Developing OpenTools

VERSION 9

Borland®
JBuilder®

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 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.

JB9devot 6E5R0503
0304050607-9 8 7 6 5 4 3 2 1
PDF

Contents

Chapter

1

Introduction to PrimeTime and JBuilder OpenTools

General description

The concepts introduced in this document describe the basic organization of the OpenTools and some of the basic terminology necessary to get started. Links from this introduction to the detailed documentation for each subsystem will get you directly to the information you need to tailor your IDE.

The codebase referred to as OpenTools started out as the codebase for JBuilder, a Java IDE written, itself, entirely in Java. It was designed to be extensible, customizable, and to take full advantage of good object-oriented programming practices in order to remain so. This has the advantages of making the codebase supported on any platform that supports Java, of permitting programmers to pick and choose among possible features freely, and of making it possible for programmers to base their work on a proven and reliable codebase. Creating other IDEs on the generic side of the codebase is a logical extension of this paradigm.

Because these OpenTools started out as JBuilder code, JBuilder samples are commonly used as examples in these documents. If you're using JBuilder, note that OpenTools features vary by JBuilder edition—please see the JBuilder documentation for the area you're interested in, to find out if you have access to all the OpenTools you want. Generally, OpenTools are available only for features supported in the JBuilder edition you're using.

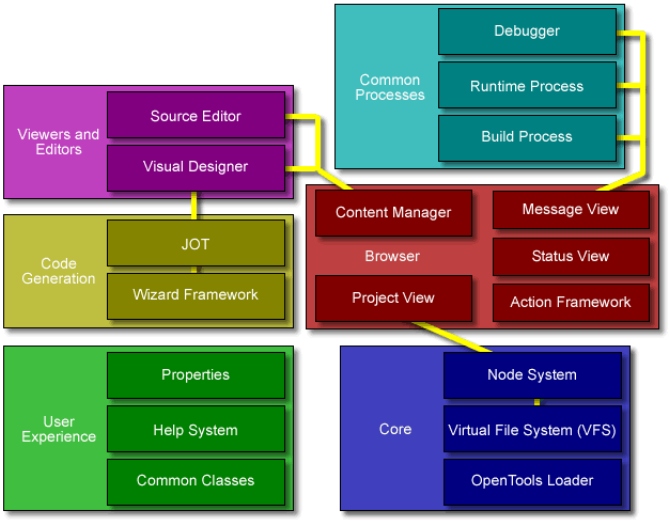
Package structure

OpenTools packages start with either `com.borland.primetime` or `com.borland.jbuilder`. Everything necessary to provide general IDE-style support is in the `primetime` packages, while support specific to editing Java code is in the `jbuilder` packages. This makes `com.borland.primetime` suitable for any programming environment which supports Java, while `com.borland.jbuilder` provides Java programmers with a ready-made suite of materials to use to customize and extend a Java environment.

Once you get used to the split personality, the organization is actually quite logical and well-structured. The `primetime` packages define very general-purpose behavior, and never depend on anything in the `jbuilder` packages.

Architectural overview

This image illustrates the general architecture of the OpenTools. The Browser hosts the viewers and editors and provides access to, and insight into, common processes. The Browser itself depends on core functions, starting with the OpenTools loader, which discovers and defines the product being launched. The viewers and editors interact with the code generation capabilities, communicating changes in the code and altering the Browser display appropriately. The user experience layer is ideally a unified, transparent layer that seems so natural and comfortable to the user that it hardly ever impinges on the user's attention.



Legend

Core	JBuilder's Core subsystem manages OpenTools, collections of files, and projects that underlie the user interface.
Browser	The general Browser user interface provides a framework for interacting with projects and files.
Viewers and Editors	The specific source code Editor and visual Designer subsystems provide a rich set of end user tools for manipulating Java code. Each of these subsystems is extensible in its own way.
Common Processes	The default JBuilder configuration knows how to compile and run basic Java projects, but the subsystems that manage the Build and Runtime processes are extensible to allow the default behavior to be further tailored.
Code Generation	While the Source Editor allows direct manipulation of Java files, automated code generation is also available. The Wizard Framework provides a common look and feel for wizards, and JOT, JOM, and JAM provide the infrastructure necessary to read, alter, and regenerate source code.
User Experience	IDE users become accustomed to a standard look and feel for the environment that is supported by a variety of additional libraries.

The Core

The Core subsystems provide functionality that is available to command-line tools as well as extensions to the full-blown graphical IDE.

OpenTools Loader

Documents: Chapter 4, "OpenTools loader concepts"
Packages: 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 launcher. 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.

Virtual File System (VFS)

Packages: 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.

Node System

Packages: com.borland.primetime.node, com.borland.jbuilder.node
Classes: Project, JBProject, Node, FileNode, LightweightNode, ProjectStorage

The Node System provides the infrastructure for managing projects. A project is the sandbox PrimeTime requires you to play in: it manages paths and properties for a user-determined group of files and resources, using a project file.

All projects are instances descending from the generic PrimeTime `Project` class. Most projects in a JBuilder environment are actually 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 can be registered to read (and optionally write) foreign project-file formats.

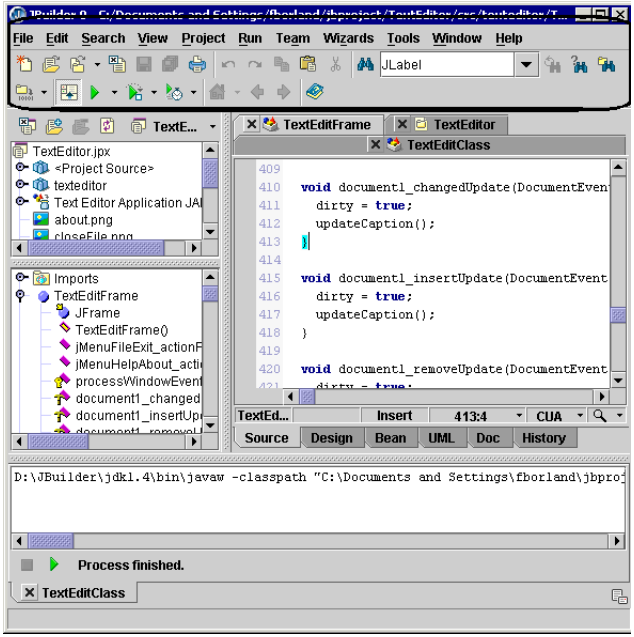
The Browser

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, with the relevant portion of the user interface marked by a rounded rectangle.

Keep in mind that PrimeTime supports displaying more than one browser at a time. All of the components described below are present in each browser window, each of which maintains its own state. Methods that register new behavior affect all browser windows simultaneously.

Action Framework: Menus and Toolbars

Documents: Chapter 5, “Browser concepts”
Packages: com.borland.primetime.actions
Classes: UpdateAction, BrowserAction, StateAction, DelegateAction, JBuilderToolBar, JBuilderMenuBar, ActionToolBar, ActionMenuBar



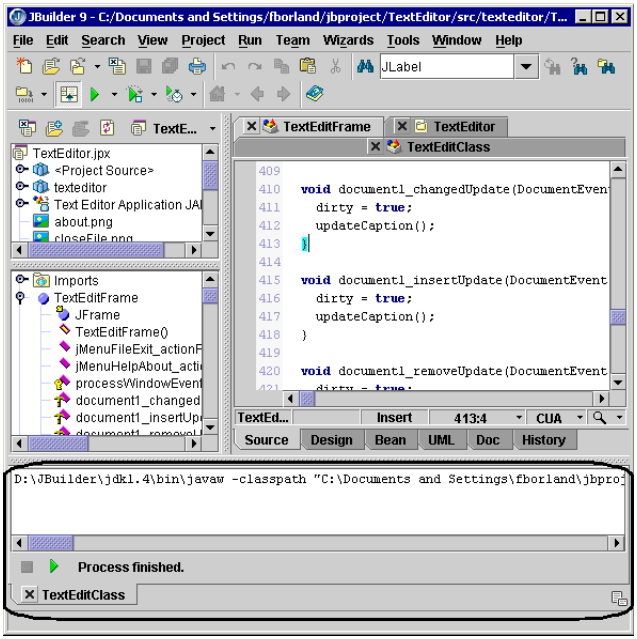
The main menu and toolbar are constructed dynamically from collections of Swing Action objects. The standard menu and toolbars can be extended with new Action objects, and new user interface additions can incorporate additional Action-based menu and toolbars.

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, while the toolbar gives these actions a "pressed" appearance.
- `DelegateAction` instances dynamically connects 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.

Message View

Documents: Chapter 10, “MessageView concepts”
Packages: com.borland.primetime.ide
Classes: MessageView, MessageCategory, Message

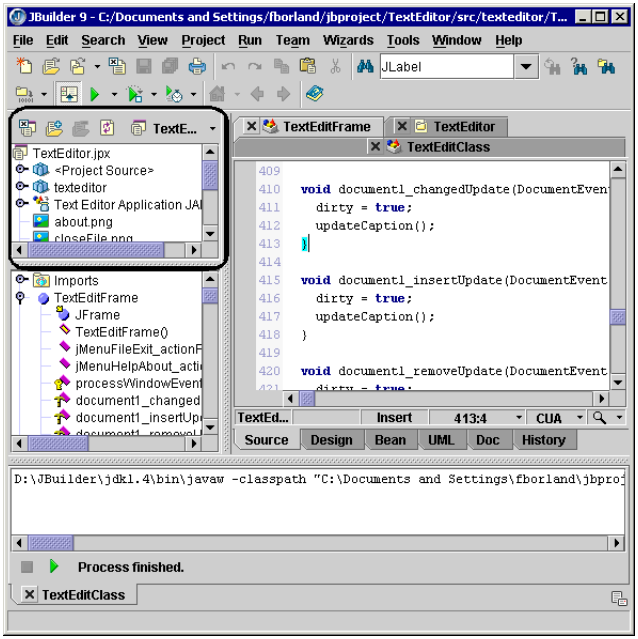


The Message View in PrimeTime automatically appears when content is added to it in order to communicate with the user. Content is organized into groups, each group visible as a tab at the bottom of the message area. Tabs can be created which organize simple messages into lists or hierarchies, or tabs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

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

Project View

Documents:Chapter 7, “ProjectView concepts”
Packages: com.borland.primetime.ide
Classes: ProjectView

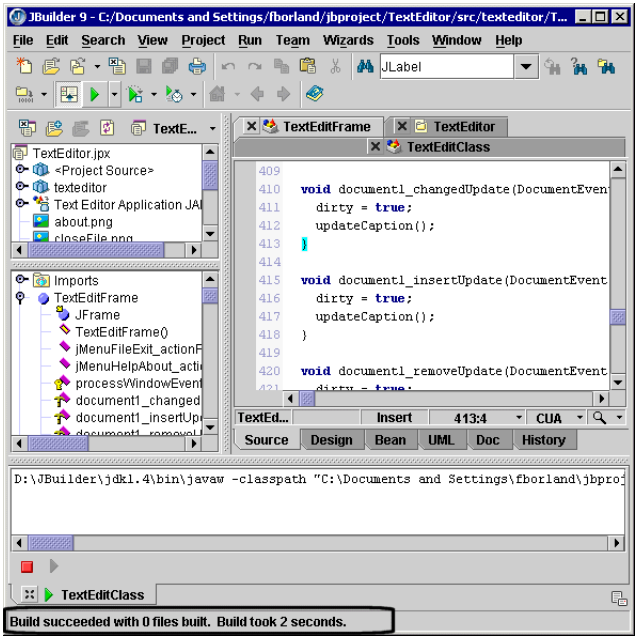


The Project View reflects the state of the active project, representing visually the information available through the Node System including the “display hierarchy” of nodes, their display names and icons. Most interaction with the project view happens indirectly by defining new node types or modifying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

Status View

Documents: Chapter 9, “StatusView concepts”
Packages: com.borland.primetime.ide
Classes: StatusView



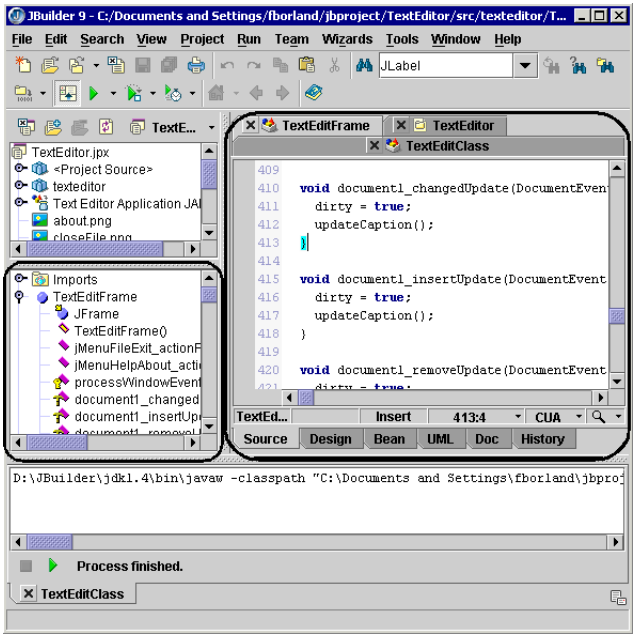
The Status View displays a single message that can be replaced on demand. Informative messages about the last user action are typically displayed here.

The status bar will display long descriptions for toolbar buttons and menu items.

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.

Content Manager

Documents: Chapter 6, “Content manager concepts”
Packages: com.borland.primetime.ide
Classes: Browser, NodeViewer, NodeViewerFactory



The Browser manages the set of open files through the Content Manager. 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, and one for the content pane.

Viewers and editors

Several standard viewers and editors are automatically registered with the Content Manager. The source code editor, visual designer, image viewer, HTML viewer, BeansExpress, and JavaDoc viewer are all examples of standard viewers included with the JBuilder product. Two of these are noteworthy because they can, in turn, be enhanced by other OpenTools.

Source Editor

Documents: Chapter 13, "Editor concepts," Chapter 14, "Keymap concepts"
Packages: com.borland.primetime.editor
Classes: TextFileNode, Scanner, EditorActions

The Source Editor in PrimeTime 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 also be customized at a global level by registering new user-selectable keymappings, each of which can borrow freely from our existing collection of actions implemented in `EditorActions`.

Visual Designer

Packages: com.borland.jbuilder.designer
Classes: JavaFileNode

The Visual Designer in JBuilder is displayed for descendants of `JavaFileNode`. It provides a full JavaBean assembly environment that can be extended in a number of ways. The most obvious is that property editors and customizers included with JavaBeans are automatically exposed through the designer.

Less obvious ways of extending the Visual Designer include:

- Allowing visual manipulation of custom layout managers. The Java layout manager interfaces do not provide enough information to allow visual editing of layout constraints, so JBuilder introduces a "layout assistant" interface. JBuilder comes with assistants for the standard Swing and AWT layout managers.
- Defining a custom designer which is automatically integrated into the overall designer framework. The menu designer and column designers are examples of this mechanism. Your custom designer automatically shares the component tree, inspector, and code generation with the user interface designer.

Standard processes

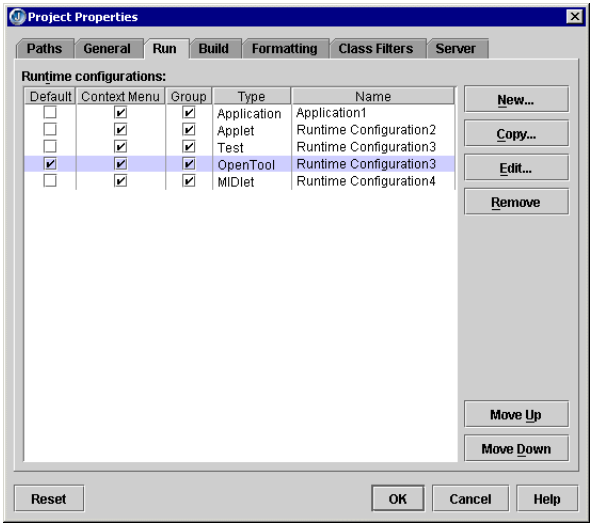
Build System

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

The Build System is responsible for scheduling a series of tasks when 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.

Runtime System

Packages: com.borland.primetime.runtime, com.borland.jbuilder.runtime
Classes: Runner, JavaRunner, RuntimeManager



Available runtime configurations are registered with a PrimeTime `RuntimeManager` as implementations of the `Runner` interface. Each `Runner` provides a name, a configuration panel, and behavior for running and debugging. In JBuilder, projects can be configured to run in any of a variety of ways using either the Run tab of the Project Properties dialog or the Configurations dialog under the Run menu.

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. Runtime configurations for applications, applets, JSP's, MIDlets, and one that's just for OpenTools are installed by JBuilder. These all use a standard runtime and debugging framework.

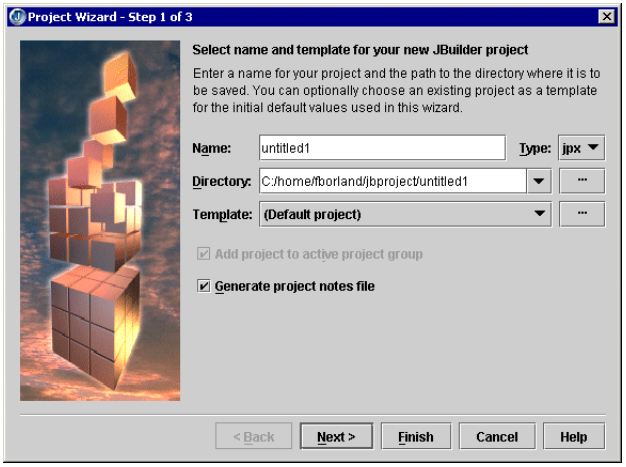
Debugger

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

Manipulating Java code

Wizard Framework

Documents: Chapter 15, "Wizard concepts"
Packages: com.borland.primetime.wizard
Classes: BasicWizard, BasicWizardPage, WizardManager



The PrimeTime wizards all share a common look 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`.

Java Object Tool (JOT)

Classes: JBProject

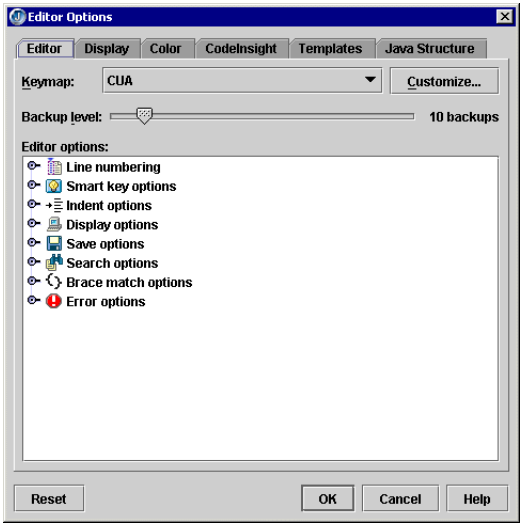
These tools provide a full parsing and code generation engine. These are what make JBuilder's Two-Way Tools and the Wizards framework possible. In some ways they act like Java reflection for source code, but with a full API for making modifications. Those familiar with the Document Object Model for XML should feel right at home with the basic concepts.

JOT can be used in a read-only fashion, in which case it can operate on either source code or class files. It can also be used to generate new source files, a facility used by many of the standard JBuilder wizards. The true power of JOT lies in making precise modifications to existing code, preserving existing formatting and comments exactly while making exacting changes to any aspect of the source code. This facility is used by JBuilder's visual designer, by BeansExpress, and by some of the more powerful wizards. JOT's full functionality is also available to OpenTools.

The user experience

Properties System

Documents: Chapter 12, "Properties system concepts"
Packages: com.borland.primetime.properties
Classes: Property, PropertyGroup, PropertyManager, PropertyPageFactory, 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.

Help

The JBuilder help system uses an HTML browser and an indexing scheme based on JDataStore technology.

Utility Classes

There are a variety of utility classes included in both PrimeTime and JBuilder that can help OpenTools maintain a consistent look and feel. The simplest is to ensure that all tree controls in your OpenTool descend from `SearchTree` to enable the "type-to-search" mechanism.

Chapter

2

OpenTools basics

Every time the IDE starts it dynamically discovers the available OpenTools and gives each one a chance to perform its own initialization at the appropriate time. 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 manifest file.
8. Test your new OpenTool.
9. Create an OpenTools JAR file.
10. Add an OpenTools JAR to JBuilder.

For additional information, see the borland.public.jbuilder.opentools newsgroup (`news://newsgroups.borland.com/borland.public.jbuilder.opentools`).

Create a new project

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.jpr` is the location of the project file. Click the Finish button to create the project.

You can actually define any number of OpenTools in a single project.

Add the OpenTools SDK library to your project

Select Project | Project Properties, click on the Required Libraries tab, and press the Add button. Select the OpenTools SDK library, and press OK to add it to your project. Click OK again to save your changes.

This library contains all of the classes that comprise the OpenTools API, so any meaningful OpenTool will need to use at least some of these definitions to extend the IDE.

Create a new Java class

Select File | New Class and replace the default package with `example` and the class name with `HelloOpenTool`. Click OK to create the new class definition and add it to your project.

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.

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, 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.

The name *PrimeTime* was our original code name for the all-Java IDE project, and it lives on as a 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;

    // Add a new menu item to the Help menu
    JBuilderMenu.GROUP_Help.add(new BrowserAction(
        "Say Hello") {
        public void actionPerformed(Browser browser) {
            JOptionPane.showConfirmDialog(null,
                "Hello, World!");
        }
    });
}
```

The first statement ensures that your OpenTool is being used with a compatible version of the OpenTools API before performing any initialization. The second statement adds a new menu item to the JBuilder help menu.

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 `AbstractAction` and overrides an abstract method to define the behavior of the new action. This newly created action is then added to JBuilder's help menu group.

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. You may also want to save your work at this point.

Create an OpenTools manifest file

For JBuilder to find your OpenTool, you must provide a description that includes the class name. Select Project | Add to Project | Add Files/Packages from the menu and specify the filename `c:/HelloOpenTool/classes.opentools` for your newly created file. (Note that if you created your project elsewhere you will want to adjust for the path you chose. Also note that the name "classes.opentools" assumes that your project's outpath is named "classes" and must be adjusted if you use a different outpath name.) Press OK on the resulting dialog to indicate that you want to create a new file.

Note: You've just taken advantage of a feature that allows you to create any file, including projects or Java source code, without using the Wizards in JBuilder. This can be extremely useful once you get used to it.

Double-click the newly created file in the project pane and add the following line to it:

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

Manfiest files must include a newline character at the end of every line, so be sure to press enter when you are done! This file will serve as the manifest file for a JAR created in the next step. You can place as many fully-qualified OpenTools classes on a single line as you like, using spaces to separate the class names.

Test your new OpenTool

Save everything you've been working on before proceeding. This step can be somewhat awkward as it involves editing the script used to launch JBuilder. It's great for development purposes, but not necessary for deployment of a finished OpenTool.

Exit JBuilder and edit the launch script in JBuilder's `bin` directory. This will be the `jbuilder.config` file on Windows, or a shell script on Linux or Solaris. Change the CLASSPATH definition to add `c:/HelloOpenTool/classes`. (Note that if you created your project elsewhere you will want to adjust for the path you chose.)

Now launch JBuilder again, and look for the Help | Say Hello menu item. If it doesn't appear, check to make sure you've followed these steps carefully. Your OpenTools classes must be in the directory added to the classpath, and the manifest file created in step 7 must be in a file whose name is the same as the directory on the classpath with an additional `.opentools` suffix.

Once you've verified that everything works as expected, remove your changes to JBuilder's CLASSPATH definition.

Create an OpenTools JAR file

There are tools in the Developer and Enterprise versions of JBuilder that automate the creation of a JAR containing your classes, but for this tutorial we'll use the command-line tools provided with the JDK.

Open a shell window or DOS prompt, make sure the JDK's `bin` directory is in your path, and make `c:/HelloOpenTool` your current directory. (Note that if you created your project elsewhere you will want to adjust for the path you chose.) Now type the following command and press enter:

```
jar -cfm HelloOpenTool.jar classes.opentools
    -C classes example
```

Voilà! You've created an OpenTools JAR that contains everything needed to extend JBuilder with the thrilling "Say Hello" menu item. Now to install it the professional way...

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
- Add new file types
- Overhaul the editor's keybindings
- 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

Build system concepts

General description

When a build is launched in the IDE, 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`, and if so, then 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 of how to use `Builder`s, see the Obfuscator sample in the JBuilder `samples/OpenToolsAPI/Build` directory.

Detailed description of feature/subsystem

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 Ant:

- Build task - a task that gets executed as part of the build process. Build tasks cannot exist stand-alone; they must belong to a target. If more than one build task belongs to a target, 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 target is 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 - a preexisting target that PrimeTime creates every time a `BuildProcess` is created. 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 serve as high-level markers for the build process, providing a broad framework into which `Builder`-created targets can be made dependencies.

Phases

JBuilder 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 JBuilder, 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());
}
```

Builders must be listed in the "Build" OpenTool 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:

```
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;
}
```

The above 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 will be created, and a new instance of MyBuildTask will be added to the target. If the target already exists, a new instance of MyBuildTask will be 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.

We have not yet established when in the BuildProcess that MyBuildTask will execute. See "Establishing dependencies" on page 3-6.

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 BuildTasks 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 3-8 for more details.

Writing a build task

As the JBuilder 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 a JNI target for the above code to work. If there isn't a JNI target, the above code does nothing; but if there is a JNI target, it becomes 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()`:

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

Do this registration in your `Builder`'s `initOpenTool()` method. 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 (rebuild); 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 the "isMakeable() and isCleanable()" on page 3-7 section 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 an IDE 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 doing a run, debug and optimize.

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, JBuilder 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 3 abstract methods:

- `getNodes`—the node(s) that 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:

```
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 an interface that some nodes implement. In "General description" on page 3-1 and "Registering a Builder" on page 3-4 we said that the children of each node are recursively passed to all the `Builder`s. In "Registering a Builder" on page 3-4 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 the `BuildProcess` methods: `getProperty()`, `setProperty()`, and `removeProperty()` methods. 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 2 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 3-7 for more performance considerations.

Using an existing Ant build task

It is possible to use existing Ant build tasks within the IDE 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 UI, in the Project Properties dialog, on the Build | General tab, by the **Autocancel build on error** checkbox.

Build extensions

The `Builder` class has two methods that are used by the IDE 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 Nodes that have multiple file extensions registered.

You can fine tune which files are displayed underneath a automatic package nodes by implementing getBuildExceptions. This function returns a Map, where the keys in the Map are com.borland.primetime.vfs.Urls, and the values are Booleans. 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 JBuilder, 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

Builders that process com.borland.jbuilder.node.PackageNodes should not process excluded package nodes—excluded packages are by definition excluded from the build process. If your Builder processes PackageNodes, 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 `fireBuild...` 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 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 JBuilder 8
 */
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, for example, 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.

Chapter

4

OpenTools loader concepts

General description

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 JBuilder: 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 be public and it must 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.

Detailed description

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. 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 4-5.

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 JBuilder 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 by JBuilder.

OpenTools API versions

The two parameters passed to an OpenTool are the major and minor version numbers. 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
  }
}
```

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 JBuilder, 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 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, and 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 an exclamation point is treated as a request to suppress the OpenTool of the specified class name:

```
OpenTools-Core: -examples.opentools.OpenToolOne
OpenTools-Core: !examples.opentools.OpenToolTwo
```

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.

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. 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 JBuilder 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. These commands will be passed everything between their command line option and the following option as a parameter when invoked. 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 brief 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.

The 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, starting 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.zip	c:\classes.zip.opentools

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

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:\jbuilder30\jdk.2\lib\jpda.jar
Scanning manifest from X:\jbuilder30\lib\AwtMotifPatch.jar
Scanning manifest from X:\jbuilder30\lib\beandt.jar
Scanning manifest from X:\jbuilder30\lib\DBCSpatch.jar
Scanning manifest from X:\jbuilder30\lib\dx.jar
Scanning manifest from X:\jbuilder30\lib\jbcl.jar
Scanning manifest from X:\jbuilder30\lib\jbuilder.jar
Scanning manifest from X:\jbuilder30\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 to register keymaps and other specialized behavior.

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

- 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

5

Browser concepts

General description

`Browser` is the main 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 4, "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* or 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 toolbar buttons. The IDE window that has the current focus can optionally provide its own context sensitive implementation of these actions. These are the delegate actions:

- DELEGATE_Undo
- DELEGATE_Redo
- DELEGATE_Copy
- DELEGATE_Paste
- DELEGATE_Delete
- DELEGATE_SelectAll
- DELEGATE_SelectCombo
- DELEGATE_SearchFind
- DELEGATE_SearchReplace
- DELEGATE_SearchAgain
- DELEGATE_SearchIncremental
- DELEGATE_ContextMenu
- DELEGATE_CustomEditorAction
- DELEGATE_Refresh

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.

Detailed description of feature/subsystem

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. All the native menus of JBuilder are registered at application startup before the *OpenTools discovery* process begins.

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 funnel 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 a `NodeViewer` class) 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 lists for two types of listeners:

- One list contains listeners that listen for events that *any* instance of `Browser` fires.

  Use `addStaticBrowserListener()` or `addStaticProjectGroupBrowserListener()` to listen for all `BrowserListener` events.

- One list contains listeners that listen for events that a *particular* `Browser` instance fires.

  Use `addBrowserListener()` or `addProjectGroupBrowserListener()` to listen for the events fired by a particular `Browser` instance.

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());
  }
});
```

Chapter

6

Content manager concepts

General description

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 or right of the content pane 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.

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 close, 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 all works together. 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 if an instance of TextFileNode (and is ultimately asked to create a TextNodeViewer which becomes the "Source" 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 another registered factory.

Detailed description of feature/subsystem

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 | IDE Options dialog box and using the File Types panel.

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) {
      ft.registerFileType("mytxt", FileType.getFileType("txt"));
    }
  }
}
```

If you don't associate your new file 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);
  }
}
```

NodeViewerFactory OpenTools registration

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(), true);
  }
}
```

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 FileNode and TextFileNode

If your file type isn't text or if you choose to provide your own editor, extend `FileNode`. Your implementation should look like 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 user opens a file in JBuilder, 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 notifying you that the file buffer has changed `only if` your viewer is active, then using that saved state when the viewer is activated to know if the content must be updated.

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 right-click on a file tab. 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.getActiveProject() != 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 15, "Wizard concepts."

Chapter

7

ProjectView concepts

General description

The `ProjectView` is an IDE panel also called the project pane. It consists of a toolbar and a tree. Any number of projects can be open, but there can be only one active project at a time. The tree provides a hierarchical display of the nodes that make up the currently active project, plus, if a member of a project group, those of any sibling projects.

Each tree entry is, or is subclassed from, one of the following node types:

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

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.

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.

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 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`.

Detailed description of feature/subsystem

Getting a ProjectView reference

Obtain a `ProjectView` reference from an instance of `Browser`. To be sure of getting the correct instance, try 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 launched from a menu or popup context menu:

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

Hiding and revealing the ProjectView

You can hide and 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. It 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.

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 15, “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 Project wizard and let the user decide whether to create a new project.

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

```
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);
}
```

Chapter

8

StructureView concepts

General description

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

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

Detailed description of feature/subsystem

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 `com.borland.primetime.node.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 FileNode and TextFileNode" on page 6-4.

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.

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) {
    }
  }
}
```

Chapter

9

StatusView concepts

General description

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

Detailed Description of feature/subsystem

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);
  }
});
```

Chapter

10

MessageView concepts

General description

The `MessageView` is an IDE panel called the message pane. It 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 the message pane, a tabbed interface separates 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 displays using its defined text, background color, foreground color, font, icon, 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. When the user single-clicks a message in the message pane, the `selectAction()` method is called. 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.

Detailed description of feature/subsystem

To get access to the `MessageView` for a particular Browser window, call the `Browser` `getMessageView()` method. The `Browser` also provides the `isMessagePaneVisible()` and `setMessagePaneVisible()` methods that are called when the user selects the View | Messages menu command, in the JBuilder IDE.

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");
```

The `MessageCategory` is always passed as a parameter in either one of the many variants of the `addMessage()` method or in the `addCustomTab()` 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 `MessageView`'s `addMessage()` method would result in an exception. To create a custom tab, call `MessageView`'s `addCustomTab()` method, passing to it the `MessageCategory` and the custom component. For example,

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

Using a MessageView tab

Unless you're using a custom tab component, tab 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(MYMESSAGES, new Message("This is test."));
mv.addMessage(MYMESSAGES, 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(MYMESSAGES, 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, highlight the UI element associated with that `Message` (and also cause it to become the active viewer if it's 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, move the focus to the UI element, in the `NodeViewer`, associated with that `Message`.

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.

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

11

JBuilder Designer/CMT OpenTools concepts

General description

This concept document explains how Designers and the Component Modeling Tool (CMT) works in JBuilder.

Designers are visual tools used for assembling the elements of a user interface (UI) quickly and easily. The Designer subsystem consists of a component palette, an Inspector, one of four designers, and a component tree. To access the design tools, open a source file in the content pane, then click its Design tab. (NOTE: The Design tab appears only if the source file inherits `java.awt.Container`.)

The JBuilder designers include:

- UI Designer - represents UI elements and their container relationships in the structure tree
- Menu Designer - represents menu/submenu hierarchy in the structure tree
- Data Module Designer - represents DataSets and their columns in the structure tree
- default designer - lists any subcomponents not claimed by any other designer to the structure tree under the "Default" folder

The Designer framework uses CMT to discover the subcomponents and property settings that make up a Java file. This information is used to determine relationships between subcomponents, to add nodes to a structure tree representing those relationships, and to provide a visual

representation of the file's UI subcomponents and relationships in the UI Designer.

The Designer framework can be extended in a number of ways. The most obvious is that property editors and customizers included with JavaBeans are automatically exposed through the designer. Developers can use CMT functionality to enhance their custom property editors.

Less obvious ways of extending the Designer subsystem include:

- Allowing visual manipulation of custom layout managers. The Java layout manager interfaces do not provide enough information to allow visual editing of layout constraints, so JBuilder introduces a "layout assistant" interface. JBuilder comes with assistants for the standard Swing and AWT layout managers. For custom layouts, a developer may wish to write an OpenTool layout assistant for integrating the custom layout into the property editor.
- Defining a custom designer which is automatically integrated into the overall designer framework. The menu designer and column designers are examples of this mechanism. Your custom designer automatically shares the component tree, inspector, and code generation with the user interface designer.

Detailed description

Architecture

The Open Tool that controls the designer subsystem is `com.borland.jbuilder.designer.DesignerManager`. Each designer that makes up the system is an Open Tool that must register itself with the `DesignerManager`. The `DesignerManager` registers the four designers included with JBuilder itself rather than having them mentioned in the manifest file; therefore, the initialization can be delayed until the user clicks on the Design Tab for the first time. The `LayoutAssistants` used by the UI Designer are also registered in this way as well.

To register a custom designer:

```
public static void initOpenTool(byte majorVersion, byte minorVersion)
   MyDesigner myDesigner = new MyDesigner();
   DesignerManager.registerDesigner(myDesigner);
}
```

Other major pieces of the designer subsystem are the Inspector and all the property editors that it uses, and the component palette. The Inspector listens to selection change events in the structure view to know which subcomponent needs its `CmtPropertyStates` displayed. The `ComponentPalette`

provides a customizable multi-page toolbar of components that can be visually designed.

Component Modeling Tool (CMT)

CMT is the tool by which subcomponents and property settings for a source file are discovered. CMT resides in `com.borland.jbuilder.cmt` as a collection of interfaces.

CMT uses the Java Object Toolkit (JOT) to find all the subcomponents in the file under design, and to determine the properties and events that each type of subcomponent exposes. For each subcomponent, the method calls made to it in the file's `jbInit()` method are discovered and stored in an array in the subcomponent. In particular, the property settings are processed so that each property setting is associated with a `CmtPropertyState`.

Through introspection or the information in a BeanInfo file, `CmtComponent` determines the properties (encapsulated in `CmtProperty`) that the component supports. `CmtPropertyStates` are the value of a particular property for a particular subcomponent, whether or not they have an associated `CmtPropertySetting`.

The `CmtComponent` for the file under design is a special case. Most of the `CmtComponent` instances are for representing the types of subcomponents used in the file under design. The file under design will be the only one that uses the methods dealing with `CmtComponentSource`. The `CmtComponent`Source file (the file being designed) uses the `CmtComponent` of its superclass to determine the properties that it exposes.

The first call to `CmtComponentSource.getSubcomponents()` causes `buildSubcomponents()` to be called. This method uses JOT to determine all the fields that make up the source file, and then gets the `CmtComponent` for each type and creates a `CmtSubcomponent` for each field. There is a `CmtComponentManager` that maintains a collection of all the `CmtComponent` instances this project has processed, so that extra reading of BeanInfo and component class files can be avoided.

Since events are rarely listed explicitly in a BeanInfo, reflection is almost always used to determine the events. Using reflection to determine a JavaBean's properties or events requires walking the superclass hierarchy, so `CmtComponents` for all the superclasses are created as well. Creating a `CmtComponent` for all the classes used in the bean being designed is one of the most expensive portions of the design process. As a way to speed up the process of creating `CmtComponents`, beans that do not do anything out of the ordinary in their BeanInfo have their property and events info written to a .pme (Property, Methods, and Events) file in the `[userhome]/.jbuilder/pme` directory so that the next time they are encountered, one tiny simple

file is processed to create the CmtComponent instead of having to examine the entire superclass hierarchy.

Accessing CMT

Assuming that you have obtained an instance of JBProject (that is, the current JBuilder project file), you can retrieve a CmtComponentManager instance that will allow access to CMT functionality through:

```
JBProject project; // this is a valid non-null instance of JBProject
CmtComponents cmtComponents = (CmtComponents) project.getComponents();
```

To obtain the file currently under design, first retrieve the current DesignerViewer node viewer, then:

```
CmtComponentSource componentSource = designerViewer.getComponentSource();
   CmtSubcomponent subcomponents[] = designerViewer.getSubcomponents();
```

You can manipulate the methods, properties, and events of a subcomponent, as well as retrieve a live instantiation of the component or determine how the subcomponent is nested.

Annotation

Designers use the CMT data structures to find the subcomponents of interest to it, usually based on the type of subcomponent. It can look through a subcomponent's method calls for help in determining how subcomponents are nested.

Component palette

Each DesignerViewer instance creates its own instance of the palette UI. There is one ComponentPaletteManager (com.borland.jbuilder.designer.palette.ComponentPaletteManager) which owns the models used by the palette and manages the reading/writing of the palette.ini.

Inspector

The Inspector displays the properties for the selected component and provides property editors for changing properties. There are two PropertyGrids which model the propertyStates and EventStates of the selected component. The Inspector listens to the TreeSelection from the structure view to know when to refresh its model.

The grid consists of two columns, the first being the name of the property/event and the second being the text representation of the value of the property/event as maintained by the propertyState. Cell editors for the value column are dynamic. They are based on the property editor

associated with the CmtProperty of each CmtPropertyState. They all extend DefaultCellEditor.

Cell editors use their cell's associated java.beans.PropertyEditor to perform their edit on the appropriate piece of Java code. The cell editors are also responsible for supplying the various "special knowledge" property editor extensions with the data they require by making the extra method calls provided for in their interfaces.

CmtPropertyEditor provides the CmtPropertyState to the editor so that it can know about the CmtSubcomponent, CmtComponentSource, etc. This setPropertyState() method is called prior to setValue() and again with null after the cell edit is over.

CmtPropertyEditor2 also provides the DesignerViewer instance via its setContext() method that allows the property editor to know about the project as well as providing a method for setup and cleanup.

CmtPropertyEditors should be careful that they only perform their expensive operations when their propertyState is non-null, which is usually the case unless they store some other data and forget to null it out when their propertyState is reset to null by the cell editor.

PropertyEditors are shared often, therefore they should not attempt to store data from one edit session to another. The cell editors will make calls to getTags() and getCustomEditor() when CmtPropertyState is null as part of their determination of which cell editor to use. For a CmtPropertyEditor that is a tagEditor and whose propertyState is null, getTags() should return an empty array, leaving the determination of what to place in tag list to the times that getTags() is called when its propertyState is non-null.

UI Designer

The UI Designer analyzes the subcomponents' add() method calls to decipher the structure of the UI. The UI Designer also substitutes the real layout property state created CmtComponent with a special layout property state of type LayoutPropertyState. Containers whose BeanInfo attribute isContainer() is false have their layout property suppressed. Another synthetic property state that the UI Designer sometimes makes is a buttonGroup property state for subcomponents that extend javax.swing.AbstractButton so that the user can specify the ButtonGroup of the button and get the ButtonGroup.add() method created as a result.

Property states whose name property is in brackets are treated slightly differently by the Inspector - the brackets and the first character are removed when forming the property name, and the property name is rendered in bold type. All such names appear at the top of the Inspector's grid, with the first character being used to determine the order of these special property states. This convention is used to distinguish synthetic properties from real ones.

The UI Designer uses `LayoutAssistants` to govern the use of nibs and component outlines based on the layout to which the container is set. The `LayoutAssistant` is responsible for establishing the constraints for newly dropped components, adjusting the constraints of moved components, and establishing constraints for each child of a container when the layout is changed.

Custom layout assistants

Occasionally developers might want to use custom layout managers and want to write a `LayoutAssistant` to incorporate the availability of the custom layout manager into the UI Designer and Inspector. Custom layout assistants must register themselves with the UI Designer specifically:

```
public static void initOpenTool(byte majorVersion, byte minorVersion) {
   UIDesigner.registerAssistant( {
      MyLayoutManagerAssistant.class,
      "com.borland.samples.opentools.layoutassistant.MyLayoutManager",
      true);
}
```

There is a basic layout assistant implementation available at `com.borland.jbuilder.designer.ui.BasicLayoutAssistant` that you may wish to extend, or your layout assistant can just implement `com.borland.jbuilder.designer.ui.LayoutAssistant` which defines the methods for the layout specific UI interaction model. Within your layout assistant you will have to define a property editor to be used for changing any properties your custom layout may provide such as horizontal and vertical gaps. You may also wish to define custom behavior for component outlines when they are being moved within a container that uses your layout manager. For instance, instead of component outlines bouncing from layout position to layout position, you can specify that the component outlines will track the mouse instead.

There is a sample layout assistant available in your `samples/OpenToolsAPI/layoutAssistant` directory. This sample illustrates how to write a layout assistant for a custom layout, register it with the UI Designer so that it shows up as a layout option in the Inspector, and customize the component outline behavior.

Chapter

12

Properties system concepts

General description

The PrimeTime 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 in their user's `user.properties` 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 or 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.

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`
- `MultipleNodeProperty`
- `ProjectAutoProperty`
- `ProjectAutoArrayProperty`
- `ProjectAutoArrayPropertyDefault`
- `GlobalProperty`
- `GlobalArrayProperty`
- `GlobalColorProperty`
- `GlobalIntegerProperty`
- `GlobalBooleanProperty`
- `GlobalFloatProperty`

Detailed description of feature/subsystem

Node-specific properties

These fall into two subtypes. 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 will cause its project to be "dirtied". These new settings are retained only after the project is closed *and* if the user chooses to save the project.

You typically define a project property by creating a public constant to represent the property and 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 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"));
```

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.

If you attempt to set a `NodeProperty` on a `Node` that 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.

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());
```

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.

Managing sets of properties with PropertyManager

Implementations of the `PropertyGroup` interface are typically registered by OpenTools in the UI category 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 indicates that the general IDE properties are being displayed.) 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.

```
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.

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.

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

Editor concepts

General description

The PrimeTime editor provides users a way to view and modify text documents. It also gives other tools, such as Code Insight and the structure pane, access to the text. The user communicates with the editor through keyboard and mouse input. The editor keybindings, as defined in Chapter 14, "Keymap concepts," translate the keyboard input into meaningful editor actions that manipulate the text in the editor.

Detailed description

The editor manager

JBuilder has a single instance of the `EditorManager` class. Its task is to keep track of most of the editor specific details. The `EditorManager` creates instances of the `EditorPane` class. It also fires events to those `EditorPanes` to notify them about global changes they should act upon, such as changes to the tab size, the font, the keymap, and so on. See the `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 the word "Attribute."

The `EditorManager` is also responsible for creating and managing keymaps. For more information about keymaps, see Chapter 14, "Keymap concepts."

Because other classes also must be notified about global editor changes, the `EditorManager` has support for adding and removing property change listeners. You can either add a class as the listener of a single property

change with the `EditorManager.addPropertyChangeListener(String, PropertyChangeListener)` method, or add a class as the listener of all property changes with the `EditorManager.addPropertyChangeListener(PropertyChangeListener)` method. The class that is added as the listener must implement the `java.beans.PropertyChangeListener` interface, which basically means you must implement the `propertyChange()` method:

```
void propertyChange(PropertyChangeEvent evt);
```

For example, this class implements the `PropertyChangeListener` interface; it's called each time the `EditorManager` fires a tab size property change:

```
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();
  }
}
```

The 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. The file opens in a new editor window. This editor window is an `EditorPane`. You can get a handle to the active editor window (the one with the focus) from `EditorAction` with code like this:

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

Open nodes in the browser that don't have the focus might also have an EditorPane associated with them. In this case you can use the getEditor() method of 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
  }
}
```

The document

Each EditorPane stores its text in an EditorDocument. To access that document, call EditorPane.getDocument(). Once you have an EditorDocument object, you can get to the document's actual text. The text is stored in objects of type javax.swing.text.Element. Each Element object basically represents one line and has a starting and an ending index. Each character of the document also has an index. Here is a code sample that demonstrates how to use Element and the indexes:

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

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

// How many lines are there in the document?
int totalLines = baseElement.getElementCount();

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

// Find the line the caret is on 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);
```

The caret

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

that work with the selected text as well. 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 highlighted selection?
int dot = caret.getDot();
int mark = caret.getMark();

if (dot != mark) {
  // There is an highlighted selection
}

// Alternate way to find out about selections
int selectBegin = editor.getSelectionStart();
int selectEnd = editor.getSelectionEnd();
if (selectBegin != selectEnd) {
  // There is an highlighted selection
}
```

The editor actions

The `EditorActions` class contains a set of actions for the editor. Learn about these actions by studying the Keymap examples in the OpenTools samples. You can find samples that show how to create a CUA key binding, an Emacs key binding, and a Brief key binding. The keymaps associate keystrokes with many of the actions defined in `EditorActions`.

When a new action is written, you must hook it up to a keystroke. Also, 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");
}
```

The text utilities

`TextUtilities` is another useful class. It has methods that you can use to move from a document index to a particular line or to find the beginning and end of words. If the text has hard-coded tab characters, methods exist that compute the width of the text, or that determine the index of a character in the text, taking tabs into account.

Chapter

14

Keymap concepts

General description

This concept discussion describes how keymaps work in JBuilder and how you can create, change, and extend them.

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 then ensures that the action fires when the user enters the proper keystroke. For all this to happen, the keymap must be the currently registered keymap in the component that has the focus.

You'll find it helpful to study some JDK classes to understand how keymaps are implemented in Swing. This is the list of classes to study:

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

Detailed descriptions

Keymaps in the JBuilder editor

JBuilder installs six keymaps: Brief, CUA, Emacs, Macintosh, Macintosh CodeWarrior, and Visual Studio. The default keymap is CUA. The user can change to another keymap through the Tools | Ide Options menu item.

Keymaps and OpenTools

Brief, CUA, and Emacs keymaps have been made into OpenTools examples, so you can learn how to create an entirely new keymap. Or you can use the OpenTools API to change one or more key bindings in an existing keymap.

Changing and extending keymaps

Basic keymap information

The `EditorManager` class does most of the basic keymap handling. The `getKeymap()` method returns the current keymap, and the `getKeymapName()` method returns the name of the current keymap. Use the `getKeymap(String)` method to retrieve an instance of a specific keymap by passing a String parameter such as `CUA` or `Emacs`. You can install a new keymap either by name or with a keymap instance with the `setKeymapName(String)` and `setKeymap(Keymap)` methods.

Notification of keymap changes

The `EditorManager` class also fires a change event when you install a new keymap. To install a class as a change event listener of the `EditorManager` class, use code such as this:

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

`myClass` is an instance of a class that implements the the `java.beans.PropertyChangeListener` interface; therefore, you should implement the `propertyChange()` method. Here is a simple code example:

```
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 you install a new keymap, it's easy to insert new key bindings or modify existing ones. For example, this code changes one of the key bindings:

```
keymap.addActionForKeyStroke(KeyStroke.getKeyStroke(KeyEvent.VK_0,
    Event.CTRL_MASK|Event.SHIFT_MASK), EditorActions.ACTION_MatchBrace);
```

The code is binding the Ctrl-Shift '0' (zero) key event (on English keyboards this is Ctrl-right parenthesis) to the editor action that finds the matching brace when the cursor is positioned at a brace (see the `EditorActions.MatchBraceAction` class). The `EditorActions` class lists the static instances of all the actions the JBuilder editor supports. The names of all those actions start with ACTION_, such as `ACTION_Backward`. The `KeyEvent` class of the JDK lists the key codes for all recognized keys. These key codes all start with VK_. 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. By studying the JDK `Keymap`, `KeyStroke`, `KeyEvent`, and `Event` classes, you should have enough information to tie any key combination to any `EditorActions` action.

Multiple key events for each keystroke

The JDK fires multiple key events for each keystroke. Usually this isn't a problem when you modify key bindings with `Keymap.addActionForKeyStroke()`, because things will work fine. If, however, you try to change the binding of keystroke with no modifiers, you might get unexpected results. For example, you might want to tie the y key to the Cut action, so you might write this:

```
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 happens because the 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_ key codes in your `addActionForKeyStroke()` method call, you are hooking into the pressed event and overriding its default action.

The typed event affects keystrokes that translate into printable characters only. The default action for typed events is to emit the character into the document. The released event is usually not important.

Why can we add actions for most keystrokes and have things work well? Because we usually want to add an action to a keystroke with the Alt, Ctrl, or Meta modifier, and those key events usually aren't printable, so they don't generate a typed event. You simply override the pressed event and you're done. For printable keystrokes, you'll get what you want if you hook into the typed event with code like this:

```
keymap.addActionForKeyStroke(KeyStroke.getKeyStroke('y'),
                             EditorActions.ACTION_Cut);
```

The code still calls the cut action, but it doesn't emit the y character into the document.

Writing your own action

You might find the supplied editor actions insufficient. It's easy to write your own action class and bind it to any key event.

Derive each new action class from the appropriate `Action` class. The `EditorAction` class can be the parent class of most new actions. If you're writing a Brief- or Emacs-specific action, derive your action from the `BriefAction` class or the `EmacsAction` class.

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");
}
```

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 must add code similar to one of the following sequence:

```
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");
```

More advanced keymap functions

The JDK Keymap class has some additional interesting methods:

- 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()`. You can then restore the original action later.

- public Action getDefaultAction() and public void setDefaultAction(Action a)

  The default action is called when there is no binding for the current keystroke. These methods are 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()

  Removes all the bindings in the current keymap. It's especially useful when you switch keymaps.

Removing key bindings

You can remove key bindings from a keymap, either individually or *en masse*. Removing a key binding, however, doesn't mean that a key event doesn't have an action associated with it any more. It simply means that there's no action associated with the key in the current keymap. Keymaps are stacked on top of each other; 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; JBuilder overrides this with the `PageUpAction`. If JBuilder removes that binding to `PageUpAction`, the Page Up key triggers the default JDK action again.

To simulate removing a key binding completely so that the keystroke fires no event, you can tie it an action that does nothing. For example,

```
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

Using the sample keymaps

To create a brand new keymap, it's usually easiest to take one of the keymaps in the samples and modify it. The UserCUA keymap sample is useful for basic keymaps that mostly bind keystrokes to existing `EditorActions`. If you need to create a keymap with subKeymaps, the `UserEmacs` keymap works well. The `UserBrief` keymap contains a lot of editor manipulation code, so it's a good one to learn from.

Naming the keymap

Always give your keymap a unique name. Don't try to install a new keymap called Emacs, Brief, or CUA, because those names conflict with the keymaps that JBuilder installs by default. You name your new keymap in two places in the `initOpenTool()` method, and that same name is what shows up in the Tools | IDE Options dialog where you change keymaps.

SubKeymaps

The UserEmacs keymap implements `subKeymaps`. Pressing `Ctrl-x` in Emacs mode switches the user to a different keymap (the `Ctrl-x` keymap), and that new keymap is called a `subKeymap`. Another way to look at this is to say that Emacs has states, and that `Ctrl-x` switches you to a different state with a new keymap. If you follow the UserEmacs keymap example, you should have few problems implementing `subKeymaps`.

Manipulating the editor

Both the UserEmacs and the UserBrief samples have code that shows you how to access the `EditorPane`. Through the `EditorPane` you can access the document and the caret. You should find enough code in those samples to see how to perform most basic editor manipulations.

Improved keymaps in JDK 1.3

In JDK 1.3, keymaps changed. 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 because the keymaps wrap the old behavior around the new behavior. You can, however, 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

15

Wizard concepts

General description

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 Wizards menu 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, you must perform at least these steps:

- Register a `WizardAction` with the `WizardManager`.
- Override methods in the `BasicWizard` class.
- Provide one or more `BasicWizardPage` panel objects used to gather user input.

Like other OpenTools, wizards are introduced into JBuilder 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 JBuilder and acts as a factory when an instance of that wizard is needed.

The entire JBuilder IDE is based on the Swing component architecture. All wizard UI components must be Swing-based.

Detailed description of feature/subsystem

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 cookie 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 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 Wizards menu, the small icon is used and the description appears on the status bar when the mouse is over the wizard menu item.

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();
}
```

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 panel user interface.

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

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 `Browser` reference 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 != null) && (node instanceof JavaFileNode));
  }
  protected Wizard createWizard() {
    return new MyWizard();
  }
};
```

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. **Important:** 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);
    }
  }
}
```

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. An easy way to 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);
  }
}
```

In a multiple step wizard, the user might use the Back button to revisit a prior step. It's also possible that you might want to enable the Finish button before the last step, thereby allowing the user to accept all the defaults for 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 would probably reside there as well.

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.

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.

Using the Tools | IDE Options dialog box, switch between alternative Look & Feel settings and verify that the new wizard still has an acceptable layout and colorization in each.

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. The icon `BrowserIcons.ICON_DOTDOTDOT` is used for a button to launch a dialog providing content for the field with which it is associated. 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:"));
```

Depending on your wizard, you may be able to do some useful early testing 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

16

UI package concepts

General description

The `com.borland.primetime.ui` package contains many support classes that can be useful not only in building your user interfaces but also in helping you to duplicate the Look & Feel of the IDE within your own wizards. The majority of these classes are public yet they aren't fully documented as part of the OpenTools API. All of the classes are Swing-based. These are the ones discussed here:

- `ButtonStrip`
- `ButtonStripConstrained`
- `CheckTree`
- `CheckTreeNode`
- `ColorCombo`
- `ColorPanel`
- `CompositeIcon`
- `DefaultDialog`
- `DialogValidator`
- `EdgeBorder`
- `ImageListIcon`
- `LazyTreeNode`
- `ListPanel`
- `MergeTreeNode`
- `SearchTree`
- `Splitter`
- `VerticalFlowLayout`

Detailed description of feature/subsystem

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 you can change this with constructor parameters or by using either the `setAlignment()` or `setOrientation()` methods after constructing `ButtonStrip`:

```
ButtonStrip buttonStrip = new ButtonStrip(FlowLayout.RIGHT, 5);
buttonStrip.add(okButton);
buttonStrip.add(cancelButton);
```

The one drawback of `ButtonStrip` is that its `FlowLayout` doesn't 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 better use of the available space and clips buttons that don't 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 fires 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 drop-down when clicked. The control fires `ActionListener` events when a value is selected. It doesn't 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 is that of the tallest icon in the array and the icons are 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 user closes the dialog by clicking the OK button, and if you provided an optional component using the `DialogValidator` interface in your `showModalDialog()` call, the `DefaultDialog` invokes 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

The `EdgeBorder` class implements the `Border` interface, providing a two-pixel inset where shadow and highlight 3D 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 your children when the user asks for them. When that happens, your implementation of the `createNodes()` method is called to produce them. For example,

```
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

`ListPanel` is an abstract class and that extends `JPanel`. It provides a scrolling `JList` next to five buttons which are by default labeled “Add...”, “Edit...”, “Remove”, “Move Up”, and “Move Down.” Change the first three labels 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

`MergeTreeNode` extends `javax.swing.tree.DefaultMutableTreeNode`. Use `MergeTreeNode` if you have a `JTree` that you need to update while still preserving any node expansions. Usually you do this in implementations that provide structure pane content. 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) {
    }
  }
}
```

SearchTree

`SearchTree` extends `JTree`. When you enter a number or 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* expands the node. The *Enter* or *Esc* key cancels the window. For this to work, each TreeNode must supply the same text in its `toString()` method as it is displaying 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);
  }
}
```

Splitter

`Splitter` extends `JPanel`. It is intended to hold one or two `JComponent` objects. If there is more than one, a draggable bar that is used to adjust the allocation of space between components appears. The components can be oriented vertically or horizontally.

```
SplitterTestFrame f = new SplitterTestFrame();
f.setBounds(100, 100, 500, 500);
f.show();

public class SplitterTestFrame extends JFrame {

  Splitter splitterA = new Splitter(true, Splitter.PROPORTIONAL, 0.8f);
  Splitter splitterB = new Splitter(false, Splitter.PROPORTIONAL, 0.2f);
  Splitter splitterC = new Splitter(true, Splitter.PROPORTIONAL, 0.5f);
  JButton pv = new JButton("ProjectView");
  JButton sv = new JButton("StructureView");
  JButton cv = new JButton("ContentView");
  JButton mv = new JButton("MessageView");

  public SplitterTestFrame() {
    try  {
      jbInit();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  private void jbInit() throws Exception {
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(splitterA, BorderLayout.CENTER);
```

```
    splitterA.setFirstComponent(splitterB);
    splitterA.setSecondComponent(mv);
    splitterA.setProportion(0.8f);

    pv.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        mv.setVisible(!mv.isVisible());
        splitterA.validate();
      }
    });

    sv.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        splitterB.setDividerSize(splitterB.getDividerSize() > 4 ? 4 : 10);
      }
    });

    cv.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        pv.setVisible(!pv.isVisible());
        splitterB.validate();
      }
    });

    mv.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        sv.setVisible(!sv.isVisible());
        splitterB.validate();
      }
    });

    splitterB.setFirstComponent(splitterC);
    splitterB.setSecondComponent(cv);
    splitterB.setProportion(0.2f);

    splitterC.setFirstComponent(pv);
    splitterC.setSecondComponent(sv);
    splitterC.setProportion(0.5f);
  }
}
```

VerticalFlowLayout

This class extends and mimics `FlowLayout`, but layouts out components vertically instead of horizontally.

```
int hGap = 5;
int vGap = 5;
int hFill = true;
int vFill = false;
setLayout(new VerticalFlowLayout(VerticalFlowLayout.TOP, hGap, vGap, hFill,
vFill));
```

Chapter

17

PrimeTime Util package concepts

General description

The `com.borland.primetime.util`package contains many support classes that may be useful in the design of your own PrimeTime OpenTools addins. 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`
- `FastStringBuffer`
- `JavaOutputStreamWriter`
- `OrderedProperties`
- `RegularExpression`
- `SoftValueHashMap`
- `Strings`
- `VetoException`
- `WeakValueHashMap`

Detailed description of feature/subsystem

AssertionException

This class extends `RunTimeException`. Generally you use `AssertionException` in combination with `Debug` to test and report self-check failures; however, you can use it by itself.

```
void doIt(String str) {
  try {
    AssertionException.assert(str != null, "null string input to doIt()");
  }
  catch (AssertionException ex) {
    ex.printStackTrace();
  }
}
```

CharsetName

`CharsetName` provides a routine which you can use 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`. Its purpose is to provide 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()];
  FastStringBuffer fsb = new FastStringBuffer(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 path name 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 re-compile.

`Debug` supports assertions. If `enableAssert()` has been used to enable them (assertions are on by default), a failed assertion results in an `AssertionException` being thrown. For example,

```
void doIt(String str) {
  try {
    Debug.assert(str != null, "null string input to doIt()");
  }
  catch (AssertionException ex) {
    ex.printStackTrace();
  }
}
```

`Debug` also provides support for tracing your code flow. If you call `addTraceCategory()` to enable a particular trace type, then messages, stack traces, or conditional warning messages will be output. This output can be either to `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");
}
```

Debug also supports the debugging of paint logic. After painting, you can call the `debugRect()` method to draw a color-coded rectangle around the area. For example,

```
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.

Example

```
static final String TRACE = "MyTrace";

void test(boolean bLog) {
  doIt(bLog ? System.out : new DummyPrintStream());
}

void doIt(PrintStream out) {
  out.println("my output");
}
```

FastStringBuffer

This class is an alternative to `StringBuffer`for those cases when you know a buffer is not shared. It avoids the complications of synchronization and sharing.

Example

```
String expandCRLF(String s) {
  FastStringBuffer fsb = new FastStringBuffer(s);
  for (int i = 0; i < fsb.length(); ++i) {
    if (fsb.charAt(i) == '\r') {
      fsb.removeCharAt(i);
      fsb.insert(i, "\\r");
    }
    else if (fsb.charAt(i) == '\n') {
      fsb.removeCharAt(i);
      fsb.insert(i, "\\n");
    }
  }
  return fsb.toString();
}
```

JavaOutputStreamWriter

This class extends `OutputStreamWriter`. It writes characters to an output stream, translating characters into bytes according to the given or current default character encoding. Characters that cannot be represented in that encoding appear as Unicode escapes.

Example

```
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. It writes international characters 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 save 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;
}
```

Strings

`Strings` contains string manipulation routines which generally fall into two groups. The first are convenience wrappers for `java.text.MessageFormat.format()`, which builds the argument array for you based on input parameters. For example,

```
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 and other characters such as delimiters. You can use the "standard" encoding or supply your own:

```
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"));
```

VetoException

`VetoException` extends `java.lang.Exception`. Wizards use it to indicate a condition that needs correction before moving to the next step or starting or completing the Finish button processing.

Example

```
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 save are put in `java.lang.ref.WeakReference`wrappers. Such references don't prevent their referents from being garbage-collected. Therefore a reference stored using `put()` might later be returned as `null` when it's retrieved using `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

18

Version control system concepts

General description

PrimeTime includes a general framework to integrate any version control system (VCS) and manage Java 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 the application for a given VCS.

A complete implementation of the VCS API is presented in the `SampleVCS` OpenTools sample. This document uses snippets of code from the `SampleVCS` class to illustrate examples of implementation of the VCS API.

Extending JBuilder

`SampleVCS` can be found in the `samples` directory and can be used as a guideline to integrate any VCS into JBuilder taking advantage of the History pane and the VCS Commit Dialog.

Detailed description of feature/subsystem

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 SourceSafe backend implementation.
- **clearcase** - ClearCase backend implementation.

To integrate a VCS into your IDE, 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 the IDE. It consists of just 19 methods to make your work as easy as possible. 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.

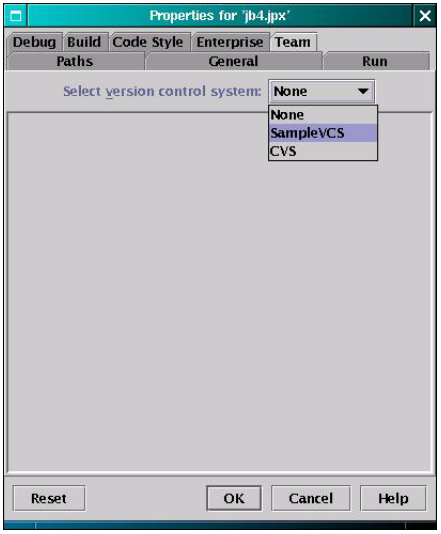
Configuration of the VCS

To make a VCS available to the IDE, the class must register itself with the VCS Factory (`com.borland.primetime.vcs.VCSFactory`). For example, in JBuilder:

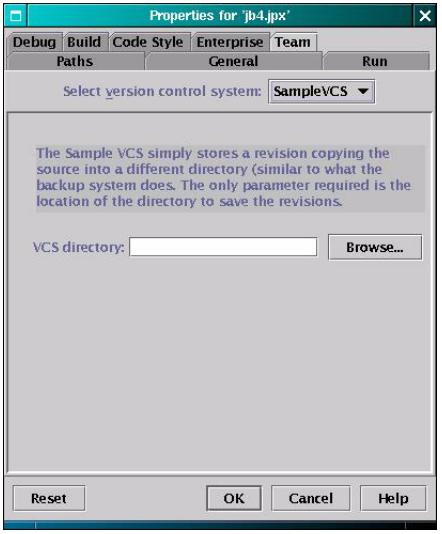
```
package com.borland.samples.samplevcs;

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 Team page in the Project Properties dialog box. Note that the `getName()` method returns the name that will be associated with the VCS.

This is the result in the Team property page in the first version of JBuilder that supported VCS integration:



The SampleVCS is added to the list of registered VCSs



The result of calling `VCS.getProjectConfigPage()`

To provide a configuration panel to the user, define a subclass of `com.borland.primetime.properties.PropertyPage` (which is 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 `com.borland.primetime.vfs.ui.UrlChooser`.

Extending JBuilder

JBuilder obtains this page by calling the `VCS.getProjectConfigPage()` method in JBuilder versions 5 through 8. For JBuilder versions past 8, this page is obtained by calling `VCS.getProjectConfigPageNew(Project project)`. It is therefore 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.

JBuilder 4 used to show the VCS property page inside the Project properties dialog box. This has changed in JBuilder 5 and now the page has its own dialog displayed by the "Team | Configure Version Control" menu option.

Saving project settings

Saving VCS settings, together with the project to make them persistent, is a Java-dependent function in the OpenTools. You can do this using the `com.borland.jbuilder.node.JBProject.setAutoProperty()` method. The method accepts three parameters, in this order:

1. The first is the *category* for the property. It should always be `VCS.CATEGORY`.
2. The second parameter is 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 last parameter is 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 the IDE 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<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 and easy to implement. The IDE calls the `VCS.getRevisions()` method to retrieve a `Vector` of `RevisionInfo` for a specified file. The method receives a `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) && (! name.endsWith(".comment"));
      }
    });
    int len = list.length;
    String revNumber;
    for( int i=0; i < 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, and 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 Strings. See `com.borland.primetime.teamdev.vcs.AbstractRevisionNumber`, `com.borland.primetime.teamdev.vcs.NumericRevisionNumber`, and `com.borland.primetime.teamdev.vcs.StringRevisionNumber`.

Note that care is taken to identify the revision that was used to check out the file in 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 the IDE'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 (execute) 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 PrimeTime 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.

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`.

Chapter

19

JBuilder JOT concepts

General description

JOT stands for the Java Object Toolkit and is a set of interfaces and classes used to parse and generate Java code. It's main use is the parsing of source or class files to extract information about their contents. For example, the Deployment wizard uses JOT to traverse files and determine what object types are actually used by a set of classes or source. Additionally, JOT can be used to generate code. Examples of this include the various Wizards in the Object Gallery (File | New) or the UI Designer.

Detailed description of feature/subsystem

The heart of JOT is a series of interfaces that describe the Java language in detail.

Accessing JOT

You are either working with a file that exists already or you are creating a new file when you are working with JOT. If the file exists, it is 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. To access files with JOT you need to get either a JotSourceFile or a JotFile object from the package manager. The package manager is represented by the JotPackages interface and you can get an instance of one from JBProject. The package manager will know about the files and packages that are available to the project and you use the package manager to access source files or class files.

```
JBProject project; // this is a valid, non-null instance of JBProject
String filename = "c:/dev/test/Unit.java";
JotPackages pkg = project.getJotPackages();
JotFile jfile = pkg.getFile(new Url(new File(filename)));
```

The above code would return a `JotFile` object for the file `c:/dev/test/Unit.java` if it exists and can be found in the classpath available to the project. If the file cannot be found, the `JotFile` object is null. You can use this mechanism to load a `.class` as well as a `.java` file, but usually work begins with source. A more common case (which is covered in the `jot` samples) gets the file Url from the project itself. The `JotFile` object that you have at this point can be accessed for reading or writing.

If you are creating a file, first make a `FileNode` from the Project. You can then extract a package manager from there for Jot to begin working. Don't forget to save the `FileNode` when you are done.

```
JBProject proj; // this is a valid, non-null instance of JBProject
String fullFileName = "c:/dev/test/Unit.java"

java.io.File f = new File(fullFileName);
if(!f.exists()){
FileNode jfile = proj.getNode(new Url(f));
// use JOT to write the contents of the file
jfile.setParent(proj);
jfile.save();
}
```

If you are creating a file or adding source to a file you must tell the package manager to `commit()` those changes. When you are done working with a file in JOT, you need to tell the package manager to `release()` the file. This frees the file for use by other subsystems.

How JOT sees a class

JOT 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.

Using JOT, you can do all that and go much deeper than using reflection alone. To allow it to parse Java code, JOT has representation for all of the Java constructs you might see in a source file. It also adds representation for the file that contains the class. These detail items follow the language guidelines laid out on the Java Language Specification (JLS); the JLS sections are cited in the Javadoc for the JOT classes:

- 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 `JotSourceFile`.
- Class: contains fields, methods, and inner classes. This is represented by `JotClass` and `JotClassSource`.
- Method: has a code block. This is represented by `JotMethod` and `JotMethodSource`.
- Code blocks: contains statements, other methods, and so on. It is represented by `JotCodeBlock` and its contents are `JotSourceElements` or descendents of that interface.

JOT 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 JOT to read Java

JOT can parse Java classes and Java source files. The methods specified by JOT 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 JOT. If you are parsing from Java source, you can extract a little more information than from a Java class. This is because some items represented in the source (such as comments) are stripped out at compile time and aren't represented within the class file.

During parsing, everything works from the outside in. At the outermost level (a `JotSourceFile`), the parsing procedure is simple. A `JotSourceFile` contains one or more classes (represented by a `JotClass` or a `JotClassSource`), 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.

JOT 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 Jot!");
```

```
    }

  // 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 `JotStatement`. This is very generic as many things are `JotStatements`, so if you are working with a bunch of statements with JOT, you would determine if the type of statement you had encountered was really an instance of `JotFor` using `instanceof`.

`JotFor` 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. `JotFor.getCondition()` returns a `JotExpression` 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 JOT:

```
package com.borland.samples.jot;

import java.awt.*;

  public class JotExample extends Component{
    static final int PI = 4;

    public JotExample(){
      // 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 `JotSourceFile` named `jsf` that represents this file. To get the package statement use this code:

```
String pkg = jsf.getPackage();
```

To get the import statements from the file, use this code and iterate through the array:

```
JotImport [] imports = jsf.getImports();
```

To get the class(es) from the file, use this code:

```
JotClass [] 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 `JotSourceFile.getClasses()`. Therefore, you can use this code to get the specific instance of `JotClass`:

```
JotClass jc = classes[0];
```

From the `JotClass` 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
int mod = jc.getModifiers();
JotType super = jc.getSuperclass();
JotType[] imps = jc.getInterfaces();
```

These methods allow you to find out the modifiers for the class, the super class and an array of interfaces implemented by the class. You could get more information about the superclass by using the methods in `JotType` to get a `JotClass` or `JotClassSource` that represents the superclass of the original object parsed.

Throughout JOT there are methods declared to return arrays of objects. If there is no value to be returned JOT returns an array of length 0. In our example, `JotClass.getInterfaces()` returns a zero length array of `JotType`. To avoid errors, check the length of any array that is returned before trying to use the objects.

Where modifiers are used in Java, as, for example a `public` class or a `static final` variable, JOT allows access through a `getModifiers()` method. `getModifiers()` returns an `int` value that you deconstruct by checking against the values in `java.lang.reflect.Modifier`. You can do this work yourself or use one of the getters in `com.borland.jbuilder.jot.util.JotModifers` or you can use the methods built into `java.lang.reflect.Modifier`. A note when working with the modifiers for classes. The bit that represents 'synchronized' is always set to true for a class. To eliminate any confusion when generating a String representation for the modifier, be sure to mask out this bit. For example:

```
JotClass firstClass; // this is a valid, non-null instance of JotClass
String c_mods = Modifier.toString(firstClass.getModifiers() &
~Modifier.SYNCHRONIZED);
```

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
JotField[] fields = jc.getFields();
JotMethod[] methods = jc.getMethods();
```

Taking this one level further demonstrates another important item in JOT: the need for casting. JOT uses interfaces to describe the aspects of Java. The implementation isn't available when using JOT, so you must write all your code against types that are really interfaces and not concrete instances of the objects where all the work is done.

When you use `JotClass.getMethods()` you get an array of `JotMethod` objects. The `JotMethod` interface describes the components that make up a method, such as the modifiers, the method name, any exceptions that it throws, and so on. It doesn't give you access to the contents of the method body itself. `JotMethod` has a child interface, `JotMethodSource`, that provides the access to the method body elements. You should be aware that generally methods will return the more generic (higher up in the inheritance tree) Interface type. This makes it easier when you are parsing or when you need to pass items as parameters to methods, but it also makes for a fair amount of casting when you need to get to specific details. To get the statements that make up the method in this sample, use this code:

```
JotMethodSource jms = (JotMethodSource)methods[0];
JotStatement[] jsa = jms.getCodeBlock().getStatements();
```

Or you could combine it all into one step and use this code:

```
JotStatement[] jsa =
    ((JotMethodSource)methods[0]).getCodeBlock().getStatements();
```

Of course, this extracts just the statements for the first method. Usually you would loop through the methods, and so on.

For a more complete example of using JOT to parse source see the sample `com.borland.samples.opentools.jot.read.ReadingSource`. This sample takes the selected file and gathers information about classes and methods using JOT. This information is sent to the message window inside the IDE. Another example of using JOT to parse source and/or classes is the sample com.borland.samples.opentools.jot.packagetree.PackageTree. This sample takes the selected node in the IDE, determines it's package and then builds a hierarchy tree for all the classes in that package.

Note: When you are finished reading the file, remember to tell the package manager to release() the file you were using.

Using JOT to write Java source

JOT can be used to write source as well. When writing source, you deal with much higher level items like methods or statements instead of the very fine-grained items available to you when reading source or class files with JOT. The methods specified by JOT for writing Java code use the naming convention `add<XYZ>` and `set<XYZ>` where XYZ represents the item you are creating or modifying. For example, `JotClassSource.addMethod()` adds a method to a given class. `JotMethodSource.setModifiers()` sets the modifier bits for that method.

Almost all items that you read and write with JOT implement the `JotMarker` interface. JotMarker is used to control placement of the items you are writing relative to other items already in code. When you are writing code with JOT, almost all of the methods will take a JotMarker as the first parameter followed by a boolean value followed by whatever data is needed for the item you are writing out (usually a String of some sort). The JotMarker is the item relative to which you are adding code. The boolean value controls the position of the added code. If it is true, the code is added before the JotMarker. If it is false, it is added after the JotMarker. The samples mentioned at the end of this section demonstrate this. Note: If you are writing code sequentially, as is the case with many Wizards, you can just use null as the value for JotMarker. The example code in this section does this.

Note: JOT is intolerant of being told to write illegal code and will throw a `JotParseException` once it has tried to read back anything you have written. If you are using JOT to generate code and your code doesn't appear, go back and make sure you are creating legal code.

As an example of writing code using JOT, 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.jot;

import java.awt.*;

  public class JotExample extends Component{
  }
```

Again, we begin with the outermost item, the source file itself. For JOT, the source file is represented by a `JotSourceFile` that encapsulates the package statements, the imports and the class declaration.

Assuming you have an instance of a `JotSourceFile` named `jsf`, use the following methods to create the class:

```
jsf.setPackage("com.borland.samples.jot");
JotImport ji = jsf.addImport("java.awt");
JotClassSource jcs = jsf.addClass(null, false, "JotExample", false);
```

```
jcs.setSuperClass("Component");
jcs.setModifiers(Modifier.PUBLIC);
```

As is the case with `getModifiers()` discussed in the section on parsing, `setModifiers()` works with `int` values. These values are taken from the `java.lang.reflect.Modifier` class. For this example, assume that there is an import statement for `java.lang.reflect.Modifer` in place, which allows the use of the short name of the class in the code.

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.jot;
import java.awt.*;

public class JotExample extends Component{
  static final int PI = 4;
  public JotExample(){
      // 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:

```
JotFieldDeclaration jf = jcs.addField(null, false, int, "PI");
jf.setInitializer("4");
```

Java allows for additional modifiers to be used together. You can do that with JOT by OR'ing the modifier values together when you pass them into `setModifiers()`. To make the field PI both static and final, do this:

```
jf.setModifiers(Modifier.STATIC | Modifier.FINAL);
```

To create the constructor and make it public, do this:

```
JotConstructorSource xtor = jcs.addConstructor(null, false);
xtor.setModifiers(Modifier.PUBLIC);
```

To set up the "shell" of the method with a parameter x, use this code:

```
JotMethodSource jms = jcs.addMethod(null, false, "boolean", "doesItRock");
jms.setModifiers(Modifier.PUBLIC);
jms.setParameter(null, false, "int", "x");
```

Add the local variable `ctr` and assign it the value of the parameter passed to the method. Because we're going to add many items to the code block of this method, create an instance variable to represent the code block:

```
JotCodeBlock rockBlock = jms.getCodeBlock();
JotVariableDeclaration jvd;
jvd = rockBlock.addVariableDeclaration(null, false, "ctr", "int");
jvd.setInitializer("x");
```

Add in the `for` statement. Because you'll be adding content to the body of the `for` statement, create an instance variable to represent the code block of the `for` statement. Note that the entire setup for the control portion of the `for` statement is passed in as a complete `String` instead of three separate JOT types.

```
JotFor jfor;
jfor = rockBlock.addForStatement(null, false, "int i = 0; i < ctr; i++");
JotCodeBlock forBlock = jfor.getCodeBlock();
```

Create the body of the `for` statement. In this case, you're just adding two `println` statements. Note that the statements you are passing in include their semi-colons at the end. This matches up with the third parameter in `JotCodeBlock.addStatement()` that specifies if a semi-colon is needed. If the value is set to true, JOT adds a semi-colon at the end of the statement for you.

```
String stmt1 = "System.out.println(\"JBuilder Rocks!\");";
String stmt2 = " System.out.println(\"Oh yes, it does\");";
forBlock.addStatement(null, false, false, stmt1);
forBlock.addStatement(null, false, false, stmt2);
```

Finally, add the return statement to the method.

```
rockBlock.addReturnStatement(null, false, "true");
```

For a more complete example of using JOT to write source see the sample `com.borland.samples.opentools.jot.write.WritingSource`. This sample generates a simple class file. Another sample of using JOT to read and write is `com.borland.samples.opentools.jot.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.

When you are done writing your file, be sure to have the package manager `commit()` and `release()` the file. If you were working with a node that you created in the project, it is also a good idea to have the file saved there, as well.

Chapter

20

JBuilder Server Plugin Concepts

JBuilder 4, 5 and 6 have had support for both App Servers (which primarily support EJBs) and Web Servers as two entities—with the App Server optionally containing a reference to a related Web Server. Since JBuilder 7, this support is streamlined so that a server can be both an EJB container service provider and Web container service provider; be able to provide one service and not the other; and support the addition of new services by third parties.

To accomplish this, we introduced a *generic* Server concept. In practice, `Server` becomes a base class for an App Server *and* a free standing Web Server. This allows us to treat the different kinds of servers similarly in situations where we need to show properties for them and list them in a common panel. Additionally, we combined the Web Run and EJB Run concepts, so that we only start one instance of a server and it supports both kinds of containers. The design of this subsystem also 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.

Some terminology:

Server	A particular version of a server, such as Borland Application Server 4.5, Borland Enterprise Server 5.0, Tomcat 4.0, or WebLogic 6.1

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 application (JSP and Servlet), JMS, deployment
Plugin	Classes written to the Server API to enable a particular server and its services to operate inside the IDE

Registering Servers and Services

The core API is in a new package, `com.borland.jbuilder.server`. 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 5.1.1 - 5.2 uses the native partition.exe, so its BES50xServerLauncher is a subclass of NativeServerLaucher, while Tomcat's Tomcat40ServerLauncher is a subclass of PureJavaServerLauncher.

Legacy Adapters

Legacy adapters are designed to wrap existing AppServer and WebServer plugins in the new Server API, to both minimize internal conversion work, and to allow external plugins to continue to operate under the new API. The AppServerManager and 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.

This legacy mechanism is in place so that the pre-JBuilder 7 plugins are still good. The old API is deprecated. When the 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 the wrapped object—whether it 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.

Configuration UI

By default, server information appears in two places in the JBuilder UI: in the Tools | Configure Servers dialog box, and in the Servers page of the Project Properties dialog box.

On the Tools menu

The new menu item "Configure Servers..." is on to the Tools menu. This allows users to configure Servers the same way that libraries/jdks are configured. Because of this,

- `Appserver.properties` is no longer pertinent—Servers are now stored as <server name and version>.library files in <user home> by default.
- In the list of Servers, the ones that haven't been set up yet will appear in light gray, with a tooltip that indicates they haven't been set up yet.

The Servers list shows any kind of Server object. This way the appropriate kind of properties panel can show up when a Server is selected in the configuration list.

On the Project Properties dialog box

The Server page of the Project Properties dialog box has two radio buttons to clearly differentiate between two approaches: the simpler situation where a single server does everything, and the more complex case where you can have any rational combination of modular services provided by different servers. The combo box lists all registered servers, from which the user can quickly select a server. In the Single Server drop-down list, any servers that are not properly configured are shown in red. The ellipsis (...) button to the right opens the Edit or Select Server dialog.

The list on the left always shows all registered service types. If the chosen server does not support a particular service, that service is greyed out. The checkbooks controls whether the service is enabled for this project or not. If a service is not supported, the checkbooks state doesn't matter—the service is disabled—but it is maintained in case the user decides to switch to modular services or back. Tooltips for the items in the list explain more about the items' 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 modular services disables the single server combo box and button, and enables the "Service Properties" fields instead. The button does the same thing, opening the Edit Server dialog.

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.

Non-GUI deployment option for OpenTools

If your Server plugin doesn't support a GUI deployment tool, change the value of `supportsGuiToolDeployment()` to `false`. This allows you to indicate that your Server plugin doesn't support a GUI deployment tool in a way that will disable the Tools | Enterprise Deployment... menu with an appropriate disabled message in the status bar.

By default this method returns true, and the `guiToolDeployment()` method is updated to be not abstract, and, by default, to bring up a message box indicating that there is no GUI deployment tool available for the current Server.

This combination of default implementations keeps the implementation familiar, and allows people who don't want to provide a GUI deployment tool to do nothing (if they so choose) and to get an informative dialog by default.

Chapter

21

JBuilder server configuration concepts

General description

In order for a Server to be available for use in JBuilder, it first has to be configured, or set up. 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, where all the Servers that JBuilder supports are listed and the user can select a Server to configure. When "OK" is selected 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 open tool implementation registers with the ServerManager, the Server will become part of the list of Servers in the "Configure Servers" dialog.

Detailed Description of Feature / Subsystem

History

Previously Servers used to be set up using the Enterprise Setup dialog. 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.

The "AppServers" tab no longer appears in the Enterprise Setup dialog. This concept is now replaced with the "Configure Servers" dialog.

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 translates as much of the previous API implementation as it can to the new API.

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. For Legacy support, if a valid AppServer implementation gets registered with the AppServerManager class, it will also appear in the "Configure Servers" dialog—all 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. 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.

Basic Page

The Basic 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 Basic page in the "Configure Servers" dialog.

Custom Page

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 to 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. Once created, this page will be preserved while the dialog 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 Basic page that corresponds 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 Basic page—for instance, when the home directory gets modified on the Basic 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 Basic page, if you need to, by implementing the `checkUpdateBasicConfigurationPage()` method. This method will get called whenever the user selects the Basic page after having selected the Custom page.

Legacy support

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:

- **Configuration legacy conversion info**
  - 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 Basic page. You can just suppress these controls, if you need to.
  - If you don't see a Custom page show 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 legacy conversion info**
  - If your legacy plugin overrides `updateDeploymentDescriptors()` in AppServerTargeting, now override the method with 4 arguments, instead of the one with 3—the one with 4 is the one that gets called.

New in JBuilder 9

Copy Server

In JBuilder 9 we introduced the concept of "Copy Server". The configure servers dialog now has an additional button, "Copy...", which allows users to create a copy configuration of a particular server. This copy will allow 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 instance, 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—i.e., it will appear in the list of available servers on the server drop-down lists in the Project Properties dialog's Server page.

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

22

JBuilder Enterprise Setup dialog concepts

General description

JBuilder provides an OpenTools API for adding a `Setup` page to the Enterprise Setup dialog that comes up when the user chooses Tools | 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 menu is titled Enterprise Setup, it's available in both Enterprise and Professional versions, although only the Database Drivers Setup page is available from in the Professional 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`

Detailed description of feature/subsystem

Registering a Setup

To register a custom Setup, you must first extend the `Setup` abstract class, and define all of its abstract functions. You should 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 at the highest level tab list 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 tab underneath the existing Setup's tab. 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 tab 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.
- **`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.

Appendix

A

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.
- `Designer`
  Examples for writing Designers in JBuilder.
- `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.

- JOT
  Contains three samples that use JOT:
  - PackageTree
    Builds a hierarchical tree based on the package of a Java node selected in the project.
  - ReadingSource
    Reads a JOT source file.
  - WritingSource
    Writes a JOT source file.
- LayoutAssistant
  Sample Assistant for wiring a layout manager in the UI designer.
- 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 all the 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.
- PropertyEditor
  An example property editor that has special knowledge of the Design environment.
- UserBrief
  An example that shows how to make a Brief emulation.
- UserCUA
  An example that shows how to make an CUA emulation.
- UserEmacs
  An example that shows how to make an Emacs emulation.
- VCS
  Demonstrates how to implement a VCS backend that integrates into the new VCS framework of JBuilder 4.
- 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.

Also check the `samples/OpenToolsAPI` directory for the very latest additions to the OpenTools samples.

Index

F

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W