

Writing Java Proxies for Rational Robot

by [Dr. Akin Akinrogunde](#)

Senior Technical Support Engineer
Rational Software BV, The Netherlands

Rational Robot® is an automated tool that records user interactions with an Application Under Test (AUT) in functional, performance, and reliability tests. It is also used in functional and reliability tests to play back recorded scripts. This article focuses on using Rational Robot for functional testing of Java applications and applets.

Currently, Rational Robot explicitly supports applications and applets developed with the following class libraries:

- *Java Foundation Classes (JFCs)*
- *Swing and Abstract Windowing Toolkit (AWT) from Sun Microsystems*
- *Visual Café from Symantec*
- *JClass Library from Sitraka Software*

There are other interesting Java class libraries out there that are not explicitly supported by Rational Robot, however, some of which are based on explicitly supported class libraries. Testers can extend the capability of Rational Robot to support these classes by writing Java proxies, using the Java Extensibility API that is shipped with Rational Robot. These proxies can be deployed only if the application or applet being tested runs on the Java Virtual Machine (JVM) supplied by Sun Microsystems, and on the JVMs shipped with Netscape Communicator and Microsoft Internet Explorer.

This article discusses how Rational Robot uses Java proxies and when third-party Java proxies are required; it also demonstrates how Java proxies are developed, using the Java Extensibility API supplied by Rational Software.

How Rational Robot Uses Java Proxies

To perform its functional, performance, or reliability tests, Rational Robot spies on user interactions with the AUT and generates Rational's SQABasic script, regardless of the language in which the application is developed. When the recorded script is played back, it reproduces the user's interaction with the AUT. For testing Java applications and applets, SQABasic has a set of well-defined Graphical User Interface (GUI) objects. Table 1 shows the set of SQABasic GUI objects for testing Java applications and



- ▶ [subscribe](#)
- ▶ [contact us](#)
- ▶ [submit an article](#)
- ▶ [rational.com](#)
- ▶ [issue contents](#)
- ▶ [archives](#)
- ▶ [mission statement](#)
- ▶ [editorial staff](#)

applets.

Table 1. SQABasic Objects for Testing Java Applications and Applets

JavaMenu	JavaMenuItem	CheckBox	RadioButton
ComboBox	ComboListBox	Label	ListBox
ScrollBar	EditBox	TrackBar	TabControl
ProgressBar	JavaPanel	JavaWindow	JavaTree
JavaSplitPanel	JavaSplitter	JavaMenuBar	JavaObject
JavaCheckBoxMenuItem	PushButton	ToolBar	JavaListView
JavaMenuSeparator	JavaTable	JavaCanvas	
JavaPopupMenu	JavaTableHeader		

These SQABasic GUI objects are characterized by their properties and functionalities. The functionalities define permissible user interactions that Robot can simulate for the objects.

Rational Robot natively knows how to manipulate SQABasic GUI objects only. To test Java applications and applets, Java GUI components have to be mapped to the SQABasic GUI objects. For example, the `JButton` component in JFC is mapped to SQABasic object `PushButton`.

Through introspection (also called reflection), Rational Robot can dynamically determine the class, functionalities, and properties of a Java GUI component loaded into the JVM. It uses the information contained in `JavaClassMap.dat` to determine which SQABasic object the class maps to, and the proxy it can use to interact with the Java component.

The template for `JavaClassMap.dat` is supplied by Rational Software. It is located in

```
<ProjectRootDir>\TestDataStore\DefaultTestScriptDataStore\TMS_scripts\dat
```

in the project directory tree. The information in `JavaClassMap.dat` is of the form:

```
[SQABasic Object]  
Java GUI component=proxy
```

Figure 1 depicts this process, showing how Rational Robot uses proxies for interacting with Java applications. For Java proxy classes supplied by Rational, the information contained in `JavaClassMap.dat` is made available to Robot internally.

