[Skip to content](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html" \l "container)

* [*Safari*](https://www.safaribooksonline.com/home/)
* [Recommended](https://www.safaribooksonline.com/r/)
* [Queue](https://www.safaribooksonline.com/s/)
* [Search](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html)
* [Expand Nav](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html)
  + [Recent](https://www.safaribooksonline.com/recent/)
  + [Topics](https://www.safaribooksonline.com/t/)
  + [Highlights](https://www.safaribooksonline.com/u/p253751/)
  + [Settings](https://www.safaribooksonline.com/u/)
  + [Feedback](mailto:feedback@safaribooksonline.com)
  + [Sign Out](https://www.safaribooksonline.com/accounts/logout/)
  + [Settings](https://www.safaribooksonline.com/u/)**10** days left in your trial. [Subscribe](https://www.safaribooksonline.com/subscribe/).
  + [Feedback](mailto:feedback@safaribooksonline.com)
  + [Sign Out](https://www.safaribooksonline.com/accounts/logout/)

[Software Test Engineering with IBM Rational Functional Tester: The Definitive Resource](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html)

* [Search in book...](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html)

Top of Form

Bottom of Form

* [Toggle Font Controls](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html)
* [Share this](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html)
  + [Twitter](https://twitter.com/share?url=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html&text=Software%20Test%20Engineering%20with%20IBM%20Rational%20Functional%20Tester%3A%20The%20Definitive%20Resource&via=safari)
  + [Facebook](https://www.facebook.com/sharer/sharer.php?u=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html)
  + [Google Plus](https://plus.google.com/share?url=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html)
  + [Email](mailto:?subject=Safari:%20Chapter%204.%20XML%20and%20Rational%20Functional%20Tester&body=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html%0D%0Afrom%20Software%20Test%20Engineering%20with%20IBM%20Rational%20Functional%20Tester%3A%20The%20Definitive%20Resource%0D%0A)

[Prev](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch12.html)

[Chapter 12. Extending Rational Functional Tester with External Libraries](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch12.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch14.html)

[Chapter 14. Developing Scripts in the VB.NET Environment](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch14.html)

**Chapter 13. Building Support for New Objects with the Proxy SDK**

**Daniel Gouveia**

*Automating custom controls usually ends up with one of two things happening: You either create a series of custom scripting methods or abandon the piece of your automation project that contains the custom controls. Rational Functional Tester actually provides a third option—building support for your control(s) directly into its record and playback capabilities. This provides a great benefit to you and the rest of your team. They do not have to work at the code-level and call the methods that you constructed to handle custom controls. Instead, they can continue engaging Rational Functional Tester’s record and playback engine, automating the application’s custom control(s) much like everything else they automated. This chapter, along with the documentation in Rational Functional Tester’s Help files, should provide you with a good starting point for building new capabilities for custom controls into Rational Functional Tester.*

*Creating record and playback support for custom control(s) is handled through the Proxy SDK. This chapter introduces you to it and covers topics at a higher and hopefully easier level to understand. It is broken down so that you can understand the information as it relates to the following three typical tasks you automate:*

• [*Verifying object properties*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13sec1lev4)

• [*Verifying object data*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13sec1lev5)

• [*Clicking via object information*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13sec1lev6)

*You also encounter some supporting tasks for creating a new proxy project, building it, “plugging” it into Rational Functional Tester, and debugging it. You do not need to read this chapter straight through. You are welcome to skip around to the different sections. However, you might want to acquaint yourself with the first two sections. They cover the content for understanding proxies and creating the proxy projects inside of Eclipse and .NET Studio.*

*The final thing to note is that proxies are developed using either Java or Visual C#®. Java is used to develop proxies for Abstract Windows Toolkit (AWT), Swing, Standard Widget Toolkit (SWT), and Applet controls. For .NET, Win32, Siebel, and SAP controls, you use C#. Rational Functional Tester’s help files provide the list of software requirements to carry out this task. Just look under the Proxy Development Environment topic.*

**Figuring Out Which Proxy to Extend**

The first step to creating a proxy is gaining an understanding of the Graphical User Interface (GUI) control that you actually try to create a proxy for. The control is either created from a base class or the base class itself. A *base class* is the unit that Rational Functional Tester understands. When you hear people talking about Rational Functional Tester supporting .NET, Java, and so on, it means that Rational Functional Tester understands and can interact with the base classes that represent the .NET and Java (and other) GUI controls. The base class is the key piece of information that you need to figure out. Acquiring knowledge of it can give you an understanding of the Rational Functional Tester proxy class that you need to extend. The out-of-the-box proxy classes give Rational Functional Tester its capability to work with the technologies it supports. For instance, clicking a .NET button, navigating a JTable, entering text into an HTML input field, and so on are performed with proxy classes that are part of Rational Functional Tester. Extending these standard proxies simply means that you build on top of their capabilities. An example of this might involve extending a proxy to add a new data test. The original out-of-the-box proxy works just fine; you just need to add a new way of testing data in your control.

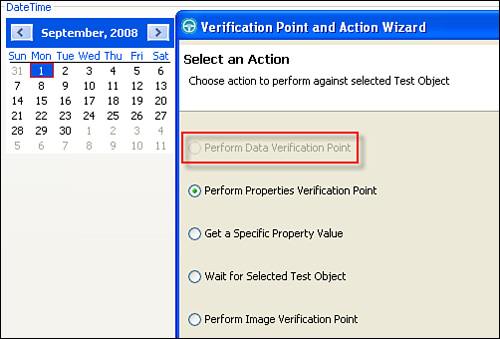
In most cases, Rational Functional Tester provides a shortcut to base class information via its Test Object Map. An example to illustrate this is the SWT DateTime control. It is part of the Eclipse SWT control-set (see [Figure 13.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig01)).

