

# Writing OpenTools plugins to extend Borland® JBuilder®

---

A step-by-step example of writing  
a JSP™ tag wizard for JBuilder

A Borland White Paper

*By Keith Wood*

July 2004

---

**Borland®**

## Contents

Introduction.....	3
JBuilder OpenTools introduction .....	3
OpenTool basics.....	5
The JSP tag wizard.....	7
JSP custom tags.....	7
Wizard processing.....	8
The wizard UI.....	11
Code generation .....	12
The result .....	17
Conclusion.....	19
About the author .....	20

## Introduction

The complexity of the Java™ 2 platform is always increasing, with new APIs and specifications constantly being added. Keeping up with these changes is a major exercise, and most people can really gain expertise only in a selection of areas. To help overcome this knowledge shortfall, processes can be automated in order to provide the necessary underlying functionality, allowing you to focus on the parts specific to your business and solution.

Wizards are one way JBuilder helps you automate tasks. You are guided through a series of steps, gathering sufficient details to complete a task. When you are finished, the wizard implements its actions, often generating a file that is added to your project. The wizard captures an expert understanding of a specialized area, reducing the need for in-depth knowledge by the user. A consistent outcome is guaranteed because the wizard produces the same results every time for the same input.

This paper illustrates how easy it is to use the OpenToolsAPI to write a wizard plugin to extend JBuilder.

## JBuilder OpenTools introduction

JBuilder is Borland's award-winning Java development environment. It is an all-Java application written atop a generic IDE base (also in Java). This paper describes how you can extend and enhance the IDE itself. You can automate repetitive processes or capture complex steps behind a simple wizard interface. You can interact with the IDE to augment its capabilities and make it easier to work with. You can interface the IDE with third-party tools, such as version-control systems and application servers. And you can add innovative functionality that takes JBuilder in new directions. Each of these tasks is embodied in an *OpenTool*, one or more Java classes that you develop and install into JBuilder.

JBuilder is built upon a generic IDE base known as *PrimeTime*. This framework provides the basic services required of an IDE for any language, with JBuilder being the Java incarnation.

OpenTools have access to most areas within JBuilder and can extend or replace nearly any part of the UI. In fact, JBuilder itself is mostly a (large) collection of OpenTools.

The areas in which you can extend JBuilder's abilities fall into several main categories:

- the *core*, essential functionality on which all the other parts rely, including loading OpenTools, processing command-line arguments, interfacing with persistent storage through the Virtual File System, and differentiating resources within a project through node types and project hierarchies;
- the *user experience*, including persistent properties at all levels (global, by project, and by node), integration with the help system, and controls to allow your extensions to blend with the rest of the JBuilder UI;
- the *browser*, which is the main JBuilder window that hosts the main menu and toolbar, the Project Pane for organizing a project, the Content Pane for viewing and editing resources, the Structure Pane for navigating through content, the Message Pane for complex or long-lived messages, and the Status Pane for short or transient messages;
- the *node viewers* and *editors*, which provide you with different views on the contents of a node, and furnish syntax highlighting and pop-up assistance when editing text-based files;
- the *wizards* framework, allowing you to automate tedious or complex tasks, including the ability to parse existing Java source and class files, and then modify them, or generate entirely new ones;
- the *build* and *runtime* systems, enabling you to transform your source code and other resources into an executable application, and then run or debug that application within its appropriate environment, be it an applet, a J2EE application, a Web application, or a standalone program; and
- the *external systems* interfaces, to connect you to existing software such as version-control systems or application servers.

## OpenTool basics

The minimum requirements of an OpenTool are very basic: a class that has a method with the following signature:

```
public static void initOpenTool(byte majorVersion, byte
    minorVersion);
```

packaged in a JAR file with a manifest entry in the following format—a category name followed by a list of class names:

```
OpenTools-Wizard: wood.keith.opentools.wizards.jsptags.JSPTagWizard
    wood.keith.opentools.wizards.jsptags.TaglibDescriptorWizard
```

During startup, the PrimeTime framework scans a set of known directories (designated through a configuration file) for JAR files and extracts their manifests. It records all entries in the various “OpenTools-” categories and then commences to load them in a predetermined order. By delaying the loading of certain sets of OpenTools until they are actually needed, PrimeTime reduces the memory requirements and time required to start the application. For example, tools in the “Wizard” category are initialized only when the user opens the Object Gallery or the Wizards menu.