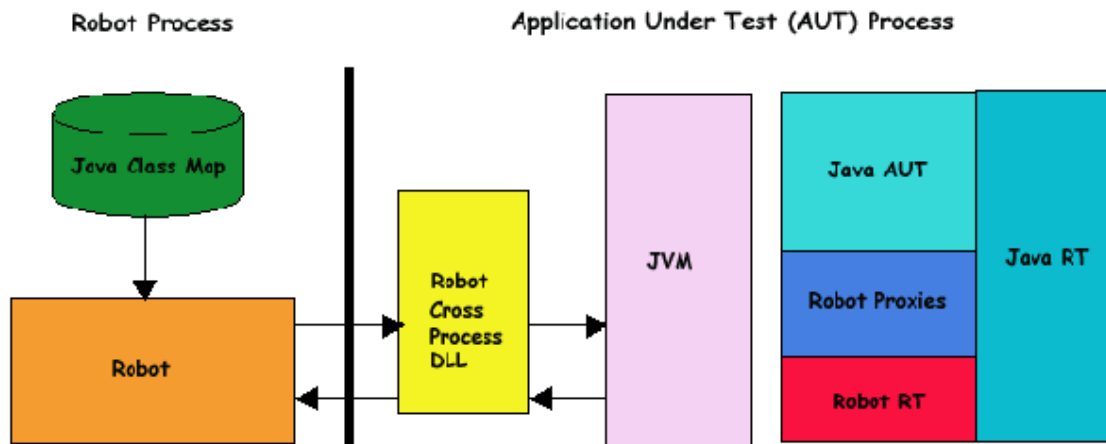


Figure 1: Rational Robot's Test Environment

The process works like this:

1. During recording, Rational Robot spies on the interaction between the user and the AUT; through introspection it knows the class of the component the user is interacting with.
2. Rational Robot uses the `JavaClassMap.dat` file, or similar information made available internally, to determine the `SQABasic` GUI object the class maps to, and the proxy for interacting with the Java component.
3. Rational Robot then uses the proxy to acquire necessary information about the Java component and generate the appropriate `SQABasic` command.
4. To play back the recorded script, the `SQABasic` command is translated to Robot calling the appropriate function or sequence of functions in the proxy class, which in turn invoke(s) the appropriate function(s) in the component being tested.

For example, to simulate a user click action on a Java GUI component -- `JButton`, for example -- Rational Robot issues the following `SQABasic` command:

```
PushButton Click, recMethod
```

where `recMethod` is a string that uniquely identifies the pushbutton within the AUT. The same command is used for all pushbuttons, regardless of the language in which they are developed.

When Is a Third-Party Java Proxy Required?



If an application is developed with any of the explicitly supported class libraries using standard Java GUI components, then proxies supplied by Rational are sufficient for Robot's interaction with the application. Standard Java GUI components, for the purposes of this article, are those components delivered in a class. `JButton`, `Jtree`, and `JPanel` are examples of standard Java GUI components in JFCs.

Applications developed with explicitly supported class libraries that have customized standard Java GUI components also do not require third-party Java proxies *if* the customized class and the super class map to the same `SQABasic` object. For example, if an application uses the class `BeechButton`, which extends `JButton` (a standard Java

GUI component), then Rational-supplied proxies are sufficient for Robot to interact with the component, because both `BeechButton` and `JButton` map to the same SQABasic object, `PushButton`.

If a standard component of a supported class library is customized to such an extent that the super class and the customized class map to different SQABasic GUI objects, then a third-party proxy may or may not be required, depending on the implementation of the custom class.

For example, let's suppose that `JPanel` (in the JFC), which maps to the SQABasic object `JavaPanel`, is customized to a component, `TreeComp`, which maps to the SQABasic object `JavaTree`.