**Figure 13.1** DateTime control



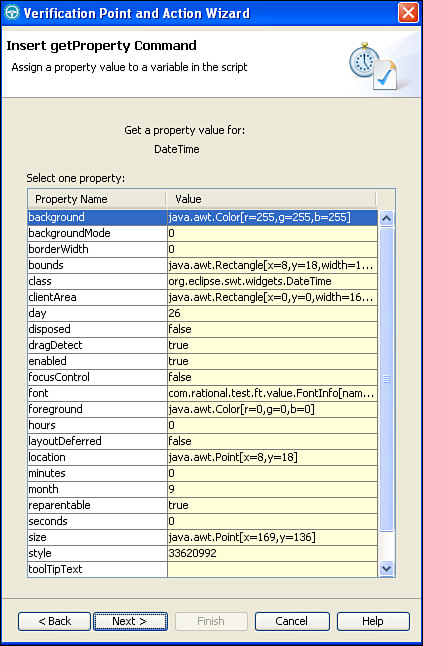
Rational Functional Tester is able to recognize this control. However, it is unable to capture data or execute a data verification point (VP) for it. Therefore, you seek to extend the out-of-the-box capabilities that Rational Functional Tester interacts with this DateTime by adding the ability to test its data (see [Figure 13.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig02)).

**Figure 13.2** Verification Point Wizard—no data verification point



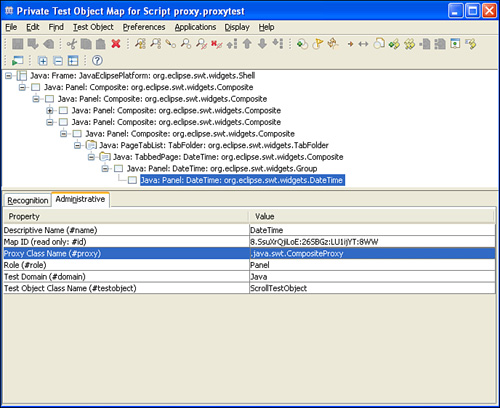
Further, when looking at the properties, you can see that a selected date type of property does not exist for this control. This is another area that you should extend, adding capability to the original proxy so Rational Functional Tester can test for a selected date (see [Figure 13.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig03)).

**Figure 13.3** Verification Point Wizard—no selected date type of property



You can create a custom proxy to add these capabilities to Rational Functional Tester’s Verification Point Wizard. As mentioned in this section’s opening paragraph, you first need to understand which proxy to extend for this control. You can acquire useful information by placing this control in Rational Functional Tester’s Test Object Map (for example, start recording and simply click on it). The Administrative properties, for this control, contain the Proxy Class Name (#proxy) property. This property lets you know what proxy Rational Functional Tester uses to work with the DateTime control (see [Figure 13.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig04)).

**Figure 13.4** Test Object Map—DateTime object’s administrative properties



You can see that Rational Functional Tester uses the CompositeProxy to work with it. It does not have an explicit proxy to interact with the DateTime control (that is no DateTimeProxy). If it did, you would have seen it in the Proxy Class Name (#proxy) property, instead of the CompositeProxy. You now know that the CompositeProxy is the base proxy you need to extend, adding the necessary verification point capabilities mentioned previously.

There are some cases where Rational Functional Tester won’t recognize the control at all. In other words, it isn’t able to add it to the map, capture properties, test for data, and so on. It is almost as if it is invisible to Rational Functional Tester. In these instances, you are not able to use the Test Object Map shortcut to understand which proxy to extend. This is where you need to figure out the base class for the control using other sources.

A good place to start figuring out which Rational Functional Tester proxy to extend is by looking for information about the target control class. Control developers, the control’s API documentation, and the actual source code itself can provide this information. Developers are usually able to provide you with the quickest answer. They have an intimate understanding of the source code with which they are working. A quick email, instant message, or phone call usually results in the information that you need—the base class.

If you don’t have access to the development staff, perusing the API documentation can be helpful. It can also be daunting. You can shorten your search by simply looking at the hierarchy information for the class that your GUI control instantiates. For instance, in the Java world, you can use the Javadoc. You can find similar API documentation for other controls; for instance, you can search MSDN for Win32 and .NET control information. Regardless of the documentation you need to use, you should be able to find the class hierarchy for your control. [Figure 13.5](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig05) shows what this information would look like for the DateTime control.

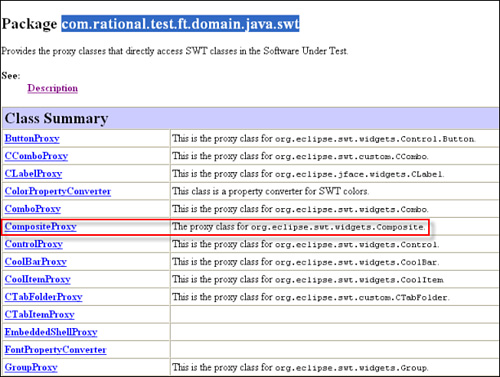
**Figure 13.5** Javadoc for DateTime control—hierarchy



Traversing the DateTime hierarchy shows you that the org.eclipse.swt.widgets.DateTime class is derived from the org.eclipse.swt.widgets.Composite class, which is derived from the org.eclipse.swt.widgets.Scrollable class, and so on. Starting at the bottom of the list with the DateTime class, you want to see which Rational Functional Tester proxy matches up. Looking in Rational Functional Tester’s help files, you can find the list of proxies for the SWT domain.

[Figure 13.6](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig06) shows that a proxy does not exist for the DateTime control. It also shows that a proxy does, however, exist for the Composite control (outlined in the red box). This matches up against the org.eclipse.swt.widgets.Composite class in the DateTime control’s hierarchy.

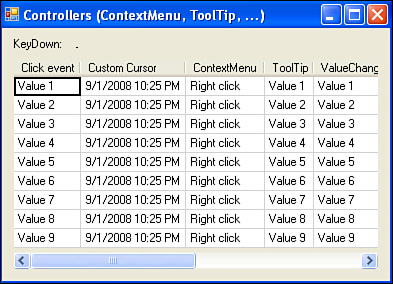
**Figure 13.6** SWT proxies—CompositeProxy



The last means of finding hierarchical information is the source code. If you have access to it, you can find the source files that contain the class your custom control is built from. Depending on your ability to read source files, this can be a difficult or easy task. It can almost seem like you are connecting the dots, looking at each class signature to see what other class it extends, and stopping only when you have found the base class.

A good example of this is DevAge’s SourceGrid control package, also covered in [Chapter 10](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10), “[Advanced Scripting with Rational Functional Tester TestObjects](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10)” (see [Figure 13.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig07)). The SourceGrid is a C# package that uses a Grid class to represent a GUI control similar to a base .NET grid.