For each OpenTool identified in a manifest, PrimeTime calls its `initOpenTool` method when loading it, at which time the tool can perform whatever initialization it requires. Typically this step involves registering an instance of itself or of a supporting class with one of the many APIs within JBuilder. For example, a wizard registers a `WizardAction` with the `WizardManager` so that it appears in the Object Gallery or on the Wizards menu and can trigger the activities of the wizard itself.

```
/**
 * Register the "JSP Tag" tool.
 * Provides the needed OpenTools interface required to register the
 * WizardAction which defines and creates this wizard.
 *
 * @param majorVersion the major version of the current OpenTools
 * API
 * @param minorVersion the minor version of the current OpenTools
 * API
 */
public static void initOpenTool(byte majorVersion, byte minorVersion)
{
    if (majorVersion != PrimeTime.CURRENT_MAJOR_VERSION) {
        return;
    }
    WizardManager.registerWizardAction(WIZARD_JSPTag);
    if (PrimeTime.isVerbose()) {
        System.out.println("Loaded JSP Tag wizard v" + VERSION);
        System.out.println("Written by Keith Wood
(kbwood@iprimus.com.au)");
    }
}
```

As you can see, this method should first check that the OpenTool can run by examining the major and minor version numbers of the current JBuilder instance. If these are acceptable, the registration continues and a load message is displayed on the console if the user has added the `-verbose` option to the command line.

Generally, installing an OpenTool is as simple as copying its JAR file, which includes the code and the customized manifest, to one of the automatically scanned directories, typically `JBuilder/lib/ext`.

## The JSP™ tag wizard

One of the more arcane areas in the J2EE specifications is the use of custom tags within JavaServer Pages™ (JSP™). Custom tags let you add specialized functionality to your JSPs in a user-friendly manner. To use the tags, no knowledge of Java or programming is required. However, a reasonable knowledge of the JSP specification is necessary in order to write the Java classes that support the tags behind the scenes. Because much of this code is boilerplate, it is a great candidate for a wizard.

### JSP custom tags

Custom tags look like other JSP tags and can generate output (typically HTML), can control the presence or absence of their body content, and can cause their body to loop over a set of values. They may accept attributes that affect their activity and they may be nested within other custom tags for added abilities.

For example, the following JSP snippet might generate a questionnaire, stepping through a list of questions (`qn:questionList`) and displaying each one's prompt and appropriate input controls (`qn:question`). The tags are identified by a common prefix, "qn", which is associated with a tag library descriptor (TLD) file at the start of the page. This file then links the names of the tags appearing on the page to the Java classes that implement their functionality within the server.

```
<%@ taglib uri="/qn_tags.tld" prefix="qn" %>
<%@ include file="header.jsp" %>
<table class="asQuestionText" width="100%">
<qn:questionList>
  <tr>
    <td class="questionLabel"><qn:question field="label"/></td>
    <td class="questionInput">
      <qn:question field="input"/>
      <qn:question if="hasOtherText"><br />
        Other <qn:question field="otherText"/>
      </qn:question>
    </td>
  </tr>
</qn:questionList>
</table>
<%@ include file="footer.jsp" %>
```

The Java code to support these various abilities can become involved, especially when working with nested and looping tags. While the code for accepting attributes into the tag is simple, it is repetitive and tedious. If you also have to remember what functionality is or is not available in each of the JSP specifications published so far, then the task of writing a tag class becomes that much more difficult. Capturing all of this knowledge and tedious coding into a wizard relieves the burden of having to recall the exact syntax necessary to perform each task and ensures that a consistent base is available for the development of any new tag classes.

## Wizard processing

JBuilder provides a robust framework for developing wizards. As you have read, the process starts with the registration of a `WizardAction` with the `WizardManager`. The purpose of the action is to appear in the Object Gallery or on the Wizards menu and to invoke the main wizard class when selected. In general, wizards appearing in the Object Gallery create new files and add them to the project, and wizards included on the Wizards menu operate on existing files to modify them, or don't affect any files at all.

Each wizard must implement the `Wizard` interface, which is usually achieved by deriving it from the `BasicWizard` class. This class manages a wizard composed of a series of pages, which can be added or removed at any time during the processing. Each of the pages must implement the `WizardPage` interface, usually by descending from the `BasicWizardPage` class.