- Without a third-party proxy, by default, Rational Robot will interact with `TreeComp` as if it were a `JavaPanel` object. In this situation, the functionalities of the `JavaTree` cannot be tested.
- If `TreeComp` is mapped to `JavaTree` in the `JavaClassMap.dat`, then Rational Robot will use the proxy for `Jtree` to interact with `TreeComp`. `TreeComp` will be recognized as `JavaTree`, and a third-party proxy is not required if `TreeComp` implements all the functions in `Jtree`.
- If `TreeComp` implements functions similar to those in `Jtree` but with different names, then Robot can still interact with `TreeComp`, but the interaction will be rudimentary. This means selection of items within the tree will be recorded as clicks on coordinates, and the Object Data Verification Point will not work, although the interaction with `TreeComp` may still play back successfully.

If this level of interaction is sufficient for the testing program, then a third-party proxy is not required. Keep in mind, however, that a third-party proxy would be required if you want the Object Data Verification Point and items selection functionality to work.

And note that third-party proxies *are always* required to test applications and applets developed with class libraries that are not explicitly supported.

Examples: When Is a Java Proxy Required for Java GUI Component Testing?

To demonstrate how to determine whether or not a Java proxy is required for testing a particular Java GUI component, I will use three Java GUI components from the JBCL class library (one of the libraries shipped with JBuilder 4 from Borland). The Java GUI components are:

1. `com.borland.jbcl.control.TextFieldControl`
2. `com.borland.jbcl.control.ButtonControl`
3. `com.Borland.jbcl.control.TreeControl`

We will discuss each one below.

`com.borland.jbcl.control.TextFieldControl`

The class hierarchy for `com.Borland.jbcl.control.TextFieldControl` is:

```
java.lang.Object
+-java.awt.Component
  +-java.awt.TextComponent
    +-java.awt.TextField
      +- com.Borland.jbcl.view.TextFieldView
        +-com.borland.jbcl.control.TextFieldControl
```

java.awt.TextField is an ancestor of com.Borland.jbcl.control.TextFieldControl. java.awt.TextField is from AWT, a class library explicitly supported by Rational Robot. Both java.awt.TextField and com.Borland.jbcl.control.TextFieldControl map to the SQABasic object `EditText`. Therefore, Robot can fully interact with this component using the Rational-supplied proxy, `AwtTextFieldProxy`.

com.borland.jbcl.control.ButtonControl

The class hierarchy for com.Borland.jbcl.control.ButtonControl is:

```
java.lang.Object
+-java.awt.Component
  +-java.awt.Container
    +-javax.swing.JComponent
      +- com.Borland.jbcl.view.BeanPanel
        +- com.Borland.jbcl.view.ButtonView
          +-com.Borland.jbcl.control.ButtonControl
```

Again, it has as one of its ancestors `javax.swing.JComponent` from a supported class library JFC. `JComponent` maps to `JavaObject` in SQABasic, which has only rudimentary functionalities common to all Java components.

com.borland.jbcl.control.ButtonControl maps to `PushButton` in SQABasic. By default, Rational Robot will recognize com.Borland.jbcl.conrol.ButtonControl as `JavaObject`. For a click action on com.Borland.jbcl.control.ButtonControl, Rational Robot will generate the following SQABasic command:

```
JavaObject Click, RecMethod, Coord
```

where `RecMethod` is a string that uniquely identifies the `ButtonControl` and `Coord`, the coordinate where the click action occurred. The object properties can be retrieved, and the recorded script will play back without a problem if the dimensions of `ButtonControl` do not change to such an extent that the coordinate of the click action falls outside of `ButtonControl`.