**Figure 13.7** SourceGrid



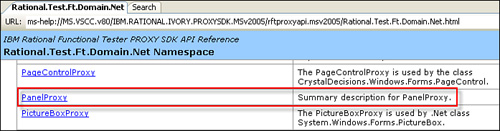
Rational Functional Tester has no understanding of this. In fact, it won’t even see it or map it (for example, you do not see the red box and tool tip around it when you are recording a verification point against it). Building a proxy for this requires that you gain an understanding of the base class. You can use the source code to “connect the dots,” starting with the Grid class, until you find it.

You can see in [Figure 13.8](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig08) that the SourceGrid Grid class extends the GridVirtual class. This is another part of the SourceGrid package. Looking further, you find that the GridVirtual class extends another custom SourceGrid class, CustomScrollControl. This class extends the System.Windows.Forms.Panel class. The System.Windows.Forms namespace tips you off that the Panel class is, indeed, the base class you are looking for. You can look in Rational Functional Tester’s Help files to find list of proxies for the .NET domain. This list would show that an out-of-the-box PanelProxy exists (see [Figure 13.9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig09)).

**Figure 13.8** SourceGrid class hierarchy

image

**Figure 13.9** .NET proxies—PanelProxy



**Creating a Proxy Project**

After you know what proxy to extend, you need to create a project for it. If you are building a proxy for a Java GUI control, you can take advantage of the Java development environment that lives in the same Eclipse shell as Rational Functional Tester. If you need to create a proxy for a .NET control, you need to build it using C#. This might be part of the .NET Studio 2003 or 2005 instance you installed Rational Functional Tester into.

**Java Project**

The first thing to state from the start: You must not use proxies that were compiled with a version of Java that is newer than Rational Functional Tester’s version of Java. This causes UnsupportedClassVersionError issues. Now that you know this, on with the proxy project topic. The Eclipse SWT DateTime GUI control example requires a Java proxy. This is handled by switching to the Java perspective in Rational Functional Tester’s Eclipse shell. You can do this by selecting: **Window** > **Open Perspective** > **Other** > **Java**. You then create a new Java project by selecting: **File** > **New** > **Java Project**. The New Java Project wizard displays. You can simply provide a name and location for your project and click the **Finish** button.

The project stores the proxy that you create. This is actually a Java class file. Prior to creating this new class file, Java development best practices suggest you create a package to store it in. This is done by right-clicking on your project and selecting **New** > **Package**. You call the package for your DateTime project **sdk.custom.swt**. This represents the custom SWT proxies you build.

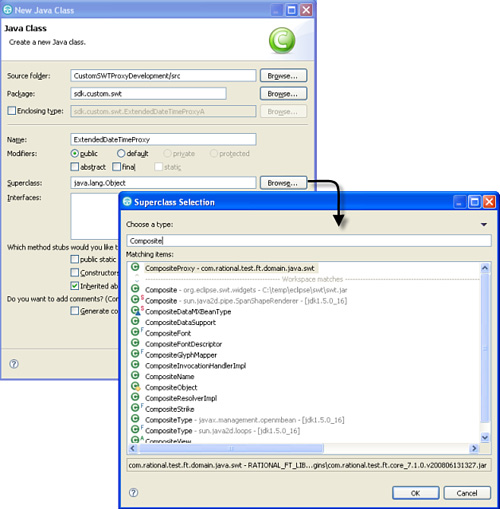
Finally, you can create the new class by right-clicking on the **sdk.custom.swt** package and selecting **New** > **Class**. This invokes the New Java Class wizard. Aside from providing the necessary package and class name information in this wizard, you need to specify the correct Superclass to extend. In this case, it is the CompositeProxy Superclass (for example, what your research returned). To make the change, do the following (see [Figure 13.10](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig10)):

**1.** Click the **Browse** button next to the Superclass text field; this launches the **Superclass Selection** dialog box.

**2.** Type **Composite** in the Choose a type: text field; this lists the classes that match your entry.

**3.** Select the **CompositeProxy** class that is part of the **com.rational.test.ft.domain.java.swt** package.

**Figure 13.10** Set CompositeProxy as Superclass



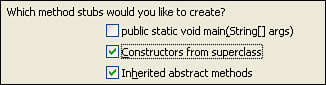
Note

If the CompositeProxy class does not show up, you need to add a reference to the RATIONAL\_FT\_LIB variable. (For example, click the Add Variable button on your project’s Java Build Path.)

**4.** Click the **OK** button.

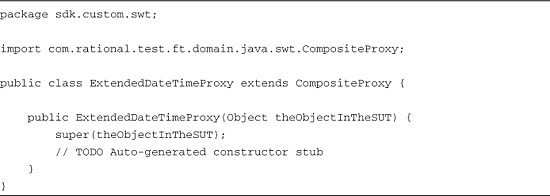
The last thing that you need to do in the New Java Class wizard, is select the **Constructors from superclass** checkbox. This ensures that the appropriate constructor method(s)—from the CompositeProxy class—are used (see [Figure 13.11](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig11)).

**Figure 13.11** New Java class wizard—constructors from superclass checkbox



Click the **Finish** button to complete the class creation. [Listing 13.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex01) shows what the resulting proxy class looks like.

**Listing 13.1** ExtendedCompositeProxy class

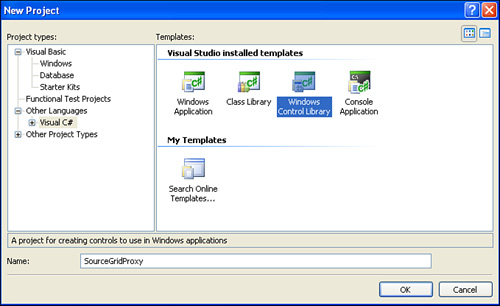


The code in [Listing 13.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex01) shows you three things. First, it shows that an import statement to the com.rational.test.ft.domain.java.swt.CompositeProxy class is automatically created. Secondly, it gives the skeleton of your ExtendedDateTimeProxy class. Finally, it shows the constructor that was created from the CompositeProxy superclass. These provide the tools for extending the capabilities of the out-of-the-box CompositeProxy class with your own.