In your main wizard class, you override the `invokeWizard` method to create the pages that guide the user through your task and add them to the wizard. You then return a reference to the initial page to display (by default, it is the first one). The inherited functionality manages the progression through these pages in the order that they were added using the standard navigation buttons. The JSP Tag wizard has a single page, which is initialized as shown:

```
/**
 * Wizard startup.
 * "JSP Tag" wizard is to become visible,
 * create and order the wizard pages to be displayed.
 *
 * @param host the WizardHost that owns this wizard instance.
 * @return the initial WizardPage to show.
 */
public WizardPage invokeWizard(WizardHost host) {
    setWizardTitle(TITLE);
    _tagPage = new JSPTagWizardPage(
        host.getBrowser().getProjectView().getActiveProject());
    addWizardPage(_tagPage);
    return super.invokeWizard(host);
}
```

Each page is validated as it is completed, with any errors preventing movement to the next page. When the user completes his entry and presses the `Finish` button, the final page is validated and the `finish` method of the wizard is called. It is here that you perform the actual functionality of the wizard. You create a new Java source node and add it to the project, populate it through a code generator object based on the values entered by the user through the UI, and then open the new node in the IDE.

```
/**
 * Perform code generation.
 * Produces the "JSP Tag" file as defined by user input on the
 * wizard pages. It is added as a node to the currently active
 * project
 * and replaces any previously generated file of the same name.
 *
 * @throws VetoException if unable to finish
 */
protected void finish() throws VetoException {
    Browser browser = wizardHost.getBrowser();
    JBProject project =
        (JBProject)browser.getProjectView().getActiveProject();
    // Save selected JSP version for next time
    JSPTagProperties.JSP_VERSION.setValue(_tagPage.getJSPVersion());
    FileNode javaNode = null;
    try {
        : // Tag extra info generation
        // Create the tag source file
        javaNode = createNode(project, _tagPage.getPackageName(),
            _tagPage.getClassName() + ".java");
        new JSPTagGenerator().writeSource(project, javaNode,
            _tagPage.getPackageName(), _tagPage.getClassName(),
            _tagPage.getJSPVersion(), _tagPage.isBodyAltered(),
            _tagPage.isRepeated(), _tagPage.isEndUsed(),
            _tagPage.isNested(), _tagPage.isTryCatchFinally(),
            _tagPage.isSimpleTag(), _tagPage.isDynamicAttributes(),
            _tagPage.getProps());
    }
    catch (Exception ex) {
        ex.printStackTrace();
        JOptionPane.showMessageDialog(wizardHost.getBrowser(),
            res.getString("GenerationError") + "\n" +
            ex.getClass().getName()
            + "\n" + ex.getMessage(), wizardTitle,
            JOptionPane.ERROR_MESSAGE);
    }
}
```

```
    }
    try {
        // Open the generated/updated file in the Content Pane.
        browser.setActiveNode(javaNode, true);
    }
    catch (Exception ex) {
        // Ignore
    }
}
```

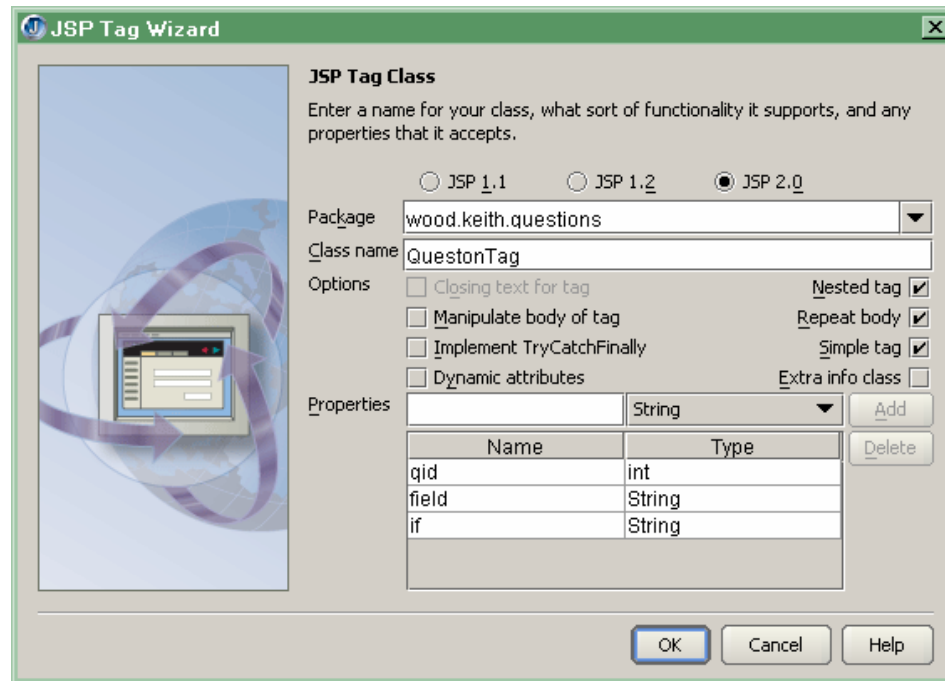
The wizard also can provide customized help when the user clicks the **Help** button by overriding the `help` method. You supply the help content as HTML files and associated images along with the code for the wizard.

## The wizard UI

The UI for a wizard usually is developed by descending from `BasicWizardPage`, which itself derives from `JPanel`. This base class provides the container for your controls while implementing appropriate functionality to integrate with the standard wizard framework.

Two main UI styles are encapsulated in this class: *regular*, which displays a large image down the left side, or *complex*, which shows only a small icon and therefore leaves more space for your own controls. You can use the designer abilities of JBuilder to construct your page. You also should override the `checkPage` method to validate the contents of the page and throw a `VetoException` if they are not acceptable.

**Figure 1** shows the completed page for the JSP Tag wizard. This simple interface masks the complexities of implementing all the functionality available to a custom JSP tag.



**Figure 1:** *The JSP tag wizard*

## Code generation

Although you could just string together the Java code required to generate the JSP tag class and write that directly to the file, JBuilder provides an alternative—the *Java Object Toolkit* (JOT). JOT is a set of interfaces and classes that allow you to parse, update, and generate Java code. A separate code generator class in this wizard takes care of the creation of the new tag class. After obtaining a reference to the new source file (`jsource`), the generator calls the `createClass` method (shown below) to add the JSP tag class to it via JOT. The parameters for the method come from the values entered by the user in the UI.

```
/**
 * Add the class declaration.
 *
 * @param jsource          the source file being generated
 * @param className       the name of the new class
```

```
* @param jspVersion          the version of JSP in use
* @param isEndUsed           true if content is written after the
    body,
*
*                               false if none
* @param isNested           true if the tag looks for an enclosing
    tag,
*
*                               false if standalone
* @param isRepeated         true if the tag iterates over its body,
*
*                               false if it processes it just once
* @param isBodyAltered      true if the tag changes the body
    content,
*
*                               false if it leaves it unchanged
* @param isTryCatchFinally  true if the tag deals with its own
    errors
*
*                               during processing, false if it does not
* @param isSimpleTag        true if tag is a JSP 2.0 simple tag,
*
*                               false if not
* @param isDynamicAttrs     true if tag accepts unspecified
    attributes,
*
*                               false if not
* @return the class being generated
*/
private JotClassSource createClass(JotSourceFile jsource,
    String className, String jspVersion, boolean isEndUsed,
    boolean isNested, boolean isRepeated, boolean isBodyAltered,
    boolean isTryCatchFinally, boolean isSimpleTag,
    boolean isDynamicAttrs) {
    JotClassSource jclass = jsource.addClass(null, AFTER, className,
        false);
    StringBuffer comment = new StringBuffer();
    comment.append(isEndUsed ? res.getString("TagIsEndUsed") : "").
        append(isNested ? res.getString("TagIsNested") :
            "").append(...);
    jsource.addComment(jclass, BEFORE, JotComment.DOC,
        MessageFormat.format(res.getString("JSPTagClassHeader"),
            new Object[] {jspVersion}) + (comment.length() == 0 ? "" :
```

```

        " " + res.getString("JSPTagClassHeaderCharacteristics") +
        comment.toString() +
        ".\n\n@author   JSP Tag Wizard\n@version 1.0 " +
        DateFormat.getDateInstance(DateFormat.LONG).format(new
        Date());
jclass.setModifiers(Modifier.PUBLIC);
jclass.setSuperclass(isSimpleTag ? "SimpleTagSupport" :
        (isBodyAltered || isRepeated ? "BodyTagSupport" :
        "TagSupport"));
if (isDynamicAttrs) {
    jclass.addInterface(null, AFTER, "DynamicAttributes");
}
if (isTryCatchFinally) {
    jclass.addInterface(null, AFTER, "TryCatchFinally");
}
return jclass;
}

```

You can see that the class itself is created first, and a Javadoc comment is added before it. The class' details are then set, including its visibility, its parent class, and any interfaces that it implements. A reference to the new class is returned from this method for use in adding members to it in subsequent methods.

Depending on the options selected by the user, many fields and methods are added to the class. Shown below is the code that generates the `doTag` method of a JSP 2.0 "simple" tag. As before, the necessary user parameters are passed in to control the process.

```

/**
 * Add SimpleTag method doTag override.
 *
 * @param jclass          the class being generated
 * @param isNested       true if the tag looks for an enclosing tag,
 *                       false if standalone
 * @param isRepeated     true if the tag iterates over its body,
 *                       false if it processes it just once (if at
 *                       all)

```

```

* @param isBodyAltered true if the tag changes the body content,
*                       false if it leaves it unchanged
* @param hasProperties true if the tag has properties, false
*                       otherwise
*/
private void addDoTag(JotClassSource jclass, boolean isNested,
    boolean isRepeated, boolean isBodyAltered, boolean hasProperties)
{
    JotMethodSource jmethod = jclass.addMethod(null, AFTER,
        "void", "doTag");
    JotComment jcomment = jclass.addComment(jmethod, BEFORE,
        JotComment.DOC,
        res.getString("DoTagMethod") + ...;
    jclass.addBlankLine(jcomment, BEFORE);
    jmethod.setModifiers(Modifier.PUBLIC);
    jmethod.addThrowSpecifier(null, AFTER, "IOException");
    jmethod.addThrowSpecifier(null, AFTER, "JspException");
    JotCodeBlock jcode = jmethod.getCodeBlock();
    if (isNested) {
        jcode.addComment(null, AFTER, JotComment.LINE,
            res.getString("DoTagFindEnclosing"));
        jcode.addComment(null, AFTER, JotComment.DOC,
            " @todo " +
            MessageFormat.format(res.getString("OuterTagTodo"),
                OUTER_TAG) + " ");
        jcode.addAssignment(null, AFTER, "_enclosingTag",
            "(OuterTag)findAncestorWithClass(this, OuterTag.class)");
        JotIf jif = jcode.addIfStatement(null, AFTER,
            "_enclosingTag == null");
        jif.getThen().getCodeBlock().addStatement(null, AFTER,
            "throw new JspException(\"" + MessageFormat.format(
                res.getString("DoTagThrowNestingError"), OUTER_TAG) + "\")");
    }
    JotVariableDeclaration jvar =
        jcode.addVariableDeclaration(null, AFTER, "JspWriter", "out");
    jvar.setInitializer("getJspContext().getOut()");

```

---

```
if (isRepeated || isBodyAltered) {
    jcode.addComment(null, AFTER, JotComment.DOC,
        " @todo " + res.getString("TagOpeningOutputTodo") + " ");
    jcode.addStatement(null, AFTER, "out.print(\"" +
        res.getString("TagOpeningOutputValue") + "\\");");
    jvar = jcode.addVariableDeclaration(null, AFTER,
        "JspFragment", "body");
    jvar.setInitializer("getJspBody()");
    if (isRepeated) {
        jcode.addComment(null, AFTER, JotComment.DOC,
            " @todo " + res.getString("RepeatInitTodo") + " ");
        jcode.addComment(null, AFTER, JotComment.DOC,
            " @todo " + res.getString("RepeatTestTodo") + " ");
        JotWhile jwhile = jcode.addWhileStatement(null, AFTER,
            "repeatBody");
        jcode = jwhile.getCodeBlock();
    }
    jcode.addComment(null, AFTER, JotComment.DOC,
        " @todo " + res.getString("DoTagAccessBodyTodo") + " ");
    jcode.addMethodCall(null, AFTER, "body.invoke", "out");
    jcode = jmethod.getCodeBlock();
    jcode.addComment(null, AFTER, JotComment.DOC,
        " @todo " + res.getString("TagClosingOutputTodo") + " ");
    jcode.addStatement(null, AFTER, "out.print(\"" +
        res.getString("TagClosingOutputValue") + "\\");");
}
else {
    jcode.addComment(null, AFTER, JotComment.DOC,
        " @todo " + res.getString("TagOutputTodo") + " ");
    jcode.addStatement(null, AFTER, "out.print(\"" +
        res.getString("TagOutputValue") + "\\");");
}
}
```

First, the new method is created within the class through JOT. A Javadoc comment is added for documentation and instructional purposes. The method's accessibility and exceptions are then specified. Accessing the code block for the method, you generate the individual statements that make up its body. Note how these are driven by the options selected by the user. Different types of statements are shown: assignments, an if test, a while loop, and variable declarations. You can also add "todo" comments to direct the user to areas that should be customized in their implementation.

## The result

The result of running the wizard as shown in **Figure 1** is the new code below. You can see that the generated text corresponds to the JOT code above and the options selected by the user. Now that you have the skeleton of a custom JSP tag that supplies the basic functionality that you requested, you can concentrate on the task of adding in the business logic for your particular need. The "todo" comments help focus your attention and direct your efforts.

```
package wood.keith.questions;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

/**
 * A JSP 2.0 tag with the following characteristics
 * - it refers to a surrounding tag
 * - it repeats the body content
 * - it has a simple lifecycle.
 *
 * @author   JSP Tag Wizard
 * @version  1.0  23 March 2004
 */

public class QuestionTag extends SimpleTagSupport {

    // Internal fields
```

```
/** @todo set default values */
private String _field;
private String _if;
private int _qid;
/** @todo replace OuterTag with the actual class name */
private OuterTag _enclosingTag = null;

/**
 * Save the value of field
 *
 * @param value the attribute value
 */

public void setField(String value) {
    _field = value;
}

: // Other field setters

/**
 * Process the tag for this instance.
 * When this method is invoked, the body has not yet been evaluated.
 * The doTag() method assumes that all property setter
 * methods have been invoked before this call.
 *
 * @throws IOException if an error occurs while writing
 * to the output stream
 * @throws JSPException if an error arises
 * @throws SkipPageException if the calling page should stop
 * executing
 */

public void doTag() throws IOException, JspException {
    //Find enclosing tag for a nested tag
    /** @todo replace OuterTag with the actual class name */
```

```
    _enclosingTag = (OuterTag)findAncestorWithClass(this,
OuterTag.class);
    if (_enclosingTag == null) {
        throw new JspException("This tag must appear within OuterTag");
    }
    JspWriter out = getJspContext().getOut();
    /** @todo specify any opening output value for this tag */
    out.print("opening tag value");
    JspFragment body = getJspBody();
    /** @todo perform initialisation for the repeating body */
    /** @todo determine the test for the repetitions */
    while (repeatBody) {
        /** @todo use body to access the contained JSP */
        body.invoke(out);
    }
    /** @todo specify any closing output value for this tag */
    out.print("closing tag value");
}
}
```

## Conclusion

With new APIs and functionality being added to Java all the time, becoming an expert in all areas is challenging. By capturing the domain knowledge in a JBuilder wizard, you can ensure that nothing is overlooked and that a consistent source base is produced for every implementation. Creating your own wizard and adding it to JBuilder is a relatively simple task, especially given the support of the OpenTools APIs and the JBuilder development environment. The example shown here generates classes to implement JSP custom tags. It is available for download from CodeCentral (<http://codecentral.borland.com/codecentral/ccWeb.exe/listing?id=21573>) and includes the full source code and JBuilder project. A second wizard in the same package scans a project for custom JSP tag classes and assists you in producing a tag library descriptor file that defines them—a further example of automating a complex task within a wizard.

## About the author

**Keith Wood** hails from Brisbane, Australia, where he is product architect with CiTR Asia Pacific. He has used JBuilder since its first release and has contributed many OpenTools to the JBuilder community. His book, *Inside the JBuilder OpenTools API*, BookSurge, 2004, provides much more detail on developing OpenTools. He also is author of the *Delphi Developer's Guide to XML*, BookSurge, 2003.

**Made in Borland®** Copyright © 2004 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. Microsoft, Windows, and other Microsoft product names are trademarks or registered trademarks of Microsoft Corporation in the U.S. and other countries. All other marks are the property of their respective owners. Corporate Headquarters: 100 Enterprise Way, Scotts Valley, CA 95066-3249 • 831-431-1000 • www.borland.com • Offices in: Australia, Brazil, Canada, China, Czech Republic, Finland, France, Germany, Hong Kong, Hungary, India, Ireland, Italy, Japan, Korea, Mexico, the Netherlands, New Zealand, Russia, Singapore, Spain, Sweden, Taiwan, the United Kingdom, and the United States. 22209