To record a coordinate-independent script, `ButtonControl` can be mapped to the SQABasic object `PushButton`, using the Rational-supplied proxy for `JButton`:

`JfcButtonProxy`. In this case, `JfcButtonProxy` can interact completely with `ButtonControl` because `JfcButtonProxy`, which implements `IRtButton` (the interface for pushbuttons), uses only functionalities in `JComponent`, which is a common ancestor for both `JButton` and `ButtonControl`.

com.borland.jbcl.control.TreeControl

com.Borland.jbcl.control.TreeControl is the last example. Its class hierarchy is shown below.

```
Java.lang.Object
+-Java.awt.Component
+-Java.awt.Container
+-Javax.swing.JComponent
+-Javax.swing.JScrollPane
+-- com.Borland.jbcl.view.TreeView
+--com.Borland.jbcl.control.TreeControl
```

The closest ancestor to `TreeControl` from a supported class is `JScrollPane`. `JScrollPane` maps to the SQABasic object `JavaObject`, whereas `TreeControl` maps to `JavaTree`.

By default, Rational Robot will interact with the `TreeControl` object as `JavaObject`; as mentioned earlier, `JavaObject` represents the functionalities common to all Java components. With `TreeControl`, it does not work to use the proxy Rational Robot supplies for `JTree` (i.e., `JfcTreeProxy`). `TreeControl` does not implement exactly the same functions in `JTree`, so it requires a third-party proxy.

JBCLControls Sample Application

To further illustrate this discussion, we will use a simple application, `JBCLControls`, which includes the three Java GUI components we looked at in the previous section. See Appendix A for the `JBCLControls` source code.

To compile this application you will need:

- The Borland class libraries (shipped with JBuilder 4 Enterprise edition) for the `jbcl` components, `jbcl.jar` and `dx.jar`
- SDK version 1.3 from Sun Microsystems

1. Copy `dx.jar` and `jbcl.jar` to a directory (e.g., `d:\jbcllib`).

2. Copy `JBCLControls.java` to directory `d:\jbclcon`.

3. Add the following string to the user's classpath environment variable:

```
D:\jbcllib\jbcl.jar;D:\jbcllib\dx.jar;D:\jbclcon
```

4. Change to the directory `d:\jbclcon`, and issue the command below to compile:

```
javac JBCLControls.java
```

5. Run the application through Rational Robot, as shown in Figure 2.

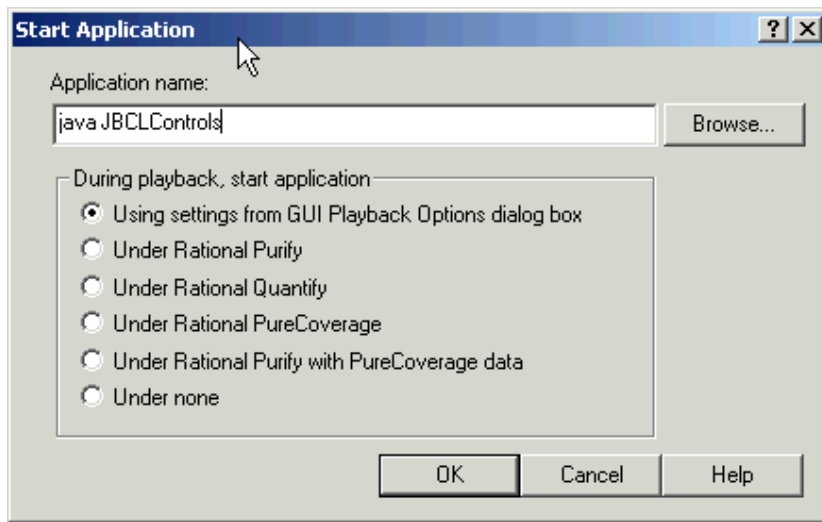


Figure 2: Running the JBCLControls.java Application Through Rational Robot

Alternatively, you can run the application as an applet, using the HTML code in Appendix B. The swing class library from Sun Microsystems must be installed for the applet to run.

Now, go ahead and test the application using Rational Robot, and then examine the SQABasic Script generated by Robot in light of the discussion about controls above.

How to Write a Java Proxy

To develop Java proxies for Rational Robot, you must be familiar with introspection in Java, the Java Extensibility API, and the class library used to develop the application or applet under test.