**C# Project**

You need to create a C# proxy project for the SourceGrid GUI control. This is done by selecting **File** > **New Project** to engage the New Project wizard (see [Figure 13.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig12)). You need to select **Visual C#** as your project type. You then need to choose **Windows Control Library** as the template.

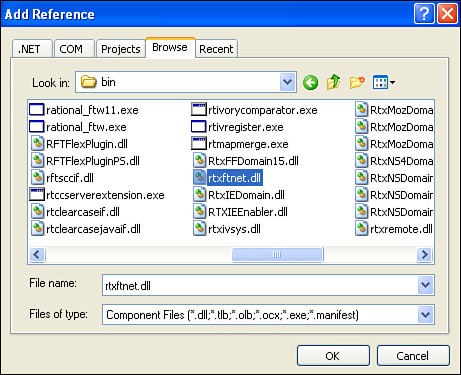
**Figure 13.12** New C# Project wizard



Provide a project name and click the **OK** button to complete the process of creating your proxy project. There are still a few setup tasks to accomplish before the proxy is ready for developing. These steps ensure that your project recognizes the different proxy objects you reference.

The first thing that you need to do is to add a reference to the rtxftnet.dll file (see [Figure 13.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig13)). This enables you to see all of the proxy information available to the .NET domain. Right-clicking on your project and selecting **Add Reference** (or selecting **Project** > **Add Reference**) displays the **Add Reference** dialog box. You need to click the **Browse** tab and then navigate to the rtxftnet.dll file found in Rational Functional Tester’s bin directory.

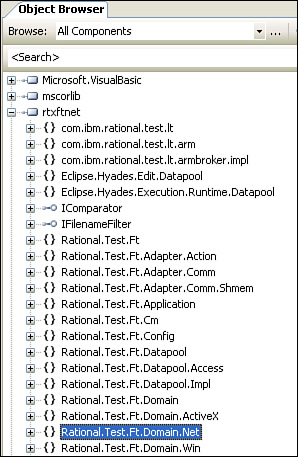
**Figure 13.13** Add project reference—rtxftnet.dll



Click the **OK** button to add the reference to your project. This enables you to select the original out-of-the-box Rational Functional Tester proxy that you need to extend. In this case, it is the PanelProxy. This gives the basis for your new proxy class.

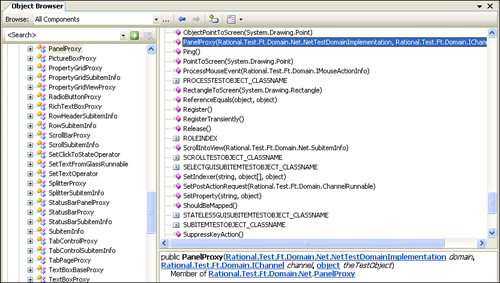
You need to find some method information for the PanelProxy. Specifically, you need to find out what constructor your class should use. The first step here is to open the **Object Browser**. You can select **View** > **Object Browser**. This lists all the object references for this project. One of those references is the .rtxftnet assembly. Expanding the .rtxftnet reference shows the different namespaces it contains. You’re interested in the Rational.Test.Ft.Domain.Net namespace (see [Figure 13.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig14)).

**Figure 13.14** Rational.Test.Ft.Domain.Net



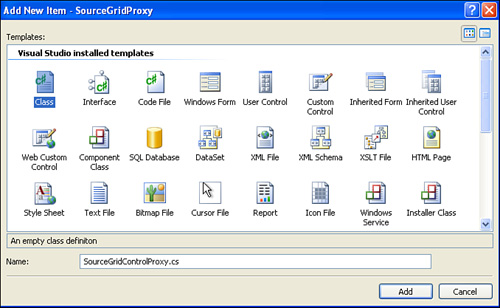
This contains the list of proxies that you can extend. Referring back to the first section in this chapter, “[Figuring out Which Proxy to Extend](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13sec1lev1),” your research shows that the PanelProxy is the out-of-the-box proxy class that you want to extend. If you expand the Rational.Test.Ft.Domain.Net namespace, you find that the PanelProxy class exists in it. Further, selecting this class and viewing its *members* (such as properties, methods, and so on) reveals the constructor you need to use in your custom proxy class (see [Figure 13.15](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig15)).

**Figure 13.15** PanelProxy class—constructor to use



All the information that you have collected up to this point enables you to quickly create the skeleton of your proxy class. This is done by right-clicking the project and selecting **Add** > **Class**. You can select the **C# Class**, provide a name for it, and click the **Add** button (see [Figure 13.16](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig16)). This gives you a starting point. Framework code is shown in [Listing 13.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex02).

**Figure 13.16** Create proxy class—called SourceGridControlProxy



**Listing 13.2** Initial class creation



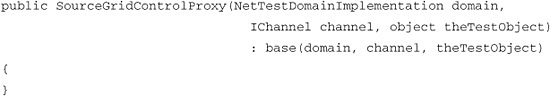
You need to modify this so that you can begin adding custom capabilities. This involves adding “using” statements for the Rational.Test.Ft.Domain and Rational.Test.Ft.Domain.Net namespaces. These namespaces provide you with access to the necessary classes for your constructor.

using Rational.Test.Ft.Domain;  
using Rational.Test.Ft.Domain.Net;

You also need to extend the Rational.Test.Ft.Domain.Net.PanelProxy class. This provides you access to all the methods to override. You can accomplish this by adding this class to the end of your class definition, separating the two by a colon. An example of what this looks like is shown in the following line of code:

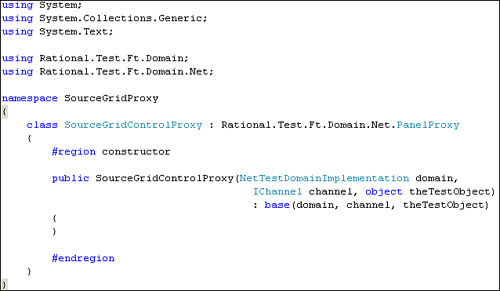
class SourceGridControlProxy :  
Rational.Test.Ft.Domain.Net.PanelProxy

The next thing is to provide the constructor information. You can copy and paste this from the PanelProxy constructor. This is easily done by returning to the Object View, expanding the Rational.Test.Ft.Domain.Net namespace, selecting PanelProxy, scrolling to and selecting the PanelProxy constructor, and copying and pasting from its definition from the bottom of the browser. You need to change the name of the constructor method from PanelProxy to the name of the proxy you are building. In your case, it is SourceGridControlProxy. The following code listing shows what your constructor should look like.



