder to retrieve a list of Actions that can be displayed when the user right-clicks on a file that is under the VCS. The method returns an `ActionGroup` initialized with all the actions that can be available when right-clicking on files under the given VCS. If the method return `null` then no menu will be displayed. Here is how the SampleVCS exports its context menu:

```
public ActionGroup getVCSContextMenuGroup() {
  ActionGroup aGroup = new ActionGroup("SampleVCS", 'S',
                                       "Manage file with the Sample VCS", null,
                                        true);
  aGroup.add(Actions.ADD);
  aGroup.add(Actions.CHECKIN);
  aGroup.add(Actions.GET_LOG);
  return (aGroup);
}
```

The `Actions` class defines the static `Action` objects used by several methods of the `SampleVCS` class:

```
public class Actions {
  ...
  public static BrowserAction ADD = new BrowserAction("Add", 'A',
                                    "Add the selected file(s) to the VCS") {
    public void update(Browser browser) {
      setEnabled(!getEnabledState());
    }

    public void actionPerformed(Browser browser) {
      Node[] nodes = browser.getProjectView().getSelectedNodes();
      SampleVCS vcs = (SampleVCS) VCSFactory.getVCS("SampleVCS");
      int len = nodes.length;
      for (int i = 0; i & lt; len; i++) {
        File workFile = ( (FileNode) nodes[i]).getUrl().getFileObject();
        vcs.addFile(workFile);
      }
    }
  };

  public static BrowserAction CHECKIN = new BrowserAction("Checkin", 'C',
                                        "Commit changes to the repository") {

    public void update(Browser browser) {
      setEnabled(getEnabledState());
    }

    public void actionPerformed(Browser browser) {
      Node[] nodeList = browser.getProjectView().getSelectedNodes();
      SampleVCS vcs = (SampleVCS) VCSFactory.getVCS("SampleVCS");
      String msg = "FIX ME !";
      vcs.checkin(nodeList, msg);
    }
  };

  public static BrowserAction GET_LOG = new BrowserAction("See log", 'L',
                                    "See the comments for changes checked in") {
    public void update(Browser browser) {
      setEnabled(false);
    }

    public void actionPerformed(Browser browser) {
      ...
    }
  };
}
```

Integration in the History pane

JBuilder's users expect to see all the available revisions in the History pane. Fortunately, this is one feature that is both very useful yet easy to implement. JBuilder calls the `VCS.getRevisions()` method to retrieve a `Vector` of `RevisionInfo` for a specified file. The method receives an `Url` (the JBuilder version, not the one in the `java.net` package) and fills the `Vector` with all the revisions found for the file. The `RevisionInfo` class is designed to report revision information in a VCS-neutral way (at least that was the goal). Here is an example:

```
/**
* Retrieve all the revisions of a given file
*/

public Vector getRevisions(Url url) {
  Vector revs = new Vector();
  File repo = getFileInRepository(url.getFileObject()).getParentFile();

  final String fname = url.getName();

  String[] list = repo.list(new FilenameFilter() {
    public boolean accept(File dir, String name) {
      return name.startsWith(fname) & amp; & amp; (!name.endsWith(".comment"));
    }
  });

  int len = list.length;
  String revNumber;

  for (int i = 0; i & lt; len; i++) {
    revNumber = list[i].substring(list[i].lastIndexOf(',') + 1);

    try {
      BufferedReader reader = new BufferedReader(new FileReader(new File(repo,
                                                 list[i] + ".comment")));
      String desc = reader.readLine();
      String who = reader.readLine();
      String time = reader.readLine();
      String label = reader.readLine();

      GregorianCalendar cal = new
          GregorianCalendar(TimeZone.getTimeZone("GMT"));
      cal.setTime(new Date(Long.parseLong(time)));
      cal.setTimeZone(TimeZone.getDefault());

      RevisionInfo r = new RevisionInfo(revNumber, cal.getTime().getTime(),
          who, desc, label);

      if (getCurrentRevision(url.getFileObject()).equals(revNumber)) {
        r.setWorkingRevision(true);
      }
      revs.add(r);
    }

    catch (Exception ex) {
      ex.printStackTrace();
    }
  }
```

```
    // The VCSCommitBrowser assumes that the first element has the newest
       revision
    Collections.reverse(revs);

    return revs;
  }
```

As the example demonstrates, the parameters passed to `RevisionInfo` are:

- The revision number
- The time of the revision (in this case time is expressed using the GMT time zone)
- The name of the user who checked in the revision
- The description of the changes
- The version label

The revision number is expressed here as a `String`, but it's actually more complex than that. There's a class designed to represent version numbers in a way that is more suitable for sorting than using `String`s. See:

- `com.borland.primetime.teamdev.vcs.AbstractRevisionNumber`
- `com.borland.primetime.teamdev.vcs.NumericRevisionNumber`
- `com.borland.primetime.teamdev.vcs.StringRevisionNumber`

Note that care is taken to identify the revision that was used to check out the file under consideration. Depending on the VCS, you can obtain this value in a variety of ways. You can notify JBuilder of this condition using the `RevisionInfo.setWorkingRevision(boolean)` method. From the previous example:

```
if ( getCurrentRevision(url.getFileObject()).equals(revNumber) ) {
  r.setWorkingRevision(true);
}
```

Providing project-wide status: The VCSCommitBrowser

There is a point in the development cycle when single-file management isn't sufficient. After modifying, adding or removing several files, a developer wants to know how many files must be committed to the repository and what types of changes occurred. The `VSCCommitBrowser` is designed to provide a project-wide view of the changes and to provide one-click access to the Visual Diff Engine, both for changes done to the workspace, and to see what happened in the repository if the VCS, like CVS, allows concurrent updates.

The `VCSCommitBrowser` class is part of the `com.borland.primetime.teamdev.frontend` package and can be called using the `vcs.VCSUtils.showProjectStatus(CommitAction checkinAction)` method.

This method calls in turn the `vcs.VCS.getProjectStatus(Project project)` method. The return value from `getProjectStatus()` is a `java.util.Map` where each key identifies a directory and the associated value is a `java.util.List` containing information about related files. Both the key and the elements of the List are instances of `vcs.VCS.VCSFileInfo`.

`VCSFileInfo` is a simple storage class used to return status information about a file managed by a VCS. The purpose of `VCSFileInfo` is to simply hold some file information since the `VCSCommitBrowser` can display data about files that are not directly under control of JBuilder's projects. The class keeps a flag that can be used by the VCS implementation to check if the file has to be committed. This flag is set or cleared by clicking the Exec check box in the `VCSCommitBrowser`.

Perhaps the most important component of the `VCSFileInfo` class is the status field. It is of type `vcs.VCSFileStatus`, an abstract class that must be subclassed by any VCS implementation that needs to use the `VCSCommitBrowser`.

Subclassing `VCSFileStatus` provides these advantages:

- You can provide a VCS-specific description of the status of the file and you can associate an `Icon` with the description.

  Stock icons are provided by JBuilder in the `frontend.VCSIcons` class.
- You can add right-click (context, or popup) menus.
- You can provide one-click diff (both workspace and repository-based) with no coding involved.
- The whole process is very simple and can be implemented quickly.

Example: RevisionStatus

Here is an example from `SampleVCS`:

```
public class RevisionStatus extends VCSFileStatus {

  public static final int STATUS_NEW = 0;
  public static final int STATUS_WSP_CHANGE = 1;
  public static final int STATUS_REPO_CHANGE = 2;
  public static final int STATUS_REMOVED = 3;
  public static final int STATUS_UNMODIFIED = 4;
  public static final int ACTION_COMMIT_CHANGES = 0;
  public static final int ACTION_ADD_TO_REPO = 1;
  public static final int ACTION_UPDATE_WSPACE = 2;
  public static final int ACTION_REMOVE_FROM_WSPACE = 3;
  public static final String ACTION_DESC_COMMIT_CHANGES = "Commit";
  public static final String ACTION_DESC_ADD_TO_REPO = "Add";
  public static final String ACTION_DESC_UPDATE_WSPACE = "Update";
  public static final String ACTION_DESC_REMOVE_FROM_WSPACE = "Delete";
  public static final String ACTION_TIP_COMMIT_CHANGES =
      "Commit workspace changes to repository.";
  public static final String ACTION_TIP_ADD_TO_REPO =
      "Add file from workspace to repository.";
  public static final String ACTION_TIP_UPDATE_WSPACE =
      "Update workspace with new changes in repository.";
  public static final String ACTION_TIP_REMOVE_FROM_WSPACE =
      "Delete file from local workspace.";
  private VCSFileActions vcsFileActions = null;

  private class FileDesc {
    public Icon icon;
    public String desc;

    public FileDesc(Icon icon, String desc) {
      this.icon = icon;
      this.desc = desc;
    }
  }

  private FileDesc[] descs = {
      new FileDesc(VCSIcons.FILE_NEW,
                   "Not found in repository"),
      new FileDesc(VCSIcons.WORKSPACE_CHANGE,
                   "Changed in workspace"),
      new FileDesc(VCSIcons.REPOSITORY_CHANGE,
                   "Changed in repository"),
      new FileDesc(VCSIcons.FILE_REMOVED,
                   "Not in workspace")
  };
```

```
public RevisionStatus(int status) {
  this.status = status;
}

/**
 * Returns the icon that will be displayed in the VCSCommitBrowser
 */
public Icon getStatusIcon() {
  return descs[status].icon;
}

/**
 * called by VCSCommitBrowser to determine if the file has repository changes
 */
public boolean isModifiedInVCS() {
  return status == STATUS_REPO_CHANGE;
}

public String getDescription() {
  return descs[status].desc;
}

public boolean isModifiedLocally() {
  return status == STATUS_WSP_CHANGE;
}

public boolean isNew() {
  return status == STATUS_NEW;
}

/**
 * Based on the VCS status of the file, return an appropriate set of VCS
 * actions that can be performed on this file.
 *
 * @return an appropriate set of actions to choose from in the commit browser
 */
public VCSFileActions getVCSFileActions() {
  if (vcsFileActions != null) {
    // Action has been cached. Don't bother with the rest.
    return vcsFileActions;
  }

  switch (status) {
    case STATUS_NEW:
      String[] descs = {
          ACTION_DESC_ADD_TO_REPO, ACTION_DESC_REMOVE_FROM_WSPACE};
      int[] actions = {
          ACTION_ADD_TO_REPO, ACTION_REMOVE_FROM_WSPACE};
      String[] tips = {
          ACTION_TIP_ADD_TO_REPO, ACTION_TIP_REMOVE_FROM_WSPACE};
      vcsFileActions = new VCSFileActions(descs, actions, tips);

      break;

    case STATUS_WSP_CHANGE:
      String[] descs2 = {
          ACTION_DESC_COMMIT_CHANGES};
      int[] actions2 = {
          ACTION_COMMIT_CHANGES};
```

```
          String[] tips2 = {
              ACTION_TIP_COMMIT_CHANGES};
          vcsFileActions = new VCSFileActions(descs2, actions2, tips2);

          break;

        case STATUS_REPO_CHANGE:
          String[] descs3 = {
              ACTION_DESC_UPDATE_WSPACE};
          int[] actions3 = {
              ACTION_UPDATE_WSPACE};
          String[] tips3 = {
              ACTION_TIP_UPDATE_WSPACE};
          vcsFileActions = new VCSFileActions(descs3, actions3, tips3);

          break;

        default:

          break;

      }
      return vcsFileActions;
    }
  }
```

Note the definition of the methods `isModifiedInVCS()` and `isModifiedLocally()`. The `VCSCommitBrowser` uses these to determine what kind of Diff tab must displayed and how to retrieve the revisions for the Visual Diff. Refer to the `SampleVCS` code (the `Actions` class) to see how this plugs into the JBuilder menu system.

The `vcs.CommitAction` class is a subclass of Swing's `Action`, with two methods to report error conditions. The action is called when the user chooses OK in the `VCSCommitBrowserDialog`. The idea behind it is that the action will scan all the `VCSFileInfos` that are set to be committed and it will perform the appropriate action based on the specific VCS rules.

In conclusion, the `VCSCommitBrowser` and related classes provide a consistent look and feel to analyze what changes need to be committed to the VCS repository. You have a minimal amount of work to do to provide this effective UI element. The most complicated bit of work is the implementation of the `CommitAction`.

Changes in Primetime 4.8

New Methods Passing Project

In Primetime 4.8, the VCS API was significantly modified. When the VCS API was originally implemented, Primetime did not have `ProjectGroup`s, and the API assumed that the VCS implementation could determine the project by invoking `Browser.getActiveProject()`. With `ProjectGroup`s, that is no longer a safe assumption, as it is possible for a VCS action to be performed on a non-active project. For example, the user can invoke a VCS context menu action on a node in a non-active project in a project group.

For all the methods that need a `Project`, but where the `Project` was not being passed in, a new method that passes either a `Project` or a `UrlNode` now exists. If an `UrlNode` is passed in, the `Project` can be determined by invoking `UrlNode.getProject()`.

Old methods	New methods
isUnderVCS(Url)	isUnderVCS(Project, Url)
getRevisions(Url)	getRevisions(UrlNode)

Old methods	New methods
getSource(Url, RevisionInfo)	getSource(UrlNode, RevisionInfo)
isBinary(Url)	isBinary(UrlNode)
getVCSContextMenuGroup	getVCSContextMenuGroup(Node [])
addToIgnoreList(Url [])	addToIgnoreList(Project, Url [])
removeFromIgnoreList(Url [])	removeFromIgnoreList(Project, Url[])

Because the old methods are part of the published API, those old methods still need to be implemented. Although the old methods are no longer internally invoked by any Primetime code, it is possible that non-Borland OpenTools may be invoking them. The implementation of the old methods unfortunately have no choice but to invoke `Browser.getActiveBrowser()`:

```
public Vector getRevisions(Url url) {
  Project project = Browser.getActiveBrowser().getActiveProject();
  UrlNode urlNode = project.getNode(url);
  return getRevisions(urlNode);
}
```

Other new methods

- **`initUI()`**

  VCS implementations should now perform any UI actions, such as registering Wizard actions, in this method. Before Primetime 4.8, VCS Wizard actions were typically registered in the VCS' constructors.

- **`getWorkingRevision(UrlNode)`** and **`getRepositoryTip(UrlNode)`**

  Both of these methods already have default implementations. They both call `VCS.getRevisions(UrlNode)`, and then iterate through the resulting `Vector` to determine the working revision or repository tip. If a VCS implementation can more efficiently determine the working revision and/or repository tip, then the VCS implementation should override accordingly.

- **`checkOut(UrlNode, RevisionInfo)`**

  Checks out the specified revision. The default implementation is to throw an exception, so every VCS implementation should override.

- **`getProjectStatus(Project, Url[])`**

  Recursively gets the project status for the specified `Url`s. This enables the user to, for example, run the Commit Browser on a particular directory without having to determine the status for the whole project.

- **`supportsProjectUpdate(Project)`** and **`update(Project, VCSUpdateProcess)`**

  The default implementation for `supportsProjectUpdate()` returns `false` and the default implementation for `update(Project, VCSUpdateProcess)` does nothing. If a VCS implementation can update a project, it should override `supportsProjectUpdate()` to return `true` and implement `update()`. The implementation for `update()` should avoid talking directly to the UI and not access either `Browser` or Swing in general. Instead it should report all progress, errors and warnings via the `VCSUpdateProcess` object. This will allow command-line updates to work when no `Browser` instance is present.

- **`AbstractAnnotatedFile getAnnotatedFile(TextFileNode textFileNode, StatusListener statusListener)`**

  This method has a default implementation to download the contents of all revisions of a file and calculate the differences between each revision. If a VCS implementation can more efficiently determine diffs between repository revisions, then it should override this method. For example, both CVS and Subversion have

"annotate" subcommands to determine repository diffs, and are more efficient at calculating the diffs since they do not need to download the content of all revisions.

- `supportsChangeComment()` and `changeComment(UrlNode urlNode, RevisionInfo revisionInfo, String newComment)`

  The default implementation of `supportsChangeComment()` returns false, and `changeComment()` does nothing. If a VCS allows editing of the checkin comments, the implementation should override `supportsChangeComment()` to return true, and override `changeComment()` to change the comment.

- `guessFileStatus(Project project, Url dir, boolean requestRepositoryChanges)`

  Returns a `Map` containing a guess of the file status of all files in `dir`. The default implementation returns an empty `Map`. Note that this call should only calculate the status of the direct children of `dir`; it should not recursively descend into children of the children. This call is invoked in a background thread by the VCS decorators, and its return value is used to decorate files in the project tree. As such, it should be quick to execute. If it is significantly cheaper to determine a partial status, rather than the full status, then the method should do so. That's why the method's name contains the word "guess".

- `getDirsNeededByVCS()` and `getFilesNeededByVCS()`

  The default implementations of these methods return empty Sets. VCS implementations should override to indicate directory names and or filenames that are for reserved use. Note that if a reserved filename only exists in a reserved directory, then `getFilesNeededByVCS()` should not return the reserved filename.

- `isConfigured(Project)`

  Indicates whether the VCS is configured for the project.

New OpenTools Category

Previously, VCS implementations would register themselves in the "UI" OpenTools category. As of Primetime 4.8, VCS implementations should register themselves in the "VCS" OpenTools category. The reason for this change, as well as for the existence of the new `VCS.initUI()` method, is that a new `-update` switch exists which allows a VCS update to be executed from the command-line without the "UI" OpenTools category being initialized.

Chapter

14

Util package concepts

The com.borland.primetime.util package contains many support classes that may prove useful in the design of your own OpenTools addins to JBuilder. The majority of these classes are public, yet are not fully documented as part of the OpenTools API. The classes are all Swing-based. The ones discussed in this document are:

- AssertionException
- CharsetName
- CharToByteJava
- ClipPath
- Debug
- DummyPrintStream
- JavaOutputStreamWriter
- OrderedProperties
- Profiler
- RegularExpression
- SoftValueHashMap
- Streams
- Strings
- Text
- VetoException
- WeakValueHashMap

Util support classes

AssertionException

This class extends RunTimeException. Generally, you use AssertionException in combination with Debug to test and report self-check failures; however, it may be used by itself:

```
void doIt(String str) {
  try {
    AssertionException.assertState(str != null, "null string input to doIt()");
   }
  catch (AssertionException ex) {
    ex.printStackTrace();
  }
}
```

CharsetName

`CharsetName` provides a routine which may be used to convert the internal file encoding name to an IANA name. Typically you use this for the `charset` string of a `meta` HTML tag.

```
String ianaName = CharsetName.javaToIANA(System.getProperty("file.encoding"));
```

CharToByteJava

This class extends `CharToByteConverter`. It provides Unicode escape sequences instead of an `UnknownCharacterException` or character substitution. This class is used by `JavaOutputStreamWriter`.

```
static byte[] stringToByte(String text) {
  CharToByteJava c2bj = new CharToByteJava();
  int strlen = text.length();
  byte[] buffer = new byte[strlen * c2bj.getMaxBytesPerChar()];
  StringBuffer fsb = new StringBuffer(text);
  int nBytes = 0;

  try {
    nBytes = c2bj.convert(fsb.getValue(), 0, strlen, buffer, 0, strlen);
    byte[] b = new byte[nBytes];
    System.arraycopy(buffer, 0, b, 0, nBytes);
    buffer = b;
  }
  catch (Exception ex) {
    Debug.printStackTrace(ex);
  }
  return buffer;
}
```

ClipPath

Use this class to truncate a file pathname in the middle (substituting "...") if it exceeds a certain width.

```
String getDisplayNameForMenuItem(FileNode fileNode, int maxPixels) {
  String text = fileNode.getLongDisplayName();
  text = ClipPath.clipCenter(UIManager.getFont("MenuItem.font"), text,
      maxPixels);
  return text;
}
```

Debug

`Debug` is intended to help you debug your code during development. When you are ready to ship your code, just tell the compiler to exclude this class and recompile.

Support of assertions is provided. If `enableAssert()` has been used to enable them (they are on by default), a failed assertion will result in an `AssertionException` being thrown.

```
void doIt(String str) {
  try {
     Debug.ensure(str != null, "null string input to doIt()");
  }
  catch (AssertionException ex) {
    ex.printStackTrace();
  }
}
```

Support for tracing your code flow is provided. If `addTraceCategory()` has been used to enable a particular trace type, then messages, stack traces, or conditional warning messages will be output. This output can be to either `System.err` (the default) or to any other `PrintStream` that you define.

```
static final String TRACE = "MyTrace";

void test(String str)
  Debug.enableOutput(true);
  Debug.addTraceCategory(TRACE);
  doIt(str);
  Debug.removeTraceCategory(TRACE);
}

void doIt(String str) {
  Debug.trace(TRACE, "doIt() input str=" + str);
  Debug.warn(TRACE, str == null, "doIt()input str is null");
}
```

Support for debugging of paint logic is provided. After painting, you can call `debugRect` to draw a color-coded rectangle around the area.

```
public void paintComponent(Graphics g) {
  Rectangle clip = g.getClipBounds();
  super.paintComponent(g);
  Debug.debugRect(g, clip.x, clip.y, clip.width, clip.height);
}
```

DummyPrintStream

This class extends `PrintStream`. Use it to redirect output that you want to suppress.

```
static final String TRACE = "MyTrace";

void test(boolean bLog) {
  doIt(bLog ? System.out : new DummyPrintStream());
}

void doIt(PrintStream out) {
  out.println("my output");
}
```

JavaOutputStreamWriter

This class extends `OutputStreamWriter`. It writes characters to an output stream with translation of characters into bytes according to the given or current default character encoding. Characters that cannot be represented in that encoding will appear as Unicode escapes.

```
public void commit(FileNode node, String outText) {
  try {
    OutputStream os = node.getBuffer().getOutputStream();
    OutputStreamWriter osw = new JavaOutputStreamWriter(os, new
        CharToByteJava());
    osw.write(outText);
    osw.close();
  }
  catch (Exception ex) {
    Debug.printStackTrace(ex);
  }
}
```

OrderedProperties

This class extends `Properties`. It remembers the order in which properties are added and writes them in the same order. International characters are written as Unicode escapes.

RegularExpression

`RegularExpression` provides simple expression pattern matching.

```
Url[] getPropertyFilesInPackage(FileNode fileNode) {
  RegularExpression regexp = new RegularExpression("*.properties");
  Url packageUrl = fileNode.getUrl().getParent();
  Url[] files = VFS.getChildren(packageUrl, regexp, Filesystem.TYPE_FILE);
}
```

SoftValueHashMap

This class extends `java.util.HashMap`. Your values to be saved are put in `java.lang.ref.SoftReference` wrappers. This allows them to be garbage-collected at times of high memory demand. Therefore, a value stored using `put()` may later be returned as `null` when retrieved via `get()`, if the value was garbage-collected. This is a way to implement a memory-sensitive cache, if you can reproduce the objects.

```
private static Map myCache = new SoftValueHashMap();
public static synchronized MyValue getMyValue(Object myKey) {
  MyValue mv = (MyValue)myCache.get(myKey);
  if (mv == null) {
    mv = new MyValue(myKey);
    myCache.put(myKey, mv);
  }
  return mv;
}
```

Streams

`Streams` contains routines to read both binary data and text from `InputStreams` and a utility method to copy data from an `InputStream` to an `OutputStream`.

```
InputStream in = VFS.getInputStream(new Url(new File("myFile.bin")));
byte[] data = Streams.read(in);
in.close();

in = VFS.getInputStream(new Url(new File("myTextFile.txt")));
char[] content = Streams.readChars(in, "default");
in.close();

in = VFS.getInputStream(new Url(new File("mySourceFile")));
OutputStream out = VFS.getOutputStream(new Url(new File("myDestFile")));
Streams.copy(in, out);
in.close();
out.close();
```

Strings

`Strings` contains string manipulation routines which generally fall into two groups:

- The first group is convenience wrappers for `java.text.MessageFormat.format()`, which build the argument array for you based on input parameters:

```
System.out.println(Strings.format("Project name is \"{0}\"",
    project.getDisplayName()));
```

- The second group is associated with encoding and decoding strings, so that they can be saved in a file and restored. This includes escaping international characters and other characters such as delimiters:

```
Strings.StringEncoding SPACE_TO_UNDERBAR_ENCODING = new
    Strings.StringEncoding(" _");
System.out.println(Strings.format("{0}={1}",
                  SPACE_TO_UNDERBAR_ENCODING.encode("string one"),
                  Strings.encode("string"));
```

  You can use the “standard” encoding or supply your own.

VetoException

`VetoException` extends `java.lang.Exception`. It is used by wizards to indicate a condition that needs correction before moving to the next step, or starting/completing the Finish button processing:

```
public void checkPage() throws VetoException {
  String text = jTextField1.getText().trim();

  if (text.length() == 0) {
    JOptionPane.showMessageDialog(this,
                                  "No string has been entered",
                                  "Error",
                                  JOptionPane.ERROR_MESSAGE);
    jTextField1.requestFocus();
    throw new VetoException();
  }
}
```

WeakValueHashMap

This class extends `java.util.HashMap`. Your values to be saved are put in `java.lang.ref.WeakReference` wrappers. Such references do not prevent their referents from being garbage-collected. Therefore, a reference stored using `put()` may later be returned as `null` when retrieved via `get()`, if the referent object was garbage-collected. This is a way to implement a memory-sensitive cache, if you can reproduce the objects.

```
private static Map myCache = new WeakValueHashMap();
public static synchronized MyValue getMyValue(Object myKey) {
  MyValue mv = (MyValue)myCache.get(myKey);
  if (mv == null) {
    mv = new MyValue(myKey);
    myCache.put(myKey, mv);
  }
  return mv;
}
```

Chapter

15

Personality concepts

Primetime supports hosting multiple toolsets within the same IDE. For example, both Java and C++ development can be supported in the same instance of Primetime.

To accomplish this, sets of tools must be enabled/disabled or made visible/invisible when the user changes from a Java project to a C++ project. Tools may also be filtered based on other criteria, such as user preferences or the type of project being developed (e.g. EJB, Web). To accomplish this, Primetime uses the concept of *personality*.

Personalities form a flexible categorization scheme for tools and their associated pieces: actions, properties, GUI components, command-line options, and so on. In this document, the general term *tool* is used for any piece of functionality surfaced through Primetime. Each tool can be associated with a set of personalities. With this association of tool to personality, tools can be filtered out of the current IDE view, based on what set of personalities is needed in any context.

Initial personality contexts

Initially, there are two contexts that define the desired personalities: a *global* context and a *project* context.

- The global context is used at startup to determine which is the startup personality — for example, start up as JBuilder or start up as CBuilder. It is only used when both C++ and Java personalities are running in the same instance of `PrimeTime`. It is used to determine which default project to open when no user project is opened.
- The project context is used when a project is opened. Personalities are stored as project properties and can be queried by any tool. Tools can be also be notified when the current personality context changes.

The initial state for personalities are read from the global properties file and from the project file. At install time,

- A default personality file is created.
- Either the default project or the `Project` class descendant (i.e. `JBProject`, `CBProject`) defines the initial personalities for new projects.
- Properties pages are added to project properties and global properties to allow for user configuration of these personality contexts.

Personality awareness

By default, every class representing features is in the Primetime personality and will be always be available. This preserves some level of backward compatibility. But tools should define a more restrictive personality to fully take advantage of this new subsystem. Classes representing features that might only be available in a specific set of personality contexts can become *personality aware*.

Personalities should be registered from opentools listed in the OpenTools-Core category of `classes.opentools`. They need to be loaded prior to all other OpenTools.

A personality is defined by the `Personality` abstract class:

```
public abstract class Personality  {
  public abstract String[] getDepend();
  public abstract String getDescription();
  public abstract boolean isRequired();
}
```

A descendent class registers an instance of itself as a personality with the `PersonalityManager` class. `PersonalityManager` also contains `Personality` utilities methods (more on this below).

Once a personality is defined, it can be used by these types of classes:

- Classes used to implement a tool that is available in one or more personality contexts
- Classes that manage personality-aware classes (Menus, Toolbars, Wizard gallery, and so on) and listen for changes in personality

In order to maintain compatibility with the current Primetime API, the concept of personality is introduced in several ways:

- Some classes that are personality aware implement a Personality property with a `getPersonality()` and a `setPersonality()`. `UpdateAction` is modified in this way. Descendent classes can override these methods to bind a more restrictive personality to the action.
- Tool components that are implemented using interfaces (`PropertyGroup`, `Command`, `NodeViewerFactory`, and so on) are made personality aware by an alternate registration mechanism. (The current registration method defaults the set of personalities for that group to Primetime.)

  For example, `PropertyManager` adds the method:

  ```
  PropertyManager.registerPropertyGroup(PropertyGroup group, Personality[]
      personalities);
  ```

- A configuration of desired personalities is defined by a `PersonalityContext`, which is essentially a query asking which personalities to exclude in the current context:

  ```
  public class PersonalityContext {
    public PersonalityContext(Personality[] excluded) {
    }
    public Personality[] getExcluded(){
    }
    …
  }
  ```

Note OpenTools writers do not normally need to concern themselves with this class.

Making tools personality aware

The work to be done to make Primetime tools personality aware falls into several categories:

- Define personalities that encapsulate the types of grouping needed for the tools (e.g., `PrimetimePersonality`, `JBuilderPersonality`, `EJBPersonality`, `JSPPersonality`, etc.)
- Either:
  - Register the OpenTool using a new personality-aware registration method.
  - Override the `getPersonality()` implementation into a natural place in the class hierarchy of the appropriate class.

    For example, `UpdateAction` is changed to implement `getPersonality()` and return that it's in the `PrimetimePersonality`. This allows all existing update actions to automatically appear in the default configuration, without any additional changes.

    Descendent classes can introduce other personalities, or an individual instance can set the personality, and thus restrict the personality contexts they will appear in.

    Actions can also query the current personality context and make enabled/disabled decisions, etc., based on the current context.
- Modify some piece of tool-using code to filter out the tools appropriate for the current personality context. This is where much of the enablement code must be added and is specific to each subsystem. This should only concern implementers of new subsystems which surface GUI in the IDE.

Tool writers usually need only concern themselves with the first two categories.

In some cases, a piece of GUI will combine personalities in a more specific way, say within one property page. Then the implementer may need to write code that does the filtering, rather than rely on the generic mechanism.

Filtering based on personalities

Various subsystems must be modified to make tools personality aware. In general, subsystems fall into two patterns:

- Those that need to listen for changes in personality context
- Those that do not, since the personality context could not change during the lifecycle of their GUI components

Browser is an example of a subsystem that listens for changes. Commands, context menus, and property pages are examples of those that only need to check the current context once.

The personality context is usually switched either by changing the currently active project or by editing the project's personality property.

The `PersonalityPropertyPage` is a new tab on the project properties page. It's populated by querying the `PersonalityManager` for all personalities that could be applied to this project. It contains a check tree which shows all the currently selected personalities and their dependants. The dependencies identified by the personalities are used to build the tree and cascade any disablement to dependent personalities. The user can then customize the desired personalities for this project and save that personality configuration.

The initially-installed default project contains a personality query that contains `PrimetimePersonality` and excludes nothing, so, by default, all tools will appear.

Checklist for OpenTools writers

Here is what you need to do to make your tool personality-aware:

1 Either create a new Personality or identify an existing one to use.

For example:

```
public class JBuilderPersonality extends DefaultPersonality {
  public JBuilderPersonality(){
    super("JBuilder");
  }
  public String[] getDepend() {
    return new String[]{JavaPersonality.class.getName()};
  }
  public boolean isRequired() {
    return true;
  }
  // Utility instances to use in common calls
  public static final JBuilderPersonality jbuilderPersonality = new
      JBuilderPersonality();
  public static final Personality[] jbuilderPersonalities = new
      Personality[]{jbuilderPersonality};
}
```

An instance of the personality is created and registered. A single-element array of this personality is also created — since so many calls have a parameter of type `Personality[]`, this is a convenience and optimization for the majority of things which are in only one personality; they can use this static instance as the parameter.

2 Modify all actions that are registered at the “top level” of the IDE to be included in one or more Personalities.

If an action is only invoked through the GUI attached through a more restrictive action, there is no need to modify that action.

`UpdateAction` has been modified to have a `Personality[]` property; the `getPersonality()` method should be overridden in subclasses and introduce a specific set of personalities.

For example, an action that would only be available if the J2EE personality is enabled would override `getPersonalities()`:

```
public Personality[] getPersonalities() {
   return new Personality[](new J2EEPersonality());
}
```

In general, consider all actions and actiongroups that are added through any of the following:

- `Browser.addMenuGroup`
- `Browser.toolBarGroup`
- `WizardManager.registerWizardAction`
- Any actions provided through `ProjectView.registerContextActionProvider(…);`

3 Use new registration calls for many non-action-based features, such as `PropertyGroup` and `NodeViewerFactory`.

In order to avoid changes to existing interfaces, several managers have a new overloaded registration method that takes a `Personality[]` param. These calls should be used if the registered features belong in a specific personality.

Changes can be made to:

- `PropertyManager.registerPropertyGroup(PropertyGroup group, Personality[] personalities)`

- `Browser.registerNodeViewerFactory(NodeViewerFactory factory, boolean asFirst, Personality[] personalities)` and `Browser.registerNodeViewerFactory(NodeViewerFactory factory, Personality[] personalities);`

Listening for changes in personality context

To be notified when the current personality context changes, register an implementation of `PersonalityListener` with the `PersonalityManager` by calling `PersonalityManager.addPersonalityListener(PersonalityListener listener)`. You can remove this listener by calling `PersonalityManager.removePersonalityListener(PersonalityListener listener)`.

Chapter

16

JBuilder JAM/JOM concepts

The JAM/JOM subsystems are a set of interfaces and classes used to parse and generate Java code. They can be used separately, but the most power and flexibility are achieved when used together. The main use is the parsing of source or class files to extract information about their contents. Examples of this include the various wizards in the object gallery (File|New) or the UI Designer.

JAM/JOM supersedes JOT as the Two-Way Tools backbone. The JOT package is deprecated; JBuilder 2005 ships with it solely in order not to break existing JOT-dependent OpenTools.

Features and subsystems

The heart of JAM/JOM is a series of interfaces and classes that describe the Java language in detail.

JAM (Java Abstraction Model) is a read-only layer that provides information about Java classes. The information only includes classes, methods and fields.

JOM (Java Object Model) is a read/write layer that deals with files and the constructs within those files. While JAM only abstracts classes, methods and fields, JOM deals with the details of source files such as position and constructs (statements, blocks, expressions etc).

The two API's work together but are kept separate for a couple of reasons. The main distinction is that JAM deals with the class as its main unit of abstraction, whereas JOM deals with the file as its main level of abstraction. The other big distinction is that, since JAM is not concerned with the writing of source code, it can use either the class file, source file, or both. JOM is only concerned with source code. This frees the user from having to worry about what representation the file comes from; if it's JAM the user doesn't care, and if it's JOM the user knows they are working with the source code.

A typical usage of JAM and JOM is using JAM to find the classes, methods and fields of interest and then converting to JOM when reading and writing source code is required.

JAM is a read-only layer that is basically a reflection layer. It hides details of whether the information comes from class or source. JAM chooses the correct backing based on what is being asked for and what is the fastest correct way of providing that information.

Accessing JAM/JOM

When you are working with JAM/JOM, you are either working with objects (classes, methods, fields) from a file that exists already, or you are creating a new file. If the file exists, it's usually part of your project (say, a selected node). If you are creating a new file, it is usually done in the context of the current project.

Accessing JAM

To access the JAM subsystem, use the static factory method `instance(Project)` of the `JamFactoryobject`, as this example:

```
Project project; // assume a valid, non-null instance of Project
JamFactory factory = JamFactory.instance(project);
JamClass jamClass = factory.getClass("com.borland.jbuilder.jam.TestJam");
```

The above code will return a `jamClass` if the class is found on the project paths.

`JamFactory` will return `JamClass` from class information if the class is up to date; otherwise it will return a `JamClass` with the information from the source file. If the information was from class information and the file is edited but not recompiled, then JAM will switch to using the source information when requested for new information.

Example: JAM from JOM alternative

An alternative way to obtain a JAM object from JOM is to use the `getJomXXX()` method on the relevant `jom` objects.

```
JamMethod jamMethod = jomMethod.getJamMethod();
```

Accessing JOM

To access the JOM subsystem use the static factory method of `instance(project)` `JomFileManager;`,

```
Project project; // assume a valid, non-null instance of Project

JomFileManager jomManager = JomFileManager.instance(project);
```

A typical usage is to get a `JomFile` for an existing `Url` file. This could be done like this:

```
...

Url fileUrl = new Url("c:/test/Foo.java");

if (fileUrl.exists()) {
  // The getJomFile method below will return a JomFile for an existing file.
}

  // Obtains a JomFile for a file. (If the fileUrl doesn't exist a new will
     be created.)
  JomFile jomFile = jomManager.getJomFile(fileUrl);

// Use JOM to modify the file:

...

...
```

Another usage is to create the file if it doesn't exist:

```
...

fileUrl = new Url("c:/test/Foo.java");

// Obtains a  JomFile for an existing file or creates a new JomFile object:
JomFile jomFile = jomManager.getJomFile(fileUrl);

// Use JOM to modify the file:
...

jomFile.commit class=(true);

...
```

The content of the source code in the file will not change until one `jomFile.commit(...);` method is called. `JomFile` should not be used after a commit on it is performed. A `JomFile` is intended to be a short-life object. You can use it to generate source code structures (`joms`) and commit the file. After this point, references to its `JomFile` (or any of its children) should not be kept around.

If the source code is modified by other thread/process on disk (or in the VFS buffer), access to `jom` objects in this `JomFile` will throw a `ConcurentModificationError`.

Example: JOM object from JAM alternative

An alternative way to obtain a JOM object from JAM is to use the `getJomXXX()` method on the relevant `jam` objects.

```
JomMethod  jomMethod = jamMethod.getJomMethod();
```

How JAM sees a class

JAM views a class in a manner similar to how the `java.lang.reflect` package does. A representation of a `Class` exists (either from source or from a compiled unit). This `Class` is composed of member items: fields and methods. In many cases, you'll find working with these higher-level items familiar if you've previously worked with the `Reflection` API. Reflection provides access to the methods, constructors, and fields of a class. You can glean information on the class itself by using `java.lang.Class` directly.

Relevant JAM objects

- **`JamClass`**
  `JamClass` has methods for discovering its superclasses, interfaces, methods and fields. The naming conventions are similar to the `Reflection` API in the JDK. For instance, to find out all the methods that are declared in the current class, use `getDeclaredMethods()`, but to find all methods including those of its ancestors, use `getMethods()`. This convention does not apply to constructors, since only the constructors that are declared in the current class are available. To get the declared constructors, use `getConstructors()`.
- **`JamMethod`**
  `JamMethod` provides information about the method you are interested in: return type, the parameter types and names, plus all the modifier information.

- **JamField**
  JamField provides information about the field you are interested in: type, name and modifiers.
- **JamType**
  Abstracts the notion of “type” and is used wherever type information is required in JAM and JOM. JamType represents both primitive types and Object types.

  The only public subclass of JamType is JamMethodType, which represents a method signature. All JamTypes are intended to represent fully qualified type information. JamType performs all conversions between qualified type names and signatures. Complex types are also represented by JamType, such as arrays, multi-dimension arrays, etc.

  Future changes to the language, such as the addition of generics and enumerated types, will be represented in JamType as well. Code written with JAM and JOM will be able to take advantage of these features, often without any changes.

How JOM sees a class

Using JOM, you can do all that and go much deeper than using reflection alone. JOM has constructs which wrap around the text in the source code, providing for an easy programmatic access to the source code. It also adds a JomFile object, which is the representation for the file that contains the source code. These detail items follow the language guidelines laid out on the Java Language Specification (JLS.)

- **File**: contains class(es) and items that aren’t directly a part of a class such as package and import statements. This is represented by JomFile.
- **Class**: contains fields, methods, and inner classes. This is represented by JomClass.
- **Method**: has a code block. This is represented by JomMethod.
- **Code blocks**: contains statements, other methods, and so on. It is represented by JomBlock and its contents are JomStatements or descendants of that class.

JOM continues where reflection stops by allowing you to work with the individual statements or expressions that make up the heart of most Java code.

Using JAM to read Java

Once a JamClass is obtained as previously described, it can be queried for methods, fields, superclasses and subclasses.

JAM introduces JamType which represents a primitive type or an Objec type. It also recursively represents arrays of a type. There are predefined types for the primitives and String, i.e. JamType.INT or JamType.String. The method asArray() returns a JamType for the type, for instance to get a type for int[].

You can create a type from text (i.e. full type name) or from the type signature (how it’s represented in the class files). Here are several ways of creating a JamType:

```
JamType type;

//Predefined types:
type = JamType.INT;
type = JamType.STRING;

//or specifying the type from the type name (long name for objects):
type = JamType.fromText("short");
type = JamType.fromText("java.lang.String");
```

```
//or from the class files signature:
type = JamType.fromSignature("Ljava/lang/String;");
type = JamType.fromSignature("B"); //Byte primitive

//creating arrays:
type = JamType.fromText("int[]");
type = JamType.INT.asArray();
type = type.asArray() //Whatever type is now type[]
```

A difficult problem that JAM addresses is finding methods. Methods have signatures as well as names. Class files deal with the signature of a method, and the compiler and sources deal with the parameter types. Both represent the same thing, but in a completely different way. JAM provides a special `JamType`, `JamMethodType`, which represents the method signature (not including the name).

```
JamMethodType methodType;

//Predefined types:
JamMethodType methodType = new JamMethodType();
//void x();

methodType.addParameter(JamType.INT);
//void x(int i);

methodType.addParameter(JamType.STRING);
//void x(int i, String s);

methodType.setReturnType(JamType.fromText("com.borland.sample.Bar");
//Bar x(int i, String s);
```

Once you have defined the method type, it's easy to find a method from a `JamClass`.

```
JamClass  jamClass  = factory.getClass("com.borland.jbuilder.jam.TestJam");
JamMethod jamMethod = jamClass.getMethod("x", methodType);
```

Finding a field is similar, except only the name is required:

```
JamClass jamClass = factory.getClass("com.borland.jbuilder.jam.TestJam");
JamField jamField = jamClass.getField("foobar");
```

Using JOM to read Java

JOM can parse Java source files. The methods specified by JOM for parsing Java use the naming convention `get<XYZ>` to extract whatever `XYZ` represents. These getter methods are generally used to parse the nested nature of Java code and usually return another item that is represented by a JOM.

During parsing, everything works from the outside in. At the outermost level (a `JomFile`), the parsing procedure is simple. A `JomFile` contains one or more classes (represented by a `JomClass`), and each of those classes is composed of variables and methods. Usually, you have a loop that iterates through all the classes in the file. From each of those classes, you can get an array of methods defined in that class and loop through each of these. The implementation of these methods makes things a bit more complicated. For example, a class can have many methods and each method can be composed of several statements, and each of those statements can be a method that is composed of many statements, and so on. At this level, recursion is common.

The base object in the JOM hierarchy is the `Jom` object. Every Java Language (JL) construct derives from this object. In order to remove any `Jom` you just need to call the remove method on it.

Every `Jom` also has an associated `beforeComment`/`afterComment`. (In order to obtain them you need to call the appropriate `getXXX()` methods.) The underlined `Jom` object of a

comment is JomComment. It can be JavaDoc, standard comment, or EOL comment. JomComment allows you to modify the comment text.

JOM has a very high level of granularity. Children of Jom objects (JomMethods for JomClass, JomStatement for JomBlock etc.) are created as lazily as possible. The child Joms are created on-demand (class=jomFile.getJom class=(<JomPosition>).) Another way to populate recursively all the children of a Jom(JomClass, JomMethod, JomBlock, JomStatement, etc.) is to call getDeepChildren() method of a JomContainer. Yet another way is to use the FillExistingChildrenVisitor object, as shown below:

```
...
Jom jomToFill = <some Jom>;

com.borland.jbuilder.jom.FillExistingChildrenVisitor fillVisitor
        = new com.borland.jbuilder.jom.FillExistingChildrenVisitor();

fillVisitor.visitJom(jomToFill);
...
```

Using JomContainer's getChildren class=(boolean includePendingRemovals) returns all the children. (Whether to include the one removed from the source is specified by the includePendingRemovals parameter.)

JOM allows you to parse down to the atomic level. Consider a simple example of a for loop:

```
public void forLoop(){

  // a bunch of statements in the method body...

  for(int i = 0; i < 5; i++){
    System.out.println("Hello Jom!");
  }

  // even more statements in the method body...

}
```

The for statement is defined in section 14.12 of the Java Language Specification, which states:

> "The for statement executes some initialization code, then executes an Expression, a Statement and some update code repeatedly until the value of the Expression is false."

In our example, the initialization code is: int i = 0. The expression is: i < 5. The statement that is executed is the println() inside the code block of the for loop. The update code is i++.

As the Java Language Specification states, a for loop is a type of statement. This means, at a high level, the for loop in the code would be parsed as being of type JomStatement. This is very generic as many things are JomStatements, so if you are working with a bunch of statements with JOM, you would determine if the type of statement you had encountered was really an instance of JomForLoop using instanceof.

JomForLoop breaks the for statement down even further. The expression portion of a for statement describes the condition that causes the loop to terminate. Any valid Java expression can be used here, ranging from a simple value check to a complex expression. JomForLoop.getCondJom() returns a JomExpression object that you could further interrogate to determine what type of an expression you had and what it was composed of, and so on.

Use the following code to see how a simple class is broken down and parsed with JOM:

```
package com.borland.samples.jom;

import java.awt.*;

public class JotExample extends Component{
   static final int PI = 4;

  public JomExample(){
    // xtor implementation goes here
  }

  public boolean doesItRock(int x){
    int ctr = x;

    for(int i = 0; i < ctr; i++){
      System.out.println("JBuilder Rocks!");
      System.out.println("Oh yes, it does");
    }

    return true;
  }
}
```

For this example, we have an instance of `JomFile` named `jsf` that represents this file. To get the package statement to use this code:

```
JomPackage pkg = jsf.getPackage();
```

To get the import statements from the file, use this code and iterate through the array:

```
String [] imports = jsf.getImports();
```

Note You can also use the `JomContainer` (such as `JomFile`) methods `getChildren(boolean includePendingRemovals)` and iterate through the children checking for `instanceof JomImport` to get the import statements.

To get the class(es) from the file, use this code:

```
JomClass [] classes = jsf.getClasses();
```

In this example, there's only one class file represented, so you know that your class is the first element in the array of classes returned by `JomFile.getClasses()`. Therefore, you can use this code to get the specific instance of `JomClass`:

```
JomClass jc = classes[0];
```

From the `JomClass` you can extract information about the class definition as well as its member fields and methods. The basic pattern remains the same; that is, a getter returns a value or object that you can interrogate further for more information.

```
// info about the class declaration
JomModifiers mod = class=jc.getJomModifiers();
JomExtends super = jc.findJomExtends();
JomTypeList imps = findJomImplements
```

These methods allow you to find out the modifiers for the class, the super class and the List of interfaces implemented by the class. You could get more information about the `superclass` by using the methods in JAM to get a `JamClass` for the `JomExtends` object.

Where modifiers are used in Java, as, for example a `public` class or a `static final` variable, JOM allows access through a `getJomModifiers class=()` method. `getJomModifiers class=()` returns an object, which provides methods for manipulating the modifiers on any JL construct.

If you continue with the parsing of the class, you can get a list of fields and a list of methods contained in the class. Again, you can further parse the results to get to the atomic portions of the fields and methods:

```
// details on the contents of the class
JomField  [] fields  = jc.getFields();
JomMethod [] methods = jc.getMethods();
```

For a more complete example of using JOM to parse source, see the sample `com.borland.samples.opentools.jom.read.ReadingSource`. This sample takes the selected file and gathers information about classes and methods using JOM. This information is sent to the message window inside the IDE.

An example of using JAM/JOM to parse source and/or classes is the sample `com.borland.samples.opentools.jom.packagetree.PackageTree`. This sample takes the selected node in the IDE, determines its package, and then builds a hierarchy tree for all the classes in that package.

Using JOM to write Java source

`Jom` models the source file and its contents. The starting component for `Jom` is `JomFile`. `JomFile` is obtained from the projects `JomFileManager`or from a `JamClass`, `JamMethod` or `JamField`. `Jom` is a high level representation of source code and provides many features to simplify the development of code.

`Jom` provides a set of advanced features for writing two way code, auto formatting, auto importing, comment support, references, renaming within file and position encapsulation. Also, `Jom` provides a complete mapping of language constructs to logical `Jom` constructs.

Some of the important JOM constructs are `JomFile`, `Jom`, `JomPosition`, `JomContainer` and `JomReference`.

- **`JomFile`**

  `JomFile` represents a single source file and provides a logical mapping of the source code. Source code can be safely inspected and modified. Retrieve a `JomFile` from the `JomManager` with an `Url` or fully qualified class name, or from a `Jam` that represents a class, method, or field in a source file.

  `JomFile` is a `JomContainer` which holds other `Jom`s. The two high level components in a `JomFile` are a `JomClass` and a `JomImport`. `JomClass` is the container for most other `Jom` types. When the modifications to a `JomFile` are complete, a commit writes the changes including (optionally) formatting, importing and adding Javadoc.

  A simple example of using `JomFile` is getting a class and then committing it. This simple program will create the class if it does not already exist:

  ```
  Url fileUrl = manager.getUrlFromClassName("root.subpackage.MyClass1");
  JomFile file = manager.getJomFile(fileUrl);
  JomClass jomClass = file.getPrimaryClass();
  ```

- **`JomPosition`**
  Contains the information about positions of `Jom`s within a source file. It translates between compiler positions, buffer positions and line/column positions. All `Jom`s have positions associated with them. `Jom`s can be tested if they are before or after other `Jom`s and using a `JomContainer`. A `JomContainer`is any `Jom` that contains another `Jom`'s classes, methods, blocks etc. `Jom`s can be used as position markers within other `Jom`s.

- **`JomContainer`**
  Holds other `Jom`s. Has methods for inserting before or after other `Jom`s and a general method for appending to the container.

- **`JomFactory`**
  This is a factory object used for creation of `Jom`s.

When writing source with JOM, you deal with much higher level items like methods or statements, instead of the very fine-grained items available to you when reading source files with JOM. The methods specified by JOM for writing Java code use the naming convention `append<XYZ>, insert<XYZ>, add<XYZ>` and `set<XYZ>` where `XYZ` represents the item you are creating or modifying. For example, `JomClass.getBody class=().append()` adds a child to a given class. `class=JomMethod.getJomModifiers class=().setFinal(boolean bValue)` sets the modifier's final bit for that method.

A convenience method in `JomFile` is `JomClass getPrimaryClass()`. This method returns the primary class of a file (the public class, the first top level class.) If the `JomFile` is empty (for instance, if just being created), this method will return a new `JomClass`, ready to be edited (using JOM) and later committed.

Notes JOM is very forgiving of being told to write bad code and will write anything you have told it to write.

JOM expects all the type strings (return types, parameter types etc.) to be fully qualified. The types and appropriate import statements will be automatically generated based on the user code generation properties.

Example: writing code using JOM

Let's create the same class that was used in the parsing section, but we'll build it up in smaller chunks. This simple "starting point" of the class looks like:

```
package com.borland.samples.jom;

import java.awt.Component;

public class JomExample extends Component{

}
```

Again, we begin with the outermost item, the source file itself. For JOM, the source file is represented by a `JomFile` that encapsulates the package statements, the imports, and the class declaration.

Assuming you have an instance of a `JomFile` named `jsf`, use the following methods to create the class:

```
jomClass = JomFactory.createJomClass("com.borland.samples.jom.JomExample",
    jomFile);

JomExtends jomExtends = jomClass.getJomExtends();

jomExtends.setName class=("java.awt.Component"); // NORES

JomModifiers mod = class=jomClass.getJomModifiers();

mod.setPublic class=();
```

To finish, you need to add the rest of the content (a member variable, a constructor and a method) to the class. This is what the code that is generated will look like:

```
package com.borland.samples.jom;

import java.awt.Component;

public class JomExample extends Component {
  static final int PI = 4;

  public JomExample() {
    // xtor implementation goes here...
  }
```

```
  public boolean doesItRock(int x){
    int ctr = x;

    for(int i = 0; i < ctr; i++) {
      System.out.println("JBuilder Rocks!");
      System.out.println("Oh yes, it does");
    }

  return true;
  }
}
```

To add the field, use code like this:

```
JomField jf = jcs.getBody().append(JomFactory.createJomField("PI",JamType.INT);

jf.setInitialValue class=("4");
```

To make the field PI both `static`and `final`, do this:

```
jf.getJomModifiers class=().setStatic(true);
jf.getJomModifiers class=().setFinal(true);
```

To create the constructor and make it public, do this:

```
JomConstructor xtor = jcs.append(JomFactory.createJomConstructor(jcs,
    JomParameterList.EMPTY));

xtor.getJomModifiers().setPublic();
```

To set up the “shell” of the method with a parameter `x`, use this code:

```
JomParameterList parList = JomFactory.createJomParameterList();

JomParameter parameter = JomFactory.createJomParameter("x", JamType.INT);

parList.append(parameter);

jomMethod = JomFactory.createJomMethod("doesItRock", parList, JamType.BOOLEAN);

mod = jomMethod.getJomModifiers();

mod.setPublic();

jomClass.append(jomMethod);
```

Add the local variable `ctr`and assign it the value of the parameter passed to the method:

```
JomVariable jv = JomFactory.createJomVariable("ctr", JamType.INT);

jv.setInitialValue class=("x");

jomMethod.append class=(jv);
```

Add in the `for` statement. There are two ways to create the `JomForLoop`:

- You could create the specific statement (`JomForLoop`) and fill its children `Jom`s appropriately, or
- You could simply generate all the code in new, arbitrary `JomStatement` objects and add it to the method. (Note that `JomText`, `JomStatement` and `JomExpression` can be used to hold any text that needs to be generated in the source code. If the text has more than 255 characters in a row, each row over 255 characters must be broken into smaller rows by `JomCommitter.NEWLINE`.)

In a `JomStatement`, you can put any arbitrary string and it will be generated in the source code. In the following `JomStatement` object, we are actually generating two statements: The `ForLoop`and the `return` statement:

```
StringBuffer statementText = new StringBuffer();

statementText.append("for(int i = 0; i < ctr; i++) {");
statementText.append(JomCommitter.NEWLINE);
statementText.append("System.out.println(\"JBuilder Rocks!\");");
statementText.append(JomCommitter.NEWLINE);
statementText.append("System.out.println(\"Oh yes, it does");");
statementText.append(JomCommitter.NEWLINE);
statementText.append("}");
statementText.append(JomCommitter.NEWLINE);
statementText.append("return true;");

jomMethod.appendclass=(JomMaker.createJomStatement(statementText.toString()));
```

For a more complete example of using JOM to write source, see the sample `com.borland.samples.opentools.jom.write.WritingSource`. This sample generates a simple class file. Another sample of using JOM to read and write is `com.borland.samples.opentools.jom.readwrite.Commenter`. This sample takes the selected source node and adds `Javadoc` comments for top level classes and methods in classes. It only adds them if there isn't a comment there already.

Using JAM and JOM together

Use JAM to discover the main construct you are interested in. Once you have the appropriate JAM object, you can directly convert to a JOM for manipulating source code.

For instance, assume you have a method named `bar` that takes an `int` and returns a `String`in the class `com.foobar.Foo`. There are 4 simple steps:

1. Get the `JamFactory` for the project.
2. From the factory, get the `JamClass`.
3. Ask the `JamClass` for the `JamMethod`, passing in a `JamMethodType` and the method name.
4. Then get the `JomMethod` from the `JamMethod`.

Notice that JAM deals in logical constructs entirely—you don't need to know where files reside on the source path, just the full class name. `JamType` and `JamMethodType`also simplify the process of defining the method you are searching for.

```
Project project; // assume a valid, non-null instance of Project

JamFactory factory = class=JamFactory.instance(project);

JamClass jamClass = factory.getClass("com.foobar.Foo");

JamMethodType type = new class=JamMethodType();
```

```
type.setReturnType class=(JamType.STRING);

type.addParameter class=(JamType.INT);

JamMethod jamMethod = jamClass.getMethod("bar", type);

JomMethod jomMethod = jamMethod.getJomMethod();
```

Chapter 17

JBuilder Server concepts

JBuilder 4, 5 and 6 had support for both App Servers and Web Servers as two entities, with the App Server containing a reference to the related Web Server. In JBuilder 7, we streamlined this support so that an App Server can be both an EJB container service provider *and* Web container service provider. JBuilder X saw significant cleanup and changes to correspond to new Module/DD Editor support. JBuilder 2005 migrated to JOM and streamlined some methods, particularly those relating to JARs.

The AppServer and WebServer features in older versions of JBuilder evolved on two totally different tracks: WebServer and AppServer. JBuilder now has an integrated and modular server API. This has several benefits:

- Easier to implement plugins for new servers. Instead of having to learn two different APIs, there is just one. This benefits both internal programmers, reducing time-to-implement, and can encourage external programmers to implement their own plugins.
- The modular architecture separates the server from its services, which allows individual services to be skipped for a particular server, added later, and for entirely new services to be added after the version has shipped (as addons, not patches).
- The unified UI provides a better end-user experience.
- Multiple services hosted by the same server can run in the same VM. This is both a convenience (because one VM uses less memory and has less startup/shutdown time than two) and a requirement for some servers to have their containers communicate in the expected manner.

Features and subsystems

- Server — a particular version of a server, such as BES 5.2, WebLogic 8.1, or WebSphere 5.0
- Service — a feature supported by the server, usually in the form of a container that is hosted in a running server or command-line utility. For example: EJB, Web, JMS, Deployment
- Plugin — classes written to the Server API to enable a particular server and its services to operate inside the JBuilder IDE

We have added a generic `Server` concept, which becomes a base class for an app server and a freestanding web server. This allows us to treat the different kinds of servers similarly in situations where we would need to show properties for them and list them in a common panel. Additionally, we combined the Web and EJB run concepts so that we would only start one instance of a server that supports both kinds of containers. The design of this subsystem allows any other kinds of services (like JMS for instance) to be added to the list of services started up when the Server runs/debugs.

Architecture

Core API

The core API lives in the `com.borland.jbuilder.server` package. The `ServerManager` is an OpenTool in the `Node` category that, in turn, initializes the categories `ServerServices` and `Servers`, in that order.

Each server plugin is a concrete implementation of the abstract base class `Server`. These `Server` instances register themselves with the `ServerManager` as part of their OpenTools startup by calling `ServerManager.registerServer()`.

The `ServerManager` will call back to `Server.registerServices()` so that the server can register its known `Service` implementations, through `ServerManager.registerService()`. Thus, services are separated from servers, and services can be bound to servers any time after the server itself is registered.

The abstract base class `Service` describes common aspects of all services. It contains an `abstract static` inner class named `Type`, which is used to identify the service itself, including its name and SKUification.

Each service subclasses `Service`, but is still an abstract class; it concretes the `Type` inner class for that service, and declares all the abstract methods to support that service. Each `Service` subclass registers an instance of its `Type` through `ServerManager.registerServiceType()` during its OpenTools startup. Each server then concretes the `Service` subclasses for the services it implements.

A `Server` also acts as a factory for `ServerLauncher` objects, which interface with the runtime system. There are two abstract subclasses, `NativeServerLauncher`, for servers that are hosted by native processes (binaries or scripts), and `PureJavaServerLauncher`, for "regular" Java-based servers. Note that native servers eventually must have a Java core in order to execute Java code such as servlets and EJBs. The `Server` must provide a concrete implementation of either one of those to actually start the server.

For example, `EjbService`, `JspServletService`, and `NamingService` are all abstract subclasses of `Service`, and they're all listed under the OpenTool category `ServerServices`. Each one has a concrete type: `EjbService.Type`, `JspServletService.Type`, and `NamingService.Type`. In the `initOpenTool()` for those classes, instances of those `Type` objects are registered. `Server` implementations like `BES50Server` and `Tomcat40Server` are listed under the `Servers` category. When they register, their `registerServices()` callback will then register their concrete Service implementations: `BES50Server` would register `BES50EjbService`, `BES50JspServletService`, and `BES50NamingService`; `Tomcat40Server` would register `Tomcat40JspServletService` and `Tomcat40NamingService`. BES 5.0 uses the native `partition.exe`, so its `BES50Launcher` is a subclass of `NativeServerLaucher`, while Tomcat's `Tomcat40Launcher` is a subclass of `PureJavaServerLauncher`.

Legacy Adapters

Legacy adapters are designed to wrap existing `AppServer` and `WebServer` plugins in the new Server API. This minimizes internal conversion work and allows external plugins to continue to operate under the new API. The `AppServerManager` and old web server `ServerManager` (different from the new `ServerManager`) construct new `LegacyServer` objects around the `AppServer` and `ServerSetup` objects they register, and register them under the new API.

Important This legacy mechanism is in place so that the pre-JBuilder 7 plugins are still able to function. **The old API is deprecated.** When the relevant plugins written in that API are updated or deprecated, the legacy mechanism will go away.

The `LegacyServer` supports the services `LegacyEjbService`, `LegacyJspServletService`, `LegacyClientJarService`, and `LegacyDeployService`. Depending on whether the wrapped object was a web server `ServerSetup`, or an `AppServer`, or an `AppServer` with a `ServerSetup` (the old "app server is web server"), different services are enabled.

Note For information on server configuration, see "Server Configuration concepts" on page 154.

Project-level Server/Service configuration

The Server/Services for the project can be selected on the Server page of the Project Properties dialog box. The page has two radio buttons on top to clearly differentiate between the two approaches: the simpler situation where one server does everything, and the more complex case where you can have separate servers for each service (or any combination). The Single Server For All Services In Project combo box lists all registered servers, from which the user can quickly select a server. In the drop-down list, any servers that are not properly configured are shown in red. The ellipsis (…) button to the right opens the Configure Servers dialog box.

The list of Services on the left of the Server page always shows all registered service types. If the chosen server does not support a particular service, that service is grayed out. The checkbox next to the service name controls whether the service is enabled for this project or not. If a service is not supported, the service is disabled and the checkbox state doesn't matter, but it is maintained in case the user decides to switch to modular services or back. Tooltips for the items in the tree explain more about the item's current state, e.g., a service is disabled because the server doesn't support it (single server mode) or no servers support it (modular services). Selecting a service shows the configuration information for the current server's implementation for that service.

Selecting the Modular Services Provided By Different Servers radio button disables the single server combo box and button. These two sets of UI perform the same basic function, but are mutually exclusive.

Only one server can be selected for a given service. The name of the server and its information panel is shown on the right for whichever server/service currently has focus, which can be different from the server that has been selected to provide the service. As with single server mode, selecting the service node (e.g., EJB) shows the information for the currently selected server.

Changes in JBuilder 2005

JOT to JOM migration

The following methods have been deprecated and will be removed as part of the migration from JOT to JOM:

- `AppServer.createClientGetInitialContext(JBProject jbProject, boolean hasLogging, JotClassSource classSource)`
- `AppServer.createClientGetInitialContext(JBProject jbProject, boolean hasLogging)`
- `EjbService.createClientGetInitialContext(EjbTestClientState client, JotClassSource classSource)`
- `EjbService.createClientGetInitialContext(JBProject jbProject, EjbModuleNode ejbModuleNode, boolean hasLogging, JotClassSource classSource)`

- `EjbService.createClientGetInitialContext(JBProject jbProject, boolean hasLogging)`
- `EjbService.createClientGetInitialContext(JBProject jbProject, EjbModuleNode ejbModuleNode, boolean hasLogging)`
- `EjbService.getInitialContextMethodName()`

The following methods should be used instead:

- `NamingService.createClientGetInitialContext(JBProject jbProject)`
- `NamingService.getInitialContextMethodName()`

ServerCommandLineTool support

Previously, if server tool implementors wanted to add external tools to be accessible from JBuilder for their server, they had to add them to the Tools menu using the `CmdLineTool` mechanism. We have now added a new `ServerCommandLineTool` class, which actually extends the `CmdLineTool` class, but it allows us to treat tools for servers in a special way, making server tools easier to add (hopefully) and use.

All `ServerCommandLineTool` implementations will appear in their own server section under the Enterprise menu on the JBuilder main menu.

This class allows you to specify various restrictions on your tools — for instance, you can specify whether you want your tool to appear only when the server is configured (the default) or not. To use this class, you will need to do the following:

- Provide `ServerCommandLineTool` implementations for your external tools.
- Add an implementation of the `Server.addTools()` method to actually add the tools.

Some notes:

- The current implementation tries to take care of removing previous incarnations of tools with the same title as `ServerCommandLineTool` implementations from the Tools menu. However, if your tools are conditionally added, based on a property defined in the Server configuration, for instance, you will need to delete the tool yourself in the `addTools()` implementation if the property is set to `false`, to take care of the case when the menu is first opened before any configuration takes place. So your code would look something like this:

```
addTools(Server server) {
  if (myProperty.getBoolean(server)) {
    addTool(new myTool(server));
  }
  else {
    deleteToolByTitle(MyTool.TOOL_TITLE);
  }
}
```

  If you didn't have previous Tools menu incarnations of the tool, you don't need to worry about deleting the tool — it will happen for you automatically in the implementation of the `addTool()` method.

- You only need to provide a single constructor for your `ServerCommandLineTool` implementation class, which takes a `Server` parameter, and calls the `super(server)` implementation.
- Currently the tools will appear regardless of whether their servers are applicable to the current project, and they will appear in subgroups per server in the Enterprise menu. There are hidden properties to optionally only enable the tools for the servers in the current project, and to show the tools in pull-right menus for each

server under the Enterprise menu. We can see which implementation is most intuitive for the final version of the implementation.

- The legacy AppServer has methods corresponding to the Server methods for `ServerCommandLineTool` manipulation so that legacy API implementors can also take advantage of this functionality.
- You no longer need to add/remove tools in the Configuration/Setup pages for your server — just the `addTools()` implementation in the Server/AppServer class is enough, unless your Server/AppServer tools have specific needs and need to be added/removed in a particular way. Then you can just call the Server methods for `ServerCommandLineTool` management directly.

Various method changes

DeployService.compileAndDeployArchives()

This method now has an additional parameter — `ArrayList selectedNodes`, which is a list of the actual nodes selected for deployment from the project. The previously and still existing `deployNodes` parameter is a list of nodes that can actually be deployed, e.g. archive nodes, whereas the newly added `selectedNodes` list contains the actual ModuleNodes and/or archive nodes selected from the project for deployment.

ClientJarService removal

The `ClienJarService` really provided one useful method — `createClientJar()`, which would enable the EnterpriselCreate EJB Client JAR submenu and call that method when the menu is selected. It was confusing to have that service, since it implied that the service was connected to Application Client module nodes. This is not the case — application client module functionality is separate from this service; an application client module contains specific descriptors and data for that module type, but not necessarily all the stubs and skeletons for communicating with another client module, whereas the EJB client JAR just contains the necessary files (stubs/skeletons) for the client to communicate with an EJB.

The functionality of this service has been moved to the EJB service. Now to replicate the `ClientJarService` behavior, you would need to implement 2 methods in your `EjbService` implementation:

- `supportsClientJarCreation()`
  By default returns `false` — implement to return `true`, if you have that functionality.
- `createClientJar()`
  Should have the same content as the `ClientJarService` method of the same name. This method will be called from the “Create EJB Client Jar” menu, if the `supportsClientJarCreation()` method returns true.

Changes in JBuilder X

In JBuilder X, significant cleanup and changes were performed in the Server API, to correspond to new Module/DD Editor support. Below are descriptions of the various areas that have been changed.

New Module support

See Chapter 18, “JBuilder Module concepts” for general `ModuleNode` behavior descriptions.

New ModuleNode subclasses

The previously used `EjbGrpFileNode`, `EarGrpFileNode`, `WebContextNode` have now been replaced with extensions of the `com.borland.jbuilder.enterprise.module.ModuleNode` class. `EjbGrpFileNode` is now replaced with `com.borland.jbuilder.enterprise.module.ejb.EjbModuleNode`, `EarGrpFileNode` is replaced with `com.borland.jbuilder.enterprise.module.application.ApplicationModuleNode`, and `WebContextNode` has been replaced with `com.borland.jbuilder.enterprise.module.web.WebModuleNode`. Additional `Connector` and `ApplicationClient ModuleNode` extensions have been added in the corresponding packages (using the same scheme as above). New modules are used to provide some consistency between our support for all J2EE modules. You will see this when you try to use the wizards for the new modules. They will start out looking the same, but might have some additional pages for each module type.

Each `ModuleNode` subclass will have a supporting `<module name>ModuleType` class. For example, `EjbModuleType` is the `ModuleType` for the `EjbModuleNode` class. There is an accessor for it in the `ModuleNode` subclass. Additionally, most of the time there will be a `<module name>ModuleUtils` class in the package for that module. For example, `EjbModuleUtils` is a supporting class for `EjbModuleNode`, and contains various useful methods (many of which are mapped from the previous `EjbGrpFileNode` usage).

Another useful class is `com.borland.jbuilder.enterprise.module.ModulePropertyGroup`. This class has various utility methods for dealing with `ModuleNodes`. For instance, to get all the `EjbModuleNodes` in a project, you can call `ModulePropertyGroup.getModuleNodes(project, EjbModuleType.getModuleType())`. There are various overloaded versions of this method in that class, as well as other useful ones.

All the `Server` API and legacy `AppServer` API methods and classes that previously used `EjbGrpFileNode` and other legacy node types (`WebAppNode`, `EarGrpFileNode`) as parameters or members have been converted to use the new `ModuleNode` extension equivalents. You will need to update these method definitions in your `Server` API or `AppServer` API implementation.

Obsolete classes

AppServerTargeting

The `AppServerTargeting` class is no longer used in the `Server`/`AppServer` API. It has been replaced with a combination of additional `DeployService()` methods and the new DD Editor support.

All the build related methods from `AppServerTargeting` have been moved to the `DeployService` class for the `Server` API, and to the `AppServer` class for the `AppServer` API.

To convert your `AppServerTargeting` implementation to the new build usage, you would need to do the following:

1. Move the implementation of `AppServerTargeting.preProcessBuild()` to `DeployService.preProcessBuild(ModuleNode)`, if you are using the `Server` API, or to `AppServer.preProcessBuild(ModuleNode)`, if you are using the legacy `AppServer` API.
2. Move the implementation of `AppServerTargeting.postProcessBuild()` to `DeployService.postProcessBuild()`, if you are using the `Server` API, or to `AppServer.postProcessBuild()`, if you are using the legacy `AppServer` API.
3. Move the implementation of `AppServerTargeting.updateBuildTask()` to `DeployService.updatePostCompileBuildTask()`, if you are using the `Server` API, or to `AppServer.updatePostCompileBuildTask()`, if you are using the legacy `AppServer` API.
4. Implement `DeploymentDescriptor`-related methods and general deployment descriptor support using the new XMT API. You will basically need to define extensions of `XmtReader` and `XmtWriter` classes for reading and writing your custom

descriptors. You will also need to implement other classes using the XMT API for displaying GUI viewers for your descriptors. See Chapter 7, “XMT concepts” for more details.

AbstractDeploymentDescriptor/DeploymentDescriptor

The `AbstractDeploymentDescriptor` class and its subclasses will no longer be used, since the new XMT API will be replacing them. If you have implementations of these classes, you will need to, instead, define a new implementation using the XMT API. See Chapter 7, “XMT concepts” for more details, and remove your current implementation.

AbstractDescriptorImporter

Since the new `ModuleNode` subclasses define their descriptors to be stored directly on disk and not inside another file, as they were, for instance, for `EjbGrpFileNode`, there is no need for an importer implementation anymore. Instead, each `ModuleNode` wizard provides a way to create a `ModuleNode` from an existing directory or archive, and, when you switch between servers for your project, new descriptors associated with the newly selected server will be written for each module node in the project (and previous ones deleted, currently) automatically. You should remove your implementations of this class, if you have them.

Other Changes

- `com.borland.enterprise`, `com.borland.console` package usage has changed. Any methods in the `Server`/`AppServer` API that required classes from the `com.borland.enterprise`, `com.borland.console` and their child packages have either been deleted or replaced with equivalent methods in the new DDEditor/XMT APIs. We will no longer be using classes from the BES DDEditor jars.
- `com.borland.jbuilder.enterprise.ejb.DeployConstants` has been removed. Use `com.borland.jbuilder.server.DeployService` constants instead.
- `<legacy module>PropertyGroup` methods/constants/properties have been moved to the `<new module>NodeUtils` or `<new module>Node` class. If you are having trouble finding various methods, constants, or properties that used to live, for instance, in `WebModulePropertyGroup`, try looking in the new `WebModuleUtils` or the `WebModuleNode` classes. For `WebModuleNode` specifically, various utility methods that used to live inside that class have been translated to versions that can be applied to all module types, and you can look for their newest incarnations in the `ModulePropertyGroup` class.
- `ContextDescriptor` and `ServletDescriptor` are no longer used. Any place where these classes were used, they will be replaced with the new `WebAppWrapper` and `ServletWrapper` XMT equivalents.
- Web support’s `DeploymentDescriptorSupport`, `WebXmlSupport`, `DescriptorWriter`, and `DescriptorReader` are no longer used. These should be replaced with the new XMT `XmtReader` and `XmtWriter` implementations.

See also

- “Server Configuration concepts” on page 154
- Chapter 18, “JBuilder Module concepts”
- Chapter 19, “JBuilder Enterprise Setup dialog concepts”
- Chapter 7, “XMT concepts”

Server Configuration concepts

In order for a server to be available for use in JBuilder, it first has to be configured. JBuilder 7 introduced the concept of Server Configuration. This can be thought of similarly to the Library and JDK configuration concepts. There is now a Configure Servers dialog box, where all the servers that JBuilder supports are listed and the user can select a server to configure. When OK is clicked in this dialog, all modified servers are saved.

To support the “Configure...” paradigm, server configurations are now stored in library files, similar to the JBuilder library and JDK definitions. The server library files are saved in the home directory (`.jbuilder<jbuilder version>` under the user home). The library file name has the format `<server fullname>.library`. These library files contain information common to all servers. Server-specific information is saved in user properties.

When a `Server` OpenTool implementation registers with the `ServerManager`, the server will become part of the list of servers in the Configure Servers dialog box.

See also

- “JBuilder Server concepts” on page 147

Server registration

When a `Server` plugin is registered with the `ServerManager` class, the `ServerManager` will try to find an existing server library file with the previously saved settings for this `Server`. If it doesn’t find one, it will use the `Server`’s default settings when the plugin is registered. A valid `Server` OpenTool registration will cause this `Server` object to appear in the Configure Servers dialog box. For legacy support, if a valid `AppServer` implementation gets registered with the `AppServerManager` class, it will also appear in the Configure Servers dialog box. This is transparent to the user.

If a server library was found, but no `Server` implementation was registered for this library, there will be a red item for this server in the Configure Servers dialog box. This item will be available for deletion (as opposed to valid registered server libraries, which cannot be deleted).

If a server item appears in gray in the dialog, it means that it has not been configured yet. A server is considered configured if the user successfully saves settings for that server by selecting OK in the Configure Servers dialog box.

General Page (Configure Servers dialog box)

The General page contains all the information that is common between servers. This is what gets stored in the server library file. Basically, what you are seeing in this page are the `ServerPathSet` values. If you have the legacy `AppServer` implementation, you will see pretty much what used to be saved for that server in the legacy `appserver.properties` file. You don’t need to implement any extra classes for your `Server` to have a General page in the Configure Servers dialog box.

Custom Page (Configure Servers dialog box)

The Custom page is used to configure any settings that aren’t covered by the basic `Server` settings. It is optional for a `Server`; however, most servers will probably want to have one, since there is always something special that a particular server needs so it can be configured to work properly.

The Custom page will appear for a server if a `CustomConfigurationPageFactory` implementation is registered with the `ServerManager`.

The `CustomConfigurationPageFactory` is responsible for returning a valid `CustomConfigurationPage` implementation for the `Server`. The `CustomConfigurationPageFactory` can be registered as follows:

```
public static void initOpenTool(byte majorVersion, byte minorVersion)
{
 ServerManager.registerCustomConfigurationPageFactory(
  new MyServerCustomConfigurationPageFactory(),
  MyServer.SERVER_NAME,
  MyServer.SERVER_VERSION);
}
```

The `createCustomConfigurationPage()` method needs to be implemented to return a valid `CustomConfigurationPage` as follows:

```
protected CustomConfigurationPage createCustomConfigurationPage(
 BasicConfigurationPageAccess
 basicConfigurationPageAccess)
{
return new MyServerCustomConfigurationPage(
 basicConfigurationPageAccess);
}
```

The `createCustomConfigurationPage()` method will be called when the server is first selected in the Configure Servers dialog box. Once created, this page will be preserved while the dialog box is up, so it doesn't have to be created again.

The `BasicConfigurationPageAccess` class is a way for the `CustomConfigurationPage` to access the settings on the General page that correspond to this server. To get to the passed-in reference from your implementation of `CustomConfigurationPage`, use the `getBasicConfigurationPageAccess()` method. This will allow you to get the latest values for the basic settings that the user entered.

The `CustomConfigurationPage` can respond to various actions that happen on the General page. For example, when the Home directory gets modified on the General page, the `homeDirectoryModified()` method of `CustomConfigurationPage` will get called, so you can perform any necessary updates of the Custom page controls based on the new home directory. You can also go the other way, and update values of some of the basic controls on the General page, if you need to, by implementing the `checkUpdateBasicConfigurationPage()` method. This method will get called whenever the user selects the General page after having selected the Custom page.

Legacy support

Previously, servers used to be set up using the Enterprise Setup dialog box. To do this, OpenTool developers would have to implement a version of a `Setup` and an `AppServerSetupPropertyPage`. This would cause the property page to appear in a separate tab under the AppServers tab in the Enterprise Setup dialog box.

The AppServers tab no longer appears in the Enterprise Setup dialog box. This concept is now replaced with the Configure Servers dialog box.

Previously, server configuration information used to be saved in the `appserver.properties` file in the `.jbuilder<jbuilder home>` directory under the user home directory. Now, since we are using the "Configure..." concept, each server configuration is saved in its own `<full servername>.library` file.

We realize that there might be existing implementations of the `Setup` and `AppServerSetupPropertyPage` classes for existing `AppServer` plugins (in fact, we still use some of these). To help developers transition more easily to the new `Server` API, we

have developed a legacy support system, which tries to translate as much of the previous API implementation as it can to the new API.

See also

- Chapter 19, "JBuilder Enterprise Setup dialog concepts" for more on the Enterprise Setup dialog

Legacy conversion

If you already have an implementation of a legacy `Setup`, which was registered with `AppServerSetup`, this will attempt to use the legacy conversion mechanism to create a Custom page corresponding to your `AppServerSetupPropertyPage`. For this support to work properly, we have extended the `AppServerSetupPropertyPage` to have most of the functionality of the `CustomPropertyPage`. There are a few things that the legacy conversion mechanism will try to do to make the corresponding Custom page correspond to the Custom page that would be appropriate for a `Server`. For instance, the description text that appeared on top of the `AppServerSetupPropertyPage` will be suppressed. This text comes from the implementation of the `getDescription()` method in the `AppServerSetupPropertyPage`. This text is no longer necessary.

You will need to modify a few things in your implementation of `AppServerSetupPropertyPage` to make sure it works with the new Configure Servers dialog box.

Configuration

- Make sure you don't show the installation directory controls. The only place that the user should be able to set a home directory for the server is on the General page. You can just suppress these controls, if you need to.
- If you don't see a Custom page showing up for your server, and you did register an `AppServerSetupPropertyPage` for it, it probably means that the name you used for your `Setup` was not one we can use to convert your page to the new one. The `getName()` implementation for your `Setup` should return the full long or full short name of the `AppServer` you are registering with. Look at the `Server` class documentation for an explanation of the short name and long name references. Basically, the full long or short name should include the name string and the version string. For the short name, the name portion is shorter.
- Check out the new methods in `AppServerSetupPropertyPage` that are implementations of its new base interface, `com.borland.jbuilder.server.BasicServerConfigurationPageBridge`. Among these are various useful notification methods you might want to be aware of when converting your Setup page (like `homeDirectoryModified()`, for instance).

Non-configuration

- (New in JBuilder X) With the addition of support for a new DDEditor, we have removed the `AppServerTargeting` class and for the legacy support, moved the relevant methods to the `AppServer` class. If you have overridden various methods in `AppServerTargeting` class previously, look for them in the `AppServer` class.

From JBuilder 9

In JBuilder 9 we introduced the concept of "Copy Server". The Configure Servers dialog box now has an additional button, Copy, which allows users to create a copy configuration of a particular server. This copy allows the user to define a server configuration that has all the behavior of a particular supported JBuilder Server, but allows them to have more than one configuration for that `Server` type. For example, a user might want to have two different WebLogic configurations, one of which is

installed in one directory, and the other in a different directory. This copy `Server` information gets saved in its own `.library` file, and can be used as a server for a project. It will appear in the list of available servers on the server drop-down lists in the Server page of the Project Properties dialog box.

In order for a `Server` plugin to be copied, the `Server` (or legacy `AppServer`) has to be considered copyable by the `Server` API. For this to be the case, a few things need to be done. These are all listed in the Javadoc for the `Server.supportsCopy()` method. Once these are all done, override the `supportsCopy()` method in your `Server` (or `AppServer`) class implementation to return `true`. Its default implementation is `false`, so that you can be aware of all the changes that need to be made to your plugin to support copying a server. For more details on JBuilder's copy server functionality, see JBuilder's Help.

Chapter

18

JBuilder Module concepts

This document describes the concept of a `Module`, added in JBuilder X. Changes for JBuilder 2005 are discussed on page 163.

JBuilder has had the concept of special nodes to represent various J2EE module types. All these modules need to be able to be built into an archive specific for that module, so that archive can be deployed onto a J2EE server. They also all have associated deployment descriptors that need to be edited by the user before the final archive gets built.

Specifically, we have had the concept of a `WebContextNode` to represent Web modules, an `EJBGRPxFileNode` to represent an `EJBModule`, and an `EarGrpFileNode` to represent EAR modules. All the J2EE modules seem to have similar needs, but different implementations. It is difficult to support still more module types unless we have a unified way to represent the module concept.

For JBuilder X, we decided to come up with a generic `Module` concept to represent all the module types currently supported in JBuilder and J2EE, and allow for easier addition of new `Module` types. This new `Module` uses the directory concept for representing its associated files, because it seems like it is convenient to have your files directly exposed on disk, in case you might want to access them outside of JBuilder without having special viewers for special file types (such as an `EJBGRPxFileNode`).

This also allows us to provide support for deploying an “exploded archive” to servers that support such a concept. All the files that would be in an archive also reside in a directory, so they can be accessed directly, and the archive structure is mirrored in the associated directory for convenience.

Additionally, since a new, more generic `DDEditor` implementation was added in JBuilder X, allowing editing deployment descriptors for any kind of module, it is convenient to have a generic `Module` concept that can be used to connect to the new `DDEditor` in a generic way.

ModuleNode

The `Module` is represented by a `ModuleNode`. A `ModuleNode` is a subclass of an `ArchiveNode`, because it will be used in an archiving process. This `ModuleNode` is the base class for various J2EE module types that we support (i.e. `EJBModuleNode`,

WebModuleNode, ApplicationModuleNode, ConnectorModuleNode, and ApplicationClientModuleNode). This ModuleNode has two children: a node representing the module's descriptors, and a node representing the module directory. The URL's for these can be accessed by calling moduleNode.getDescriptorsDirectoryUrl() and moduleNode.getModuleDirectoryUrl().

The module directory is a directory that a user chooses when creating the module using the associated module wizard. By default, it has the same name as the user chooses for the module, and is located under the main project directory. For example, if the user chose to create an EJBModule called ejbModule1, its associated directory will be <project directory>\ejbModule1. It will also have an associated archive URL; in this case, by default, this would be <project directory>ejbModule1.jar. This is configurable by the user, and its URL can be retrieved by calling moduleNode.getArchiveUrl().

The descriptors node represents the descriptors subdirectory in the module directory, and will only show the descriptors that are defined to be valid for the module. For example, if we were creating an EJBModule, its descriptors would need to live in a META-INF directory in the final archive, so the descriptors node would point to the <project directory>\ejbModule1\META-INF subdirectory. The module directory will show all files that are put in that directory by either our generation processes, or by the user, unless these are specifically filtered out by the file types for the module on the main module property page.

Every time the ModuleNode is built, the files that are defined to go into the module's archive are also copied to the module's directory, if that is the option specified. This can be specified by the user by checking Synchronize Module Directory With Archive Content on the module's Build property page. The default setting is *on*, meaning the module directory will be synchronized with the archive content. This setting can be retrieved by calling moduleNode.getBuildExplodedArchive().

Note It is important to note that the module directory content should be considered as current as the last build.

It is optional for a ModuleNode to create an archive when it builds, or to copy files to the associated directory. A build can happen without actually updating either one. Based on the type of module to be built, or the server that is currently chosen to support the project, the defaults for these settings might be different. For example, for Tomcat, it's not necessary to build an archive; the module directory just needs to contain the necessary files.

In general, there are quite a few places where a ModuleNode will defer certain decisions to the currently used Server, or actually the appropriate Service in the current project. For example, anything that needs to be customized for deployment for a Server that's associated with a ModuleNode, is deferred to a method in the project's current DeployService. If you look at DeployService, you will see methods like getDefaultBuildExplodedArchive(), which allows the current DeployService to customize the decision for the default for this setting.

ModuleType

Each ModuleNode needs to be associated with a ModuleType. A ModuleType defines the behavior of a type of ModuleNode. This class is necessary because there are various situations where we need to take certain actions based on the behavior of modules that might exist, but don't yet, or else the behavior needs to apply to all types of a module as opposed to a specific one. For these situations, we can use the ModuleType.

The ModuleType is also the class that can be used to create an actual instance of an associated ModuleNode. For example, the EJBModuleType knows how to create an EJBModuleNode. This can be done by calling either the moduleType.createModuleNodeFromDirectory() methods (if you are creating a module node that is based on existing content), or by calling the

moduleNode.createModuleWizardContext(), and then using the resulting ModuleWizardContext (specific to each ModuleNode) to create an empty ModuleNode.

ModuleTypes are singletons, and can be retrieved by calling <YourModuleType>.getModuleType().

Defining a ModuleType for a module also allows us the flexibility to deploy modules by that type to different servers. So now, on the Project Properties|Server page, when you select Modular Services Provided By Different Servers for your project, you will notice that the Deployment Service item is an expandable node. The contents of that node are the different registered ModuleTypes. In the right-side panel, you can select a different server for each of the available ModuleTypes, if you so desire. This allows you, for instance, to deploy your EJBs to one server, and your Web applications to another, if you are actually using two separate servers for your EJB and Web containers.

Creating a ModuleNode

The basic way to create a ModuleNode is to go through the ModuleWizard associated with that node's ModuleType. The ModuleWizard uses a ModuleWizardContext specific to each ModuleType to get basic information about a module, and to keep the information that is being defined in the various wizard pages. The ModuleType knows how to create an associated ModuleWizardContext (see moduleType.createModuleWizardContext()).

The ModuleWizard provides two standard pages:

- The first one allows the users to decide whether they want a module to be created from scratch, or whether the module should be copied from a directory or an archive.
- The second page allows the users to define the name and directory they want to use for their module (by default, these will use the same name).

  Additionally, it allows the users to determine which specifications the created module will support. The specifications list is specific to the ModuleType and has appropriate defaults selected, which are also associated with that ModuleType. The defaults are usually the latest versions of the specifications. For example, if an EJB module is created, the EJB 2.0 specification is selected from the list. Currently, the user can select all of the specifications provided; there might actually be a server that supports all of them.

  The default selected specifications are customizable by the current project's DeployService (see the deployService.getDefaultSelectedFeatures() method).

The ModuleWizard also allows each module implementation to add its own custom pages. These are provided by the ModuleWizardContext implementation for the particular module type, and should all be extensions of the ModuleWizardPage class.

To add custom pages, you need to override the ModuleWizardContext.getCustomWizardPages() method. Additionally, module wizard pages can be implemented as property pages, so that they can show up both in the wizard and in the property dialog for the module node. To do this, extend your wizard pages from ModuleWizardPropertyPage.

ModuleNodes can also be created without going through the wizard. This will happen when the user uses the Project For Existing Code wizard, and in that source we detect a directory that looks like it could represent a module. The detection check is module-specific.

Additional build tasks

The module node allows each of its implementations to customize what happens when the module gets cleaned, after the module gets compiled, and after the archive gets built and/or the associated module directory gets copies of the necessary files. It also

allows the associated `DeployService` to add its own tasks after any compilation takes place and after the build process for a `ModuleNode` completes.

Note Here is where the extra server-specific build commands would be added (like `ejbc` for WebLogic, for instance).

All the `Server` specific build methods are part of the `DeployService`, which is now really a combination of deploy build tasks and actual deploy tasks. You only want to build an archive/exploded directory for the server to which you actually want to transfer the deployable archive/directory, so it doesn't make sense to separate the build/file transfer concepts.

To add extra tasks, `the DeployService`'s `updatePostCompileBuildTask()` and/or `updatePostArchiveBuildTask()` can be overridden.

Property pages

Currently, there are four special property pages that appear for a `ModuleNode`. There are also other property pages that are normally shown for an archive node.

Module property page

The Module property page is a variation on the basic archive content property page. It allows the user to configure the module name, archive name, and associated directory. Additionally, this page has a list of file types associated with the module. This is basically a filter of files that can go into the archive and that will be shown in the module directory.

It is possible that in the future the file types concept will be merged with the classes page for an archive, where the user can specify patterns for the archive. In this case, this page will not need the file type selection controls.

Each module can have its own set of file types to be included in the archive/module directory.

Note Add the extra `ModuleNode` specific property pages after the Module page. For example, if you look at the property dialog for a `WebModuleNode`, you will see the Web property page right after the Module property page. Each module type can add additional pages that are specific to that type. These would appear under a `ModuleType` titled page.

Clean property page

The Clean property page defines the file patterns that can be included or excluded when the module is being cleaned (this happens on a clean build request and before every rebuild of the module). It looks very similar to the Classes page, in that it allows adding include/exclude filters, with a few exceptions:

- You are filtering patterns for files to remove from, not add to, the archive/directory.
- File selection is not allowed, since these might not exist yet. File patterns, however, can be added for files that will exist in the future.

This page has some defaults defined for cleaning. For example, we remove all the contents of the descriptors directories, but not the descriptors themselves, so the users can keep their latest changes.

Build property page

The Build property page defines various build properties for a module. It allows the user to decide whether they want an archive and/or directory content to be created when the module is built, as well as when the build of the node will happen (with the project, only when a node is requested to be built, etc.).

Directories property page

The Directories property page represents directories to be filtered out of the associated archive. This is useful for excluding various backup- and VCS-associated directories that don't make sense for a module archive. This concept might also need to become part of the general classes archive page, in which case this page might not be necessary.

Viewing the DD Editor for a ModuleNode

When the user double-clicks the node that represents the `ModuleNode` in the project viewer, one of the viewers that comes up will be the DD Editor. For most module types, it will be the first viewer, but for the `EjbModuleNode` it will be the second one. The EJB Designer will be the first viewer, in that case.

The DD Editor will allow people to visually configure the descriptors associated with the module. When the descriptor files under the descriptors node are opened, however, these will be viewed as plain XML files. This allows people to edit the actual source for the files, if they wish to do so.

Changes in JBuilder 2005

Read Only Module Support

JBuilder now supports creating a read-only module. This means that this module is only available for viewing and deploying as is. This module cannot be built.

We attempt to take care of preventing building of any module-related tasks if the module is read only; however, if you have your own builders for your server, we cannot prevent you from running your builder, since the build system has been designed to make each build task independent of any other actions.

Things to keep in mind:

- If you do have your own BuildTask, you should make `moduleNode.isReadOnly()` part of the `isMakeable()` and the `isCleanable()` checks.
- If you do any pre-build tasks, you should add the read-only module check at the top of your pre-build code (i.e. in `builder.updateBuildTask()`); this will prevent the read-only module from getting built when it cannot be and thereby causing various errors.

General ModuleNode Related Changes

ModuleNode FileType filter support

Instead of having an explicit string extensions list via the `getExtensions()` method, `ModuleNode`s now just use a `FileType` filter set in the module content. This is modifiable on the new module Content property page (basically the previous Classes property page renamed, and more generic).

To support this, all the places that used the `ModuleNode.getExtensions()` method now use the `ModuleNode.getFileTypes()` method. Also, there are corresponding method additions, e.g., `ModuleType.getDefaultModuleContentFileTypes()`.

The default file types can now be registered with the `ModuleType` by calling:

```
ModuleManager.registerDefaultFileTypeSettings(new
    DefaultFileTypeSettings(FileType.getFileType(EXTENSION)))
```

The `DefaultFileTypeSettings` class is just a wrapper of the basic information associated with a `FileType` for a module (i.e, whether it should be contained in the `ModuleType`) and whether it is to be preserved on clean.

Cleaning support was also added for `FileType`s — default value methods that used an extension pattern for cleaning, now have `FileType` equivalents — i.e. `ModuleType.getDefaultFileTypesToPreserveOnModuleClean()`.

Most of the extensions methods have been deprecated, so that we can still, at least for this release, work properly if there were an override of such a method: we call some of these and convert the result in the new `FileType` methods.

Build (Synchronize) Module Directory is now a ModuleBuildRule instead of a boolean

Previously, you could only choose to synchronize the module directory with the contents of the module archive or not, meaning that the directory would contain the same files are the archive or not. This setting was available on the ModuleNode's Build page, and was controlled by the `ModuleNode.getBuildExplodedArchive()` method. Now this method is still available, but is not controlled by a `boolean` setting. Instead it is controlled by the value of an associated `ModuleBuildRule`. To get and set the rule, new `getExplodedArchiveBuildRule()` and `setExplodedArchiveBuildRule()` methods have been introduced.

To make the method names more meaningful, the previously existing `getBuildRule()` and `setBuildRule()` methods have been renamed to `getZippedArchiveBuildRule()` and `setZippedArchiveBuildRule()` (previous versions have been deprecated).

Chapter 19

JBuilder Enterprise Setup dialog concepts

JBuilder provides an OpenTools API for adding a `Setup` page to the Enterprise Setup dialog box that comes up when the user chooses Enterprise|Enterprise Setup.

This dialog brings up various custom setup pages that can be used to let the user set up some custom tools that might be added to JBuilder. JBuilder itself uses this setup mechanism for setting up CORBA, Application Servers, and Database Drivers. Although this dialog box is titled Enterprise Setup, it's available in both Enterprise and Developer versions. Only the Database Drivers Setup page is available in the Developer version.

The basic process for adding your own `Setup` is to register a class that extends the `Setup` abstract class, and also to provide a class that extends either the `SetupPropertyPage` class or the `NestingSetupPropertyPage` class. These page classes are extensions of the `PropertyPage` class, and they are the ones that define the actual controls that will be used to provide the setup values.

The classes and interfaces that are used for the OpenTools setup mechanism reside in the `com.borland.jbuilder.ide` package. They consist of:

- `Setup`
- `SetupPage`
- `SetupPropertyPage`
- `NestingSetupPropertyPage`
- `SetupManager`

Features and subsystems

Registering a Setup

To register a custom `Setup`, you must first extend the `Setup` abstract class and define all of its abstract functions. Then, define an `initOpenTools()` method to register this class with the `SetupManager`, as shown in the following example:

```
public static void initOpenTool(byte majorVersion, byte minorVersion) {
 SetupManager.registerSetup(new AppServerSetup());
}
```

This adds a `Setup` that appears in the tree in the Enterprise Setup dialog. If you provide an optional second argument to `registerSetup()` that is the name of an existing `Setup`, the setup you register will appear as a node underneath the existing setup's node. However, make sure that the existing `Setup` is one that provides a `NestingSetupPropertyPage` definition. Here is an example:

```
public static void initOpenTool(byte majorVersion, byte minorVersion) {
 SetupManager.registerSetup(new IASSetup(),
 AppServerSetup.APPSERVER_SETUP_NAME);
}
```

You must also provide a class that extends the `SetupPropertyPage` class, or the `NestedPropertyPageClass`. This class provides a `JPanel` with controls to display on its setup page. This class should be instantiated and returned in the implementation of the `getSetupPropertyPage()` function in your extended `Setup` class. Here's an example:

```
public PropertyPage getSetupPropertyPage(final Browser browser) {
 return new IASSetupPage(browser, getName());
}
```

Defining a Setup

The functions you need to implement in your class that extends the `Setup` class are fairly simple:

- **`getName()`** — returns the name of this `Setup`. This will be used as the title of the node that displays controls for this `Setup`.
- **`isEnabled()`** — determines under which conditions this `Setup` should be enabled. If it's not enabled, its page won't show up in the dialog box.
- **`getSetupPropertyPage()`** — returns the actual page that will display the controls used to set up your tool. This can be a nested page, that will display other `SetupPropertyPages`.

Defining a SetupPropertyPage

`SetupPropertyPage` is an abstract class that extends the `PropertyPage` class. It requires you to implement two functions:

- **`getDescription()`** — returns a description of what the user should do on this page. This description will appear as text at the top of the page.
- **`createSetupPanel()`** — returns a `JPanel` that will be displayed under the description text, and will contain the controls necessary to set up your tool.

If you are extending `NestingSetupPropertyPage`, all you need to define is the `getDescription()` method and the subordinate pages will be added by the `Setup` internal mechanism.

Chapter

20

OpenTools code samples

The `samples/OpenToolsAPI/` directory contains several sample projects that demonstrate how to create OpenTools that extend Primetime and JBuilder. These samples include:

- `Actions`

  Demonstrates an action to display a simple greeting in a modal dialog.

- `Build`

  Uses RetroGuard to create obfuscated JARs for Archive nodes. It replaces the target JAR of each Archive node with an obfuscated JAR.

- `CommandLine`

  Demonstrates how to register a JBuilder command-line option.

- `CurlyBraceKeyBinding`

  Shows how to add a keybinding to complete curly braces.

- `DiffViewer`

  Demonstrates how to implement a viewer that shows differences between two files.

- `EditorStats`

  Adds a tools menu item and hooks it up to a dialog with editor statistics.

- `JOM`

  Contains two samples that use JAM/JOM:

  - `ReadingSource`

    Reads a source file.

  - `WritingSource`

    Writes a source file.

- `LineCommentHandler`

  Show how to change a keybinding and make it talk to the editor.

- `ModifyCaret`

  How to change the shape of the caret.

- `ModifyKeyBinding`

  How to change keybindings in editor emulations.

- `NodeDemo`

  Explains how to create a new file node type and a viewer to display the contents of that file type. It also shows you how to add menu items to popup (context) menus.

- `Viewers`

  Samples of file node viewers for text and image files:

  - `Delphi`

    A sample `TextFileNode` viewer using custom syntax highlighting and generating structure pane content with the OpenTools API.

  - `Image`

    An example non-text file node viewer with the OpenTools API.

- `Wizards`

  Sample wizards:

  - `Doclet`

    Implements a doclet and extends the Javadoc Wizard in JBuilder. Techniques used here are very similar to those used to extend the Archive Builder.

  - `Find`

    Example of a wizard using the `MessageView` with the OpenTools API.

  - `Gallery`

    Example of an object gallery wizard using a `PropertyGroup` with the OpenTools API.

  - `PackageWizard`

    Example of a wizard using a project pane popup Action to add file and package nodes with the OpenTools API.

  - `Simple`

    Example of a simple wizard built with the OpenTools API.

- `XmtPropertiesExample`

  Demonstrates a range of key aspects of the XMT, including node and tree management, reading, writing, configuring, and validating.

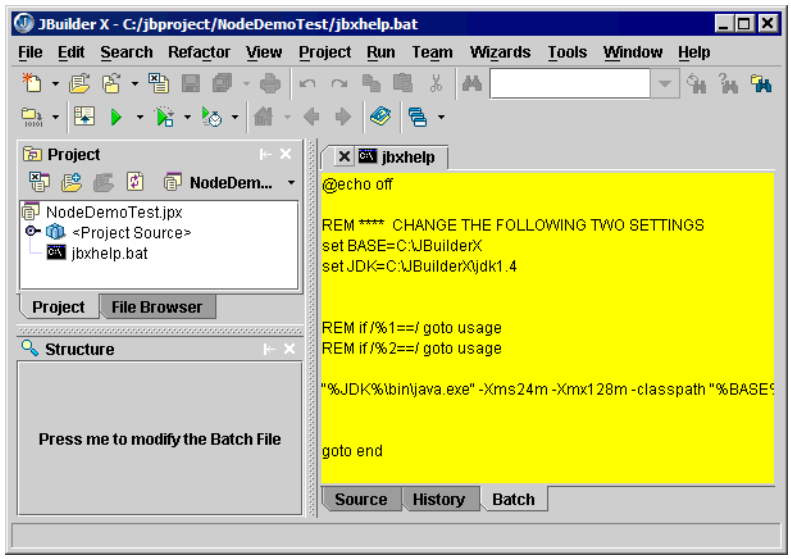
All of these reside in the `samples/OpenToolsAPI` directory in your JBuilder installation.

Chapter

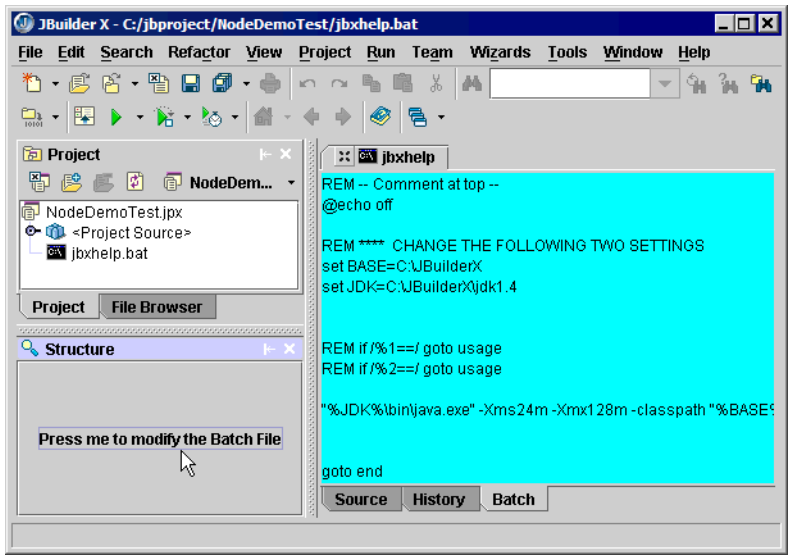
21

Adding a file node type and a viewer

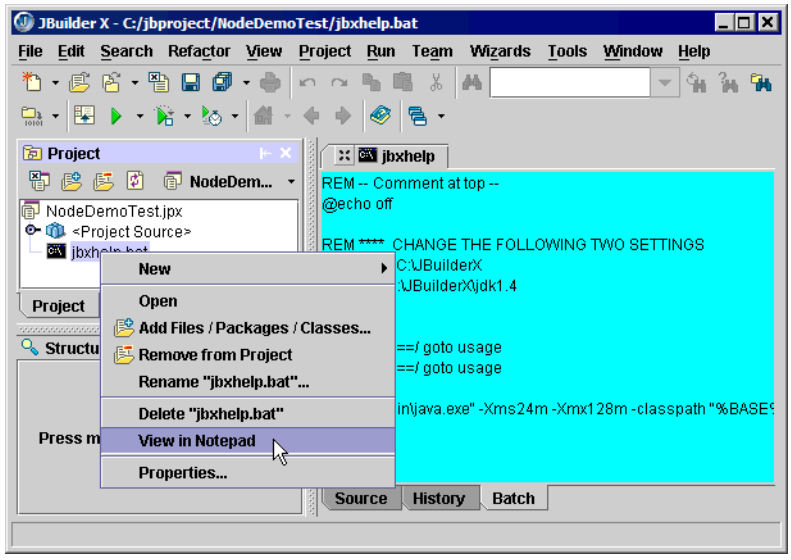
This tutorial creates OpenTools that create a new file node type and a viewer that can display the new node type. In this case, the new node type is a batch file node. The new batch node viewer is a text viewer that displays the contents of a batch file on a bright yellow background. The viewer has a Batch tab at the bottom of the content pane.



Note that the a giant button replaces the structure pane. When the user clicks the button, a comment line is added to both the top and bottom of the batch file and the background color changes to cyan.



This tutorial also adds a menu item to context menus. A View in Notepad menu option appears in the project pane's context menu and in the content pane's context menu when the Source tab is selected.



This tutorial teaches these primary skills:

- How to create a new file node type.
- How to create a viewer to view the new file node type.
- How to work with the virtual file system.
- How to replace the structure view with a component.
- How to add menu items to context menus that are pertinent to a particular file node.

You can find the source code for this OpenTool sample in the `samples/OpenToolsAPI/NodeDemo` directory.

Getting started

To begin creating the OpenTool,

1. Create a new project called `NodeDemo.jpx`.
2. Add the OpenTools SDK library as a required library to your project. This adds the classes which make up the OpenTools API to your project classpath.

If you need help performing these tasks, see Chapter 2, "OpenTools basics."

This tutorial requires you to create four classes:

- `BatchFileNode`

  The new file node type that is registered with the IDE.

- `BatchViewerFactory`

  A factory that creates a batch file viewer when it determines one is needed.

- `BatchViewer`

  The new viewer that knows how to display the contents of batch files.

- `BatchMenu`

  A menu class that adds the View in Notepad menu option on the project pane and source pane context menus when the new batch file node is selected in the project pane.

Creating the BatchFileNode class

Use the Class wizard to begin the new class:

1. Choose File|New Class to display the Class wizard.
2. Enter the name of the class as `BatchFileNode`.
3. For the base class, specify `com.borland.primetime.node.TextFileNode`.
4. Verify that the Public and Override Superclass Constructors options are checked, and uncheck all the other options.
5. Choose OK.

Your code in the editor should look like this:

```
package NodeDemo;

import com.borland.primetime.node.TextFileNode;
import com.borland.primetime.node.Project;
import com.borland.primetime.node.Node;
import com.borland.primetime.vfs.Url;
import com.borland.primetime.node.DuplicateNodeException;

public class BatchFileNode extends TextFileNode {

  public BatchFileNode(Project project, Node parent, Url url)
      throws DuplicateNodeException {
    super(project, parent, url);
  }
}
```

Because the batch file node is text rather than binary data, it extends the `com.borland.node.TextFileNode`, which in turn extends `com.borland.node.FileNode`.

Adding an icon

The new file node requires an icon to display in the project pane. First, use the Override Methods wizard from the Edit|Wizards menu to check the `getDisplayIcon()` method under `com.borland.primetime.Node`, uncheck the Generate Javadoc for Method Overrides option, then click OK.

Your code in the editor should look like this:

```
package NodeDemo;

import com.borland.primetime.node.TextFileNode;
import com.borland.primetime.node.Project;
import com.borland.primetime.node.Node;
import com.borland.primetime.vfs.Url;
import com.borland.primetime.node.DuplicateNodeException;
import javax.swing.Icon;

public class BatchFileNode extends TextFileNode {

  public BatchFileNode(Project project, Node parent, Url url)
      throws DuplicateNodeException {
    super(project, parent, url);
  }

  public Icon getDisplayIcon() {
    return null;
  }
}
```

Now you need to define the icon to use. Add this code to the class definition:

```
public static final Icon ICON = new
   ImageIcon(BatchFileNode.class.getResource("Batch.gif"));
```

You should get an error because `ImageIcon` is undefined. Click on the ErrorInsight icon next to the line, select the Suggest Class Reference option on the popup, then the `javax.swing.ImageIcon` from the Suggest Correction dialog. This will add the needed import to your class and fix the error condition.

To provide access to the icon you just defined, modify your `getDisplayIcon()` method to return a reference:

```
public Icon getDisplayIcon() {
  return ICON;
}
```

Add a `Batch.gif` file to your project in the same directory as your BatchFileNode class. You can find one in the `samples/OpentoolsAPI/NodeDemo` directory.

Registering the batch file node type

Each OpenTool must have an `initOpenTool()` method that is called as JBuilder is loading. The `initOpenTool()` method of `BatchFileNode` registers the new file node type with JBuilder. Add this method to the class:

```
public static void initOpenTool(byte major, byte minor) {
    FileNode.registerFileNodeClass("bat", "Batch file", BatchFileNode.class,
    ICON);
}
```

After JBuilder loads this new OpenTool and the user adds a file with a `.bat` extension to a project, the new file node is represented with the icon in the project pane. At this point, there is no viewer associated with the new node type.

For more information about creating a new node type, see "Registering a new file type" on page 28 and other topics in "Content Manager concepts" on page 26.

Creating the BatchViewerFactory class

Each node viewer is created by a node viewer factory, which decides whether it is able to create a viewer to display a particular node type. If the node viewer factory thinks it can create the node viewer, it attempts to do so. You need a new node viewer factory that can create a viewer for displaying batch files. For more information about node viewers and the node viewer factories that create them, see "Content Manager concepts" on page 26.

Use the Class wizard to begin the new class:

1. Choose File|New Class to display the Class wizard.
2. Enter the name of the class as `BatchViewerFactory`.
3. For the base class, choose `java.lang.Object`.
4. Verify that the Public option is checked, unchecking all other options.
5. Choose OK.

Use the Implement Interface wizard to have the class implement the `com.borland.primetime.ide.NodeViewerFactory` interface:

1. Choose Edit|Wizards|Implement Interface.
2. Navigate to the `com.borland.primetime.ide.NodeViewerFactory` interface.
3. Uncheck the Generate Javadoc for Interface Methods option.
4. Choose OK.

Your code in the editor should look like this:

```
package NodeDemo;

import com.borland.primetime.ide.NodeViewer;
import com.borland.primetime.ide.Context;
import com.borland.primetime.node.Node;
import com.borland.primetime.ide.NodeViewerFactory;

public class BatchViewerFactory
    implements NodeViewerFactory {
  public boolean canDisplayNode(Node node) {
    return false;
  }

  public NodeViewer createNodeViewer(Context context) {
    return null;
  }
}
```

Examining the file node type

The `NodeViewerFactory` interface has just two methods you must implement: `canDisplayNode()` and `createNodeViewer()`. Start with `canDisplayNode()` first.

The JBuilder IDE must quickly poll the registered node viewer factories to determine whether a factory exists that can create the right type of viewer for the selected file node. It does this by calling the `canDisplayNode()` method of the registered node viewer factories.

The `canDisplayNode()` method simply determines whether the file node passed to it is the right type of file node; in this case, whether it is an instance of a `BatchFileNode`. Modify the `canDisplayNode()` method in your code so that it looks like this:

```
public boolean canDisplayNode(Node node) {
  return node instanceof BatchFileNode;
}
```

`canDisplayNode()` returns true if the node in question is an instance of `BatchFileNode`.

Creating a node viewer

The `createNodeViewer()` method attempts to create a viewer to display the node. In this case, it tries to create a `BatchViewer` instance, which is a viewer you'll create later for displaying batch files in the content pane. Modify the `createNodeViewer()` method in your code so that it looks like this:

```
public NodeViewer createNodeViewer(Context context) {
  if (canDisplayNode(context.getNode())) {
    return new BatchViewer(context);
  }
  return null;
}
```

The `context` parameter specifies a particular browser/node pair. You should be seeing an error because BatchViewer does not yet exist.

Registering a node viewer factory

Each OpenTool must have an `initOpenTool()` method that is called as JBuilder is loading. The `initOpenTool()` method of `BatchViewerFactory` registers the new file node type with JBuilder. Add this method to the class:

```
public static void initOpenTool(byte major, byte minor) {
  Browser.registerNodeViewerFactory(new BatchViewerFactory());
}
```

You should be seeing another error. Again use ErrorInsight and its Suggest Class Reference to find `com.borland.primetime.ide.Browser` and generate the needed import statement.

Creating the BatchViewer class

Now you need a viewer class that your new `BatchViewerFactory` can create. The `BatchViewer` class you are going to create displays the contents of the a batch file in the content pane. It has its own Batch tab the user can use to switch to this view if it's not the one currently displayed. `BatchViewer` also replaces the usual hierarchy in the structure pane with a button the user can use to modify the contents of the buffer the viewer is presenting.

Use the Class wizard to begin the new class:

1. Choose File|New Class to display the Class wizard.
2. Enter the name of the class as `BatchViewer`.
3. For the base class, choose `com.borland.primetime.viewer.AbstractBufferNodeViewer`.
4. Verify that the Public, Override Superclass Constructors, and Override Abstract Methods options are checked, unchecking all other options.
5. Choose OK.

Use the Implement Interface wizard to have the class implement the `java.awt.event.ActionListener` interface:

1. Choose Edit|Wizards|Implement Interface.
2. Navigate to the `java.awt.event.ActionListener` interface.
3. Uncheck the Generate Javadoc for Interface Methods option.
4. Choose OK.

Your code should look like this:

```
package NodeDemo;

import javax.swing.JComponent;
import com.borland.primetime.viewer.*;
import com.borland.primetime.vfs.*;
import com.borland.primetime.ide.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class BatchView extends AbstractBufferNodeViewer
    implements ActionListener {

  public BatchView(Context context) {
    super(context);
  }

  public BatchView(Context context, int updateMode) {
    super(context, updateMode);
  }
  public JComponent createViewerComponent() {
    return null;
  }
  public byte[] getBufferContent(Buffer buffer) {
    return null;
  }
  protected void setBufferContent(byte[] content) {
  }
  public String getViewerTitle() {
    return "";
  }
  public JComponent createStructureComponent() {
    return null;
  }

  public void actionPerformed(ActionEvent e) {
  }
}
```

Most node viewers refresh each time a buffer change takes place, even if the viewer isn't visible or it doesn't have the focus. `BatchViewer`, however, extends `AbstractBufferNodeViewer`, which is a node viewer that can cache buffer changes and update the view only when necessary.

Modifying the constructor

Modify the default constructor so that a `context` parameter that identifies a unique browser/node pair is sent to it and the parent class is informed that the viewer buffer should update only when it is visible:

```
public BatchViewer(Context context) {
  this(context, UPDATE_WHEN_VISIBLE);
}
```

`UPDATE_WHEN_VISIBLE` is a class constant defined in `AbstractBufferNodeViewer`.

Creating the viewer component

Modify the `createViewerComponent()` method in your class so that it creates an instance of `JTextArea`, makes the text area read only, and sets its background color to yellow. First define a `JTextArea` field in the class:

```
private JTextArea area;
```

Then implement the `createViewerComponent()` method:

```
public JComponent createViewerComponent() {
  area = new JTextArea();
  area.setEditable(false);
  area.setBackground(Color.yellow);
  return area;
}
```

Also implement the `getViewerTitle()` method so that it returns the string "Batch". This string is used as the label of the tab for the viewer:

```
public String getViewerTitle() {
  return "Batch";
}
```

Adding a structure pane component

Usually the structure pane displays information relating to the node being viewed, often in a hierarchical structure. Instead, you can place any component in that space. In this case, you add a `JButton` the user can click to modify the buffer displayed in the viewer.

First add a new field to the class for the button:

```
private JButton button;
```

Then implement the `createStructureComponent()` method with this code:

```
public JComponent createStructureComponent() {
  button = new JButton("Press me to modify the Batch File");
  button.addActionListener(this);
  return button;
}
```

As the new button instance is created, it's passed the "Press me to modify the Batch File" string. When the button is "pressed," an `actionPerformed()` method is called. You

must implement that method in the `BatchViewer` class. Modify `actionPerformed()` so that it looks like this:

```
public void actionPerformed(ActionEvent e) {

  area.setText("REM -- Comment at top --\n" + area.getText() +
               "REM -- Comment at bottom\n");

  try {
    setBufferModified();
  }
  catch (ReadOnlyException ex) {
    setBufferContent();
  }
}
```

Working with the buffer

The `actionPerformed()` method adds a comment to the top and bottom of the text in the viewer component. Then `setBufferModified()` is called, which the Virtual File System (VFS) takes care of. `setBufferModified()` throws a `ReadOnlyException`, however, so you must handle it. Remember you specified the viewer as read only. In this case, the catch clause calls `setBufferContent()`, a method of the `AbstractBufferNodeViewer` class.

Modify the `setBufferContent()` method with this code, making sure you change the `parm1` parameter name to `content`:

```
protected void setBufferContent(byte[] content) {
  try {
    area.setText(new String(content, getEncoding()));
  }
  catch (UnsupportedEncodingException ex) {
  }
}
```

The `AbstractBufferNodeViewer`, the parent class of your `BatchViewer` class, decides when to call the `setBufferContent()` method. This allows `AbstractBufferNodeViewer` to save up changes to the buffer when the viewer isn't active or visible. When `setBufferContent()` is called, the viewer is refreshed if the buffer has changed. So the `setBufferContent()` you implemented attempts to modify the viewer contents by setting the new text. It also calls the `getEncoding()` method to discover the appropriate encoding to use when converting any binary data.

Responding to buffer changes

The `AbstractBufferNodeViewer` implements the `BufferListener` interface, which contains a `bufferStateChanged()` method that is called by the `Buffer` object when the buffer changes. You must implement `bufferStateChanged()`. Add a `bufferStateChanged()` method to your class that changes the background color to cyan, and, if the buffer state is ready only, changes the state so it can edited. Here is the code to do that:

```
public void bufferStateChanged(Buffer buffer, int oldState, int newState) {
  area.setBackground(
      (newState & Buffer.STATE_MODIFIED) == Buffer.STATE_MODIFIED ?
      Color.cyan : Color.yellow);
  button.setEnabled(
      (newState & Buffer.STATE_READONLY) != Buffer.STATE_READONLY);
}
```

Returning the buffer content

The `AbstractBufferNodeViewer` also implements the `BufferUpdater` interface, which defines a `getBufferContent()` method. Your derived class must supply an implementation of this method. Here `getBufferContent()` returns the current content of the buffer as an array of bytes:

```
public byte[] getBufferContent(Buffer buffer) {
  return area.getText().getBytes();
}
```

Adding menu items to a context menu

When the user right-clicks a batch file node in the project pane, the popup (or context) menu that appears contains a View in Notepad menu command the user can choose to display the batch file in the notepad.exe utility. The same menu command appears on the popup menu that appears when the user right-clicks a batch file displayed in the Source pane. Although the menu commands look the same, how they are implemented differs. The `BatchMenu` class shows you how to use both methods.

To begin the `BatchMenu` class,

1. Choose File|New Class to display the Class wizard.
2. Enter the name of the class as `BatchMenu`.
3. For the base class, specify `java.lang.Object`.
4. Verify that the Public option is checked, unchecking all other options.
5. Choose OK.

`BatchMenu` implements the `ContextActionProvider` interface that defines just one method, `getContextAction()`.

Use the Implement Interface wizard to have the class implement the `com.borland.primetime.ide.ContextActionProvider` interface:

1. Choose Edit|Wizards|Implement Interface.
2. Navigate to the `com.borland.primetime.ide.ContextActionProvider` interface.
3. Choose OK.

The wizard adds the `getContextAction()` method to your code. Your code should look like this:

```
package NodeDemo;

import com.borland.primetime.ide.ContextActionProvider;
import javax.swing.Action;
import com.borland.primetime.ide.Browser;
import com.borland.primetime.node.Node;

public class BatchMenu
    implements ContextActionProvider {
  public Action getContextAction(Browser browser, Node[] nodeArray) {
    return null;
  }
}
```

Providing an action

The `getContextAction()` method specifies an action that occurs within a particular context; in this case, when the user right-clicks a batch file node in the project pane.

Modify `getContextAction()` so that if the currently selected node is a single batch file node, an action that opens the file in the notepad.exe utility occurs. (Be sure you change the name of the parameters passed to `getContextAction()` to `browser` and `nodes`.) Here is the code:

```
public Action getContextAction(Browser browser, Node[] nodeArray) {
  if (nodeArray.length == 1 && nodeArray[0] instanceof BatchFileNode)
    return ACTION_VIEW_NOTEPAD;
  return null;
}
```

You must now define the `ACTION_VIEW_NOTEPAD` action that `getContextAction()` returns. Here is the action code in its entirety:

```
public static final Action ACTION_VIEW_NOTEPAD = new BrowserAction( "View in
    Notepad") {
  public void actionPerformed(Browser browser) {
    Node node = browser.getProjectView().getSelectedNode();
    if (node instanceof BatchFileNode) {
      BatchFileNode batchNode = (BatchFileNode)node;
      try {
        String path = batchNode.getUrl().getFileObject().getAbsolutePath();
        Runtime.getRuntime().exec("notepad " + path);
      }
      catch (Exception ex) {
      }
    }
  }
};
```

The action created is a `BrowserAction` given the text string “View in Notepad” that appears on the context menu. The current `Browser` instance is passed to the `actionPerformed()` method, which begins by obtaining the selected node in the project pane and then determining whether that node is a batch file node type. If it is, the method calls `notepad.exe`, giving it to open the fully-qualified name of the batch file the node represents.

Writing the initOpenTool() method

In the `initOpenTool()` method you must create to register the new action with JBuilder, you must declare a `menu` field of type `BatchMenu`, and then register it as a context action provider for the project pane. Here is the code to do that:

```
public static void initOpenTool(byte major, byte minor) {
  BatchMenu menu = new BatchMenu();

  ProjectView.registerContextActionProvider(menu);
}
```

Your `BatchMenu` class now adds a menu item to the context menu of the project pane when the user selects a batch file node in the project pane.

Doing it another way

You still need to add the code that adds a menu item to the context menu of the editor when a batch file is open. Instead of implementing the `EditorContextActionProvider` interface in the `BatchMenu` class and then registering the whole class as the provider, you could choose instead to create and register a local class as the provider.

Begin by starting the definition of a `ViewNotepad` class, which implements the `EditorContextActionProvider` interface:

```
static EditorContextActionProvider ViewNotepad = new
    EditorContextActionProvider() {
};
```

Classes that implement `EditorContextActionProvider` must also implement a `getContextAction()` method, but this one is passed an instance of an `EditorPane` that appears in the Source pane when the batch file is opened in the code editor. Below is the code for `getContextAction()`; place it in the `ViewNotepad` class definition.

```
public Action getContextAction(EditorPane editor) {
  Browser browser = Browser.findBrowser(editor);
  Node node = browser.getActiveNode();
  if (node instanceof BatchFileNode)
    return GROUP_ViewNotePad;
  return null;
}
```

`getContextAction()` obtains the selected node, and if the node is an instance of a batch file node, it returns the `ActionGroup GROUP_ViewNotePad`. By specifying a separate `ActionGroup` for the menu command, it will appear in its own group, separate from the other menu items on the editor context menu.

An implementation of `EditorContextActionProvider` must also include a `getPriority()` method:

```
public int getPriority() {
  return 4;
}
```

The priority of a menu item determines where the item appears on a menu. Possible priorities range from 1 - 100. A priority less than 5 usually results in the menu item appearing at the bottom of the menu, while a priority of 99 usually makes it appear at the top. Priorities are shared with other menu entries, so no priority value guarantees a specific position on the menu.

For clarity, the entire `ViewNotepad` implementation is presented here:

```
static EditorContextActionProvider ViewNotepad = new
    EditorContextActionProvider() {

  public Action getContextAction(EditorPane editor) {
    Browser browser = Browser.findBrowser(editor);
    Node node = browser.getActiveNode();
    if (node instanceof BatchFileNode)
      return GROUP_ViewNotePad;
    return null;
  }

  public int getPriority() {
    return 4;
  }
};
```

You have yet to define the `GROUP_ViewNotePad` that `getContextAction()` returns. The group contains just one action. Define the `ActionGroup` like this in your class:

```
protected static final ActionGroup GROUP_ViewNotePad = new ActionGroup();
static {
  GROUP_ViewNotePad.add(ACTION_EDITOR_VIEW_NOTEPAD);
}
```

Finally, define the action that is called when the user right-clicks the batch file in the editor; place it *above* the `GROUP_ViewNotePad` definition you just added in the class:

```
public static final AbstractAction ACTION_EDITOR_VIEW_NOTEPAD =
    new UpdateAction("View in Notepad",
                     'v',
                     "View in Notepad") {

      public void update(Object source) {
        Browser browser = Browser.findBrowser(source);
        Node node = browser.getActiveNode();
        setEnabled(node instanceof BatchFileNode);
      }

      public void actionPerformed(ActionEvent e) {
        Browser browser = Browser.findBrowser(e.getSource());
        Node node = browser.getActiveNode();
        if (node instanceof BatchFileNode) {
          // Show the batch file in Notepad.
          BatchFileNode batchNode = (BatchFileNode)node;
          try {
            String path = batchNode.getUrl().getFileObject().getAbsolutePath();
            Runtime.getRuntime().exec("notepad " + path);
          }
          catch (Exception ex) {
          }
        }
      }
    };
```

The `ACTION_EDITOR_VIEW_NOTEPAD` action is defined as an `UpdateAction` that contains two methods: `update()` and `actionPerformed()`.

The `update()` method determines whether the menu item text “View in Notepad” appears on the menu. The method obtains the active node and enables the menu item if the node is a `BatchFileNode` instance.

The `actionPerformed()` method also gets the currently selected file node and determines whether it’s a `BatchFileNode` instance. If it is, `actionPerformed()` passes the fully-qualified name of the file to the `notepad.exe` utility to open.

Registering the ViewNotepad class as a ContextActionProvider

The final step is to register your new `ViewNotepad` class as a `ContextActionProvider` with the `EditorManager`. Just as before, you do this in the `initOpenTool()` method. Once you’ve added the necessary code to `initOpenTool()`, the `initOpenTool()` code should look like this, with the new code shown in bold:

```
public static void initOpenTool(byte major, byte minor) {
  BatchMenu menu = new BatchMenu();

  ProjectView.registerContextActionProvider(menu);

  EditorManager.registerContextActionProvider(ViewNotepad);
}
```

Finishing up

To finish your OpenTools, follow these steps:

1. Use File|New|Archive|OpenTools to define the JAR which you will deploy. This will add an OpenTool node in the Project pane and also an empty `classes.opentools` file.
2. Modify the generated `classes.opentool` file to contain the following lines which define the classes which have an `initOpenTools()` method and at what point in JBuilder startup they are to be called:

   ```
   OpenTools-Core: NodeDemo.BatchFileNode
   OpenTools-UI: NodeDemo.BatchViewerFactory NodeDemo.BatchMenu
   ```

   It is important to know that:

   - This layout is very strict.
   - There must be a carriage return terminating the last line that has text.
3. Choose Project|Make Project to compile the classes, fix any syntax errors that might have crept in, and generate the JAR.
4. To test or debug your project, you need to first define a run configuration. Choose Run|Configurations, click New, set the Type as OpenTool, and then click OK. Now you can use Run|Run Project or the toolbar button to run your OpenTool.

   Note that this test mode runs using the classes on your project output path, not the content of the JAR.

For more detailed information on these final steps, see Chapter 2, “OpenTools basics.”

Index

P

R

S

T

V

W

X