To demonstrate how to write a proxy, let's develop one for `com.Borland.jbcl.control.TreeControl`, which we discussed above.



A Java proxy for Rational Robot is simply a Java class that implements the interface specified for the SQABasic object to be tested. As mentioned earlier, `TreeControl` maps to the SQABasic object `JavaTree`. `IRbtTree` is the interface for `JavaTree`. `IRbtTree` extends `IRbtBase`. This means the proxy for `TreeControl` must implement all the functions specified in `IRbtTree` (including those specified in `IRbtBase`).

In reality, some of the proxies supplied by Rational usually serve as base classes for proxy development. `TreeControl` has both `javax.swing.JComponent` and `java.awt.Component` as its ancestors. Either `AwtBaseProxy` or `JfcBaseProxy` (both supplied by Rational) can serve as the base class for the proxy development. For this article, the `AwtBaseProxy`, which implements `IRbtBase`, is selected to serve as the base class. What now remains to be done is to implement the remaining functions specified in `IRbtTree`.

The implementation of the proxy class begins with specifying the package the proxy belongs to, and importing packages required for writing the proxy:

```
package beech;

import Java.lang.Object;
import Java.awt.Point;
import rational.robot.awt.*;
import rational.robot.openapi.*;
```

Next comes the body of the class:

```
public class JBCLTreeProxy extends AwtBaseProxy implements IRbtTree
{
public JBCLTreeProxy(){}
.
.
}
```

If a constructor is provided, then the public keyword must be used. Otherwise, Robot will not find the proxy and will respond with the following error message whenever it attempts to load the proxy.

```
Java proxy is unavailable: Proxy [<proxyName> requested for class
<className>; using default proxy.
```

Complete source code for the proxy is attached as Appendix C. A look at the source code shows that the proxy simply invokes functions in the component being tested. For example, the interface function `getNodeImage` is used to retrieve the text at each node of the `TreeControl` object. Robot passes the object at the node of interest to the `getNodeImage`, and the function, in turn, invokes the function `get` in `TreeControl`. The function `get` returns the string at the node to `getNodeImage`, which passes it to Robot. The implementation of `getNodeImage` is shown below.

```
public String getNodeImage(Object node)
{
    // parameter type being passed to function get in TreeControl
    Class locClass[] = new Class[1];
    locClass[0] =
        RbtReflection.loadClass("com.borland.jbcl.model.GraphLocation");

    // value of the parameter being passed to get in TreeControl
    Object fParam[] = new Object[1];
    fParam[0] = node;

    if(node==null)
        return " ";

    //invoke function get in TreeControl and
    //return String get returns.
    return
        (String)RbtReflection.invokeMethod("get",
            theObject,
            fParam,
            locClass );
}
```

Compiling Source Code

To compile the source code, assuming the environment variable classpath has been modified as described in the last section, follow the procedure below.

1. Run the Java Enabler (see Figure 3):

```
Start->Program->Rational SuiteTestStudio->Rational Test-> Java Enabler
```

This process essentially prepares the communication link between Rational Robot and the Java processes.

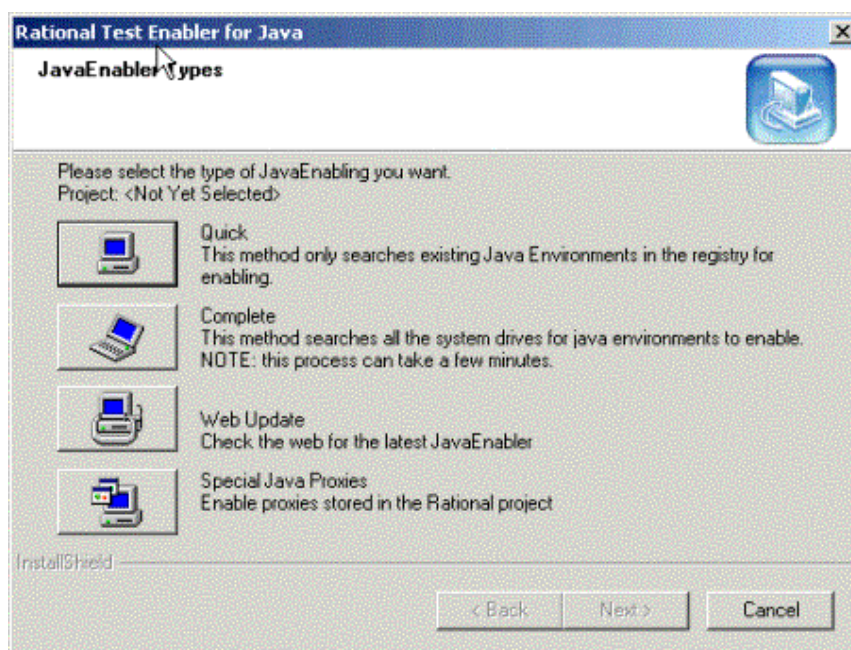


Figure 3: Running the Rational Test Enabler for Java

Click **Complete** for the installation process to search for all installed Java Environments. A machine usually has more than one Java Environment.

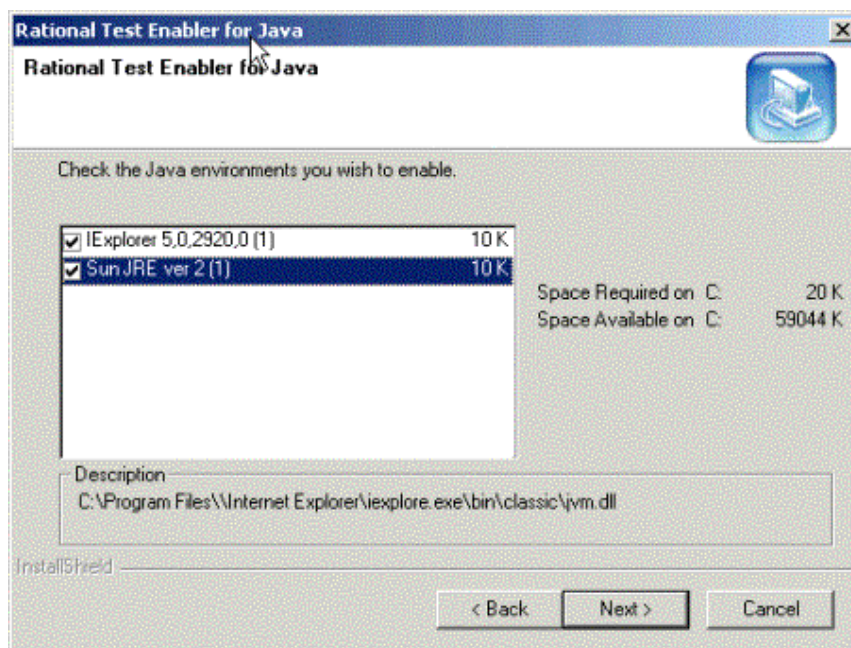


Figure 4: Selecting Java Environments for Testing Runs

From the list of Java Environments displayed, select those you want the AUT to run on (see Figure 4). All the environments you select will be enabled for testing Java applications and applets. Only Support JVM should be selected.

Click **Next** to complete the installation process, which will attach the `sqarobot.jarfile` to the `classpathenvironment` variable.

2. Issue the command:
`javac JBCLTreeProxy.java -d`

This creates the beech directory in the current directory and places the JBCLTreeProxy.class there.

3. Issue the command:

```
jar fcv beechProxy.jar beech
```

This compresses the JBCLTreeProxy.class to beechProxy.jar.

Deploying a Java Proxy



Once the class file has been compressed to jar format, the proxy is ready to be deployed. The procedure is as follows.

1. From the Rational Robot menu, select **Tools-> Extension Manager**

Ensure that the Java extension is checked (see Figure 5).

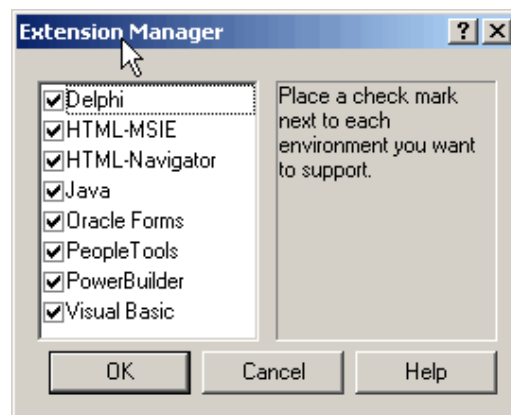


Figure 5: Extension Manager Environment List

2. Edit the JavaClassMap.dat directly or through the GUI by selecting:

Tools -> General Options from Robot, and then clicking on the Java Class Mapping tab. In the Java object type box, select JavaTree, as shown in Figure 6.

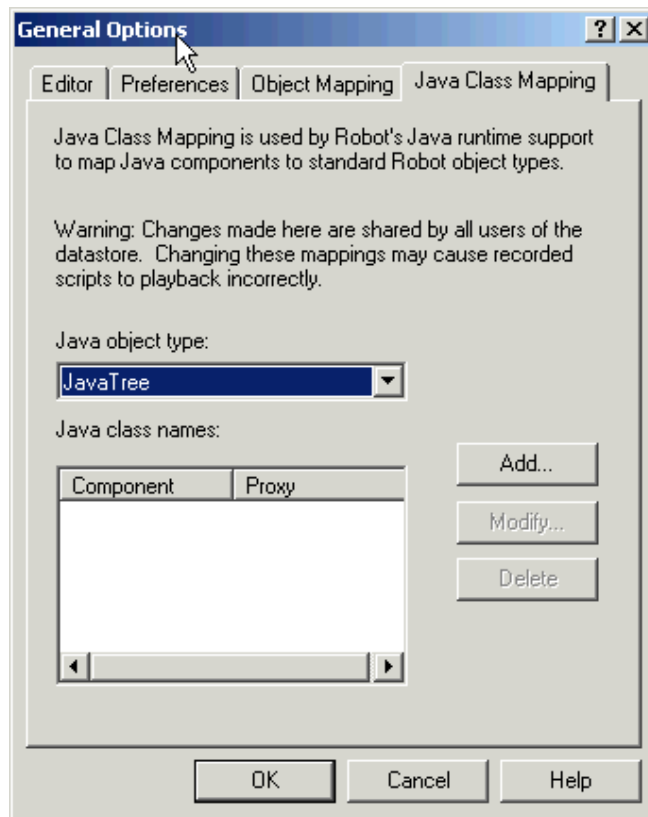


Figure 6: Java Class Mapping Dialog

Click **Add** to enter the `TreeControl` class and its proxy (see Figure 7).

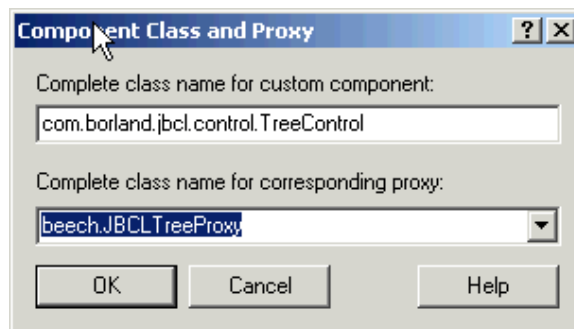


Figure 7: Component Class and Proxy Dialog

Click **OK**.

The `JavaTree` section in the `JavaClassMap.dat` should now look like this:

```
[JavaTree]
com.Borland.jbcl.control.TreeControl=beech.JBCLTreeProxy
```

3. Copy the file proxy file `beechProxy.jar` and the Borland files `jbcl.jar` and `dx.jar` to:

```
<ProjectRootDir>\TestDataStore\DefaultTestScriptDataStore\ TMS_scripts\JavaProxies
```

4. Run the Java Enabler:

```
Start->Program->Rational SuiteTestStudio->Rational Test-> Java Enabler
```

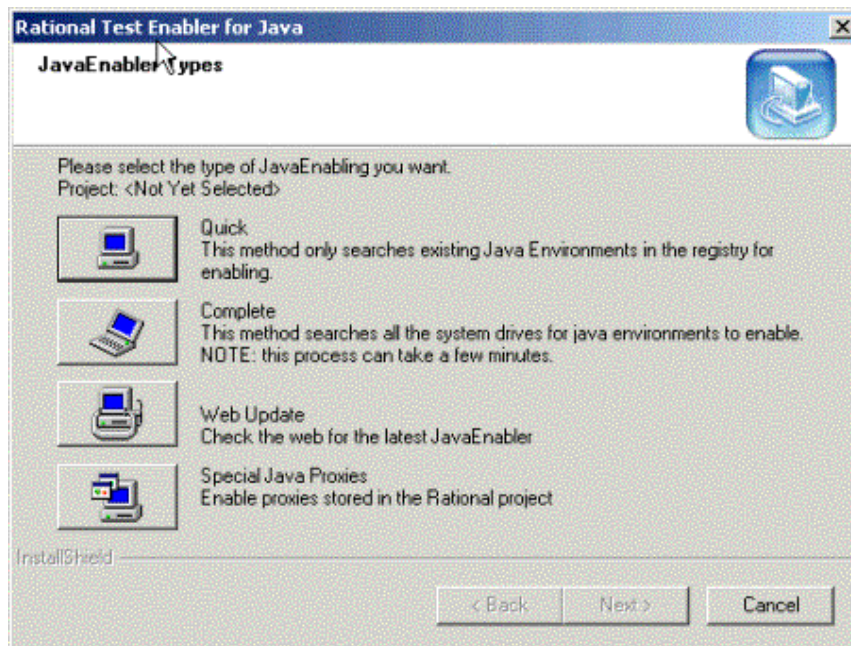


Figure 8: Rational Test Enabler for Java

Click the **Special Java Proxies** button (Figure 8) to select the project in which the proxy file is located. Click either on **Complete** or **Quick** for the installation process to search for all installed Java Environments and enable them.

5. Test the proxy using the JBCLControls application.

Conclusion

With Java proxies, Rational Robot can support virtually all Java class libraries, as long as the applications developed with them run on the supported JVM. We have seen how Rational Robot uses Java proxies to extend its capability, when a third-party proxy is required, and how to write a Java proxy. You can now use the application in Appendix A (or the HTML page in Appendix B to run it as an applet) to test the Java proxy you develop.

Appendices

[Appendix A](#): Source Code for the Application JBCLControls

[Appendix B](#): HTML Page for Running JBCLControls as an Applet

[Appendix C](#): Source Code for JBCLTreeProxy

Resources

1. <http://www.borland.com/techpubs/jbuilder/jbuilder4/ref/jbcl/Package-com.borland.jbcl.control.html> (Borland JBCLControls documentation).

2. <http://java.sun.com/products/jdk/1.1/docs/guide/reflection> (information on Java

Reflection).

3. <InstallationDirectoryForRationalProducts>\Rational
Test\JavaEnabler\api\index.html (Rational's Java Extensibility API online
documentation).



***For more information on the products or services discussed in this article,
please click [here](#) and follow the instructions provided. Thank you!***

Copyright [Rational Software](#) 2002 | [Privacy/Legal Information](#)