[Figure 13.17](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig17) shows what your initial class should look like.

**Figure 13.17** Initial class after modifications



One last thing to consider at this point is adding a reference to the SourceGrid.dll and DevAge.Core.dll files (found in the SourceGrid installation’s bin directory). It is not imperative that it is done at this point. However, this enables you to avoid any missing class issues as you go through the different sections.

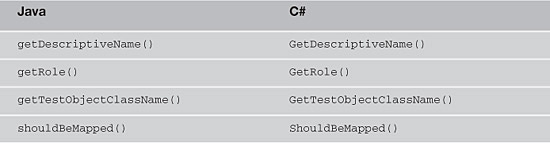
You should now have a basis for what it takes to create a new proxy. The next step is to build out your proxy by overriding the methods found in the out-of-the-box proxy class. This would be the CompositeProxy class for the DateTime control and the PanelProxy class for the SourceGrid Grid control. This is done by creating your own custom Java or C# code for these methods. You also find that you end up creating your own custom methods to handle certain supporting tasks.

**Adding Object Recognition**

Rational Functional Tester might or might not recognize custom controls in your application. If it does, it usually is able only to give you limited information and provide limited capabilities for the control. This is true for the DateTime example. Rational Functional Tester recognizes it. However, it does not provide a verification point for its object data nor does it have a selected date property. In your (.NET) example, on the other hand, the SourceGrid Grid control isn’t even recognized by Rational Functional Tester. In fact, Rational Functional Tester sees only the top-level form object that this control lives on. This mean you can’t even get basic information or capabilities for it without a proxy.

When you encounter instances like these, you need to coax Rational Functional Tester into recognizing (or further recognizing) your object. This is done with the four methods found in [Table 13.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13tab01).

**Table 13.1** Proxy Development: Object Recognition Methods to Override



**getDescriptiveName() / GetDescriptiveName()**

This method provides name capabilities for Rational Functional Tester. The name (as a string) that you return from this method is the initial name provided for the TestObject that gets created when you record against the application’s custom control.

**getRole() / GetRole()**

This method tells Rational Functional Tester the type of control that it is dealing with. This sets the icon that Rational Functional Tester uses to describe the TestObject that communicates with your proxy.

**getTestObjectClassName()**

Overriding this method tells Rational Functional Tester what type of TestObject a script should use to represent your control.

**shouldBeMapped() / ShouldBeMapped()**

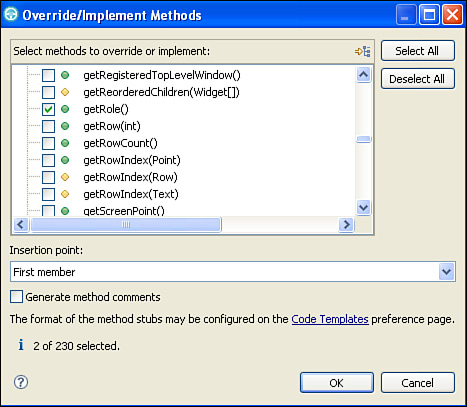
This method tells Rational Functional Tester that it should map the custom control, providing a TestObject wrapper class for the proxy class. Basically, this puts the custom control into the Test Object Map.

*Overriding* these methods in your proxy class enables Rational Functional Tester to do more with the custom control than it can using the out-of-the-box proxy.

**Java—Using the DateTime Example**

The Eclipse Java perspective makes it easy to override methods. You can simply select **Source** > **Override/Implement Methods**. You are presented with a dialog box (see [Figure 13.18](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig18)) where you select the methods that you want to override (or implement). Select the methods and click the **OK** button, which puts the stub code into your proxy for you. It is then up to you to change the definition of the method (that is override it).

**Figure 13.18** Override and implement methods



Because the DateTime example is recognized by Rational Functional Tester, it is safe to say that it is being mapped. You just want to make some changes so you can help Rational Functional Tester understand a little more about it. Therefore, you can simply override the getDescriptiveName(), getRole(), and getTestObjectClassName() methods.

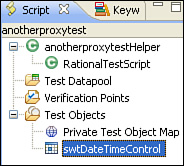
**getDescriptiveName()**

Here is how the getDescriptiveName() method was overridden for your ExtendedDate TimeProxy class. You tell Rational Functional Tester that the TestObject for the DateTime class should be named SWT DateTime Control.

Image

In the preceding example, the initial name for the TestObject is swtDateTimeControl (see [Figure 13.19](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig19)).

**Figure 13.19** Initial name of TestObject

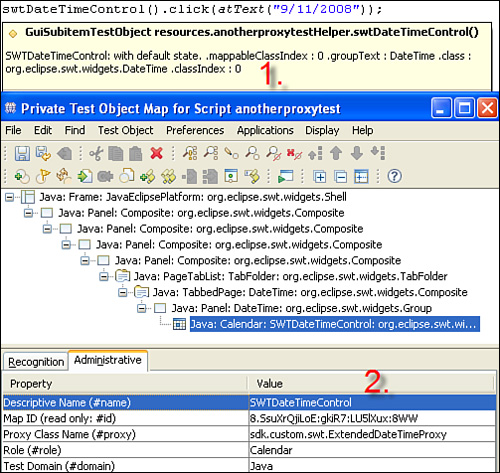


When Rational Functional Tester records and maps the DateTime control, it causes the following two mapping behaviors to occur (see [Figure 13.20](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig20)):

**1.** The Descriptive Name displays in the tooltip when you place your cursor on the object in your script.

**2.** The name you provide is placed in the Administrative Properties of the Test Object Map.

**Figure 13.20** Result of getDescriptiveName()



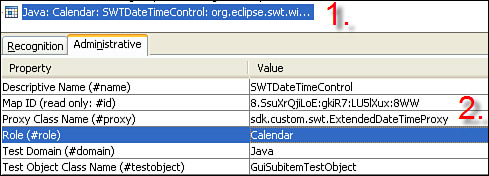
**getRole()**

Here is how the getRole() method is overridden in your proxy. You tell Rational Functional Tester that the TestObject that is created for the DateTime control should be a calendar.

Image

When Rational Functional Tester interacts with and maps the DateTime control, it does two things (see [Figure 13.21](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig21)). First, it associates a calendar icon with the DateTime Test Object it created. Second, it captures its role in the Administrative Properties of the TestObject Map.

**Figure 13.21** Result of overriding getRole()



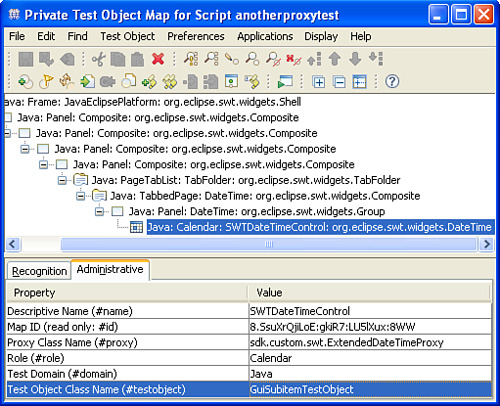
**getTestObjectClassName()**

Here is the example for overriding the getTestObjectClassName for your DateTime proxy. You tell Rational Functional Tester that it should treat the resulting TestObject as a GuiSubItemTestObject.

Image

This gives it capabilities that can be taken advantage of later. For example, it allows for Subitems to be passed to its click() methods (for example, atText(), atCell(), atIndex(), and so on). Like the other two methods you overrode, this information gets stored in the Administrative Properties of the TestObject Map (see [Figure 13.22](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig22)).

**Figure 13.22** Result of overriding getTestObjectClassName()



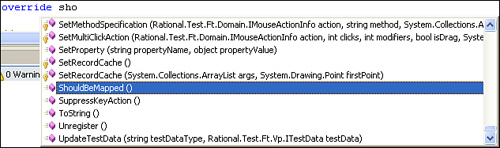
Note that you might need to import the ProxyTestObject class if it isn’t initially recognized (for example, an error marker shows up on the line). You can do this by right-clicking the error marker and choosing Quick Fix. You can also engage the **CTRL+SHIFT+O** keystroke combination. This imports any necessary packages/classes into your class files. In this case, it imports the com.rational.test.ft.domain.ProxyTestObject into your proxy class file. Moving forward with this chapter, you might need to import the necessary packages and classes as you go. The keystroke combination mentioned previously helps you deal with these imports.

**C#—Using the SourceGrid Grid Example**

The proxy you build for the SourceGrid Grid control is somewhat similar to the previous Java example. You override the GetDescriptiveName(), GetRole, and GetTestObjectClass Name() methods. One key difference is that you also override the ShouldBeMapped() method. If you recall from the first section in this chapter, “[Figuring Out Which Proxy to Extend](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13sec1lev1),” Rational Functional Tester is unable to recognize and map this control. Overriding this method enables you to change that behavior.

The C# editor in Visual Studio makes it easy to override methods. You simply need to type out the keyword **override**, press the Space key, and start typing the name of a method (see [Figure 13.23](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig23)). This shows you the list of methods that you can override for your proxy.

**Figure 13.23** Overriding in C#

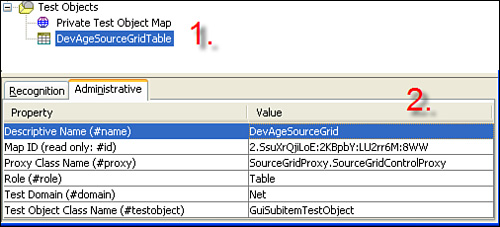


**GetDescriptiveName()**

You can return “DevAge SourceGrid” here. This provides the initial name for the TestObject that is recorded, as well as set up the Administrative properties (see [Figure 13.24](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig24)).

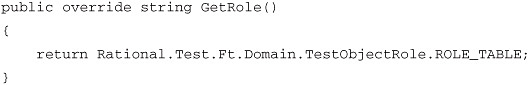


**Figure 13.24** Result of overriding GetDescriptiveName()

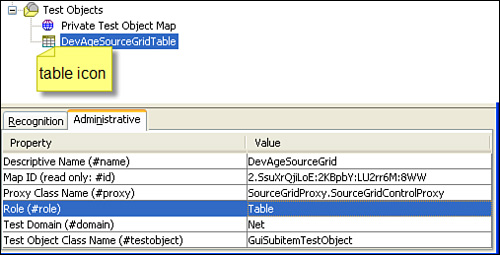


**GetRole()**

Providing the following code enables your proxy to inform Rational Functional Tester that it should treat the resulting TestObject as a table. This provides a table icon for the TestObject. It also updates the Administrative Properties in the Test Object Map (see [Figure 13.25](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig25)).

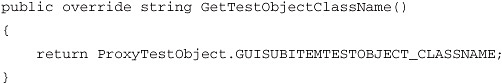


**Figure 13.25** Result of overriding GetRole()

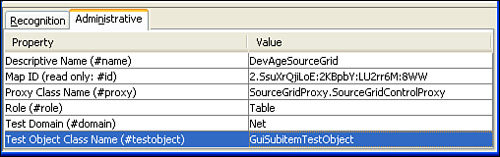


**GetTestObjectClassName()**

This example for overriding the getTestObjectClassName() tells Rational Functional Tester that it should treat the resulting TestObject as a GuiSubItemTestObject. This enables you to add some more functionality, such as using Subitems (atIndex(), atCell(), atText(), and so on) to your clicks if you need to (see [Figure 13.26](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig26)).

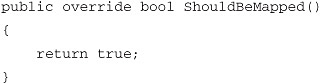


**Figure 13.26** Result of overriding GetTestObjectClassName()



**ShouldBeMapped()**

Overriding this method is quite simple. You return true, telling Rational Functional Tester that it should map this control, providing a TestObject class wrapper.



As you can see in [Figures 13.24](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig24), [13.25](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig25), and [13.26](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig26), overriding this method to return true resulted in the SourceGrid.Grid class getting mapped. Rational Functional Tester now gives you a TestObject class for this.

At this point, you should understand how to develop your proxy to instruct Rational Functional Tester how to recognize (or better recognize) your application’s custom control. This is a good place to start when first building a proxy. In some instances, it sets the stage for other pieces of your proxy development (for example, working with click actions and using Subitems).

**Verifying Object Properties**

Adding custom verifications is a big part of proxy development. Sometimes it is the only reason you build a custom proxy. The out-of-the-box proxy doesn’t provide the verification that you want to implement; therefore, you need to extend it to accommodate the test you want to perform. One way to do this is by overriding the following two methods in [Table 13.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13tab02).

**Table 13.2** Proxy Development: Object Property Methods to Override

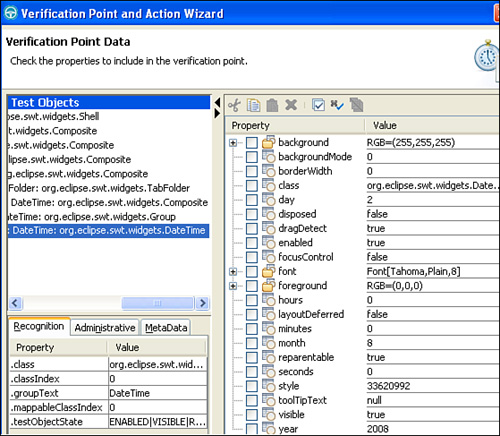


These methods are used to add new properties to the list exposed by the TestObject. Overriding the getProperties()/GetProperties() method enables you to expose your own custom properties to Rational Functional Tester’s Verification Point Wizard. Further, you are able to use the Test Object Inspector to see these properties when you hover over your custom control. The getProperty()/GetProperty() method enables you to create your own custom properties, making them accessible through the TestObject API method getProperty()/GetProperty(). The following examples show that you can combine these methods in a clever way so that getProperties()/GetProperties() uses getProperty()/getProperty() to do its work.

**Java—Using the DateTime Example**

The first section in this chapter, “[Figuring Out Which Proxy to Extend](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13sec1lev1),” mentioned that the DateTime GUI control did not contain a selected date property. If you use an Object Properties Verification Point with the out-of-the-box property, you do not see a property that tells you what the selected date is (see [Figure 13.27](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig27)).

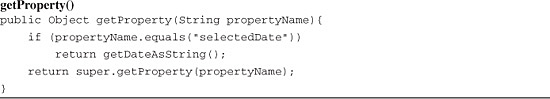
**Figure 13.27** Object Properties Verification Point—no “selected date” property



Overriding the getProperties() and getProperty() methods in a proxy enable you to create a new property, selectedDate, and add it to the list of properties Rational Functional Tester can test for. See [Listing 13.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex03).

**Listing 13.3** Overridden methods that create a new property



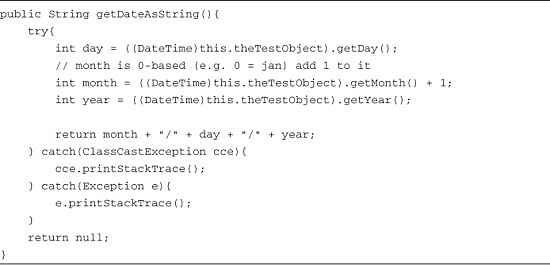


Reviewing the two code listings shows that getProperty() services getProperties(). The code for getProperties() is simple. It makes a call to the out-of-the-box proxy’s getProperties() method to get the list of properties it normally captures. It stores this in a Hashtable, aptly called properties. This enables you to manage the list of properties as a list of names and values. The getProperties() method then adds your custom property to the properties Hashtable. It uses selectedDate as the name and calls the getProperty() method to get the value. If it is successful on adding the new property() to the Hashtable, it returns it. It if is not successful, it writes out a debug message, returns the original set of properties, and ignores your attempt to synthesize your own property.

The overridden getProperty() method is straightforward to understand. It accepts the property name as a string argument called propertyName. In your example, this is selectedDate. If the propertyName argument equals selectedDate, it calls another method, getDateAsString(), to handle attaining the property value to return. If the propertyName argument is not selectedDate, it calls the out-of-the-box proxy’s getProperty() method, passing in propertyName and returning the value it acquires.

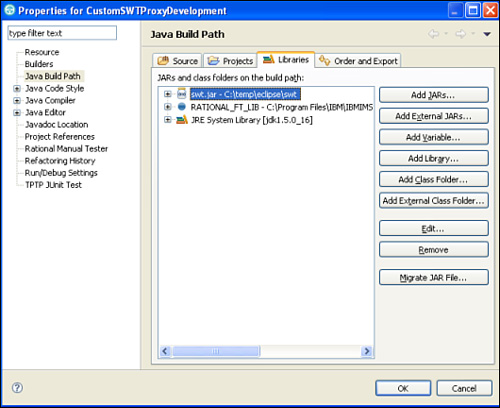
You saw that the overridden getProperty() method called out to another method to do its work, getDateAsString(). It is a custom method written for the express purpose of returning the value of a selected date (see [Listing 13.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex04)).

**Listing 13.4** getDateAsString()—custom method



The first thing getDateAsString() does is try to cast the object reference that is being manipulated as a DateTime class. When you see this.theTestObject, you use a stolen reference to the control object for which you write the proxy. This is the GUI object that Rational Functional Tester records against. Casting it to the DateTime class enables you to call the methods needed to get the day, month, and year that are currently selected. These are getDay(), getMonth(), and getYear(), respectively. To use the DateTime class, you need to place the swt.jar file in your proxy project’s Java Build Path (see [Figure 13.28](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig28)).

**Figure 13.28** swt.jar added to proxy project’s Java Build Path



The cast also requires you to add the following imports statement at the top of your class file:

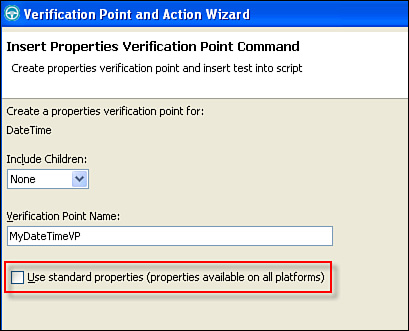
import org.eclipse.swt.widgets.DateTime;

If the cast is successful, you are able to manipulate the different methods of the DateTime class. If it isn’t successful, the ClassCastException is caught and handled, printing out the stack trace.

Because the goal of this method is to return a selected date value, you can use the following DateTime methods: getDay(), getMonth(), and getYear(). They each return an integer that represents their specific piece of the date that was selected by the end user. For instance, if getDay() returns 15, it represents that the 15th day of the year was selected. The getYear() method is self-explanatory, returning the selected four-digit year. Finally, the getMonth() method requires a little bit of extra processing. This is because it returns the selected month based off a 0-based numbering. In other words, 0 represents January, 1 represents February, and so on. This is easily resolved by adding one to the selected month. The return value for this method is a string. This string gets constructed and returned as the following “selected date” format: month/day/year.

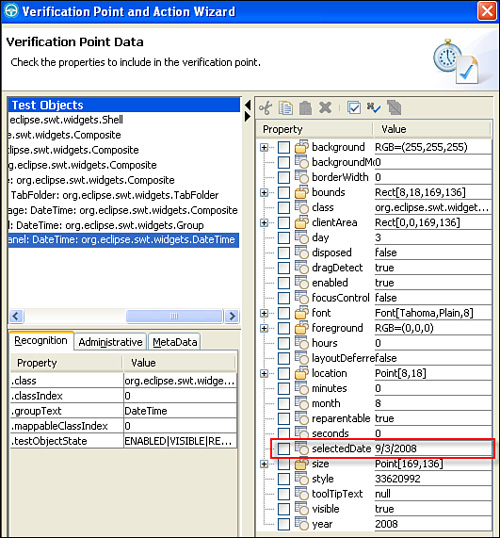
Successfully overriding these methods results in Rational Functional Tester adding the custom selectedDate property to its Object Properties Verification Point wizard. To see your custom property in the list, you need to uncheck the **Use standard properties** checkbox (see [Figure 13.29](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig29)).

**Figure 13.29** Uncheck the Use standard properties checkbox



This shows you all the properties that you can test for, including your own custom property (see [Figure 13.30](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig30)).

**Figure 13.30** Object Properties Verification Point—selectedDate property



You are also able to obtain the value of this property using the getProperty() and getProperties() TestObject API scripting methods (see [Listing 13.5](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex05)).

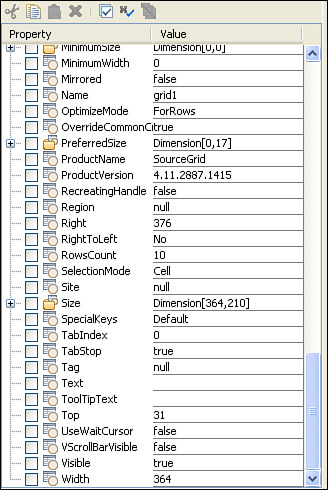
**Listing 13.5** TestObject API methods

Image

**C#—Using the SourceGrid Grid Example**

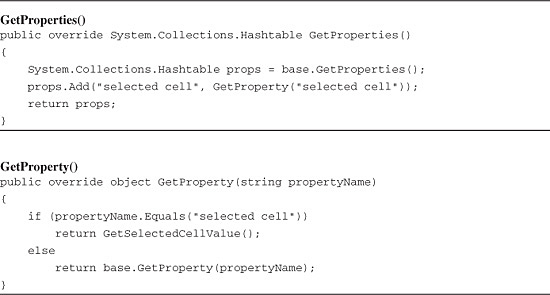
The SourceGrid control suffers a similar fate as the DateTime control. A key property is missing (see [Figure 13.31](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig31)).

**Figure 13.31** Object Properties Verification Point—missing selected cell property



You can see that simply by using the out-of-the-box proxy’s properties Hashtable, you can get key values such as row count, column count, object name, and so on. Unfortunately, a property does not exist to tell you which cell is currently selected. This can easily be addressed by overriding the GetProperty() and GetProperties() methods (see [Listing 13.6](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex06)).

**Listing 13.6** Overridden property handling methods

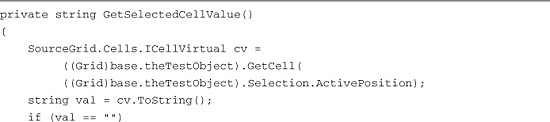


The behavior of this is similar to the Java example. The GetProperties() method makes an initial call out to the out-of-the-box proxy method GetProperties(). This returns the list of properties that the base proxy can acquire for you. It stores things in a Hashtable called props, so you can cache the base name and value pairs. It then adds a synthetic property to the props Hashtable, passing “selected cell” in for the name and calling GetProperty() to get the value. It passes “selected cell” to the GetProperty() method.

The GetProperty() method accepts one argument: propertyName. This is a string. It checks to see if the propertyName string matches “selected cell”. If it does, it calls the GetSelectedCellValue() method to acquire the actual value of the selected cell. If it doesn’t match, this method passes the propertyName string to the base class GetProperty() method, returning the value that was obtained.

The GetSelectedCellValue() method is the helper that handles acquiring the value for a selected cell. It is not an overridden method. It is a custom method created specifically to help with the other two overridden methods: GetProperties() and GetProperty().

**Listing 13.7** GetSelectedCellValue()—custom method



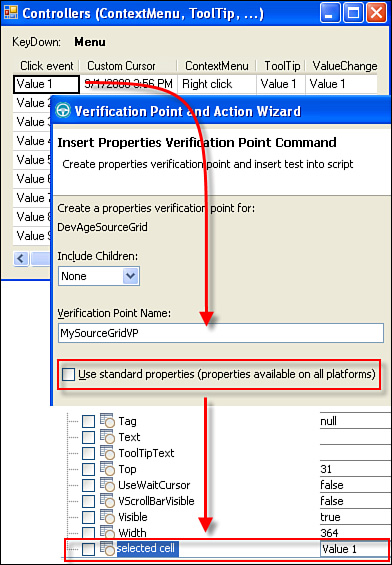
Image

The first thing this method does is create a SourceGrid.Cells.ICellVirtual variable called cv. It then casts the stolen object reference—currently being manipulated by Rational Functional Tester—as a SourceGrid.Grid class twice: first to call the Grid’s getCell() method and second to pass the "Section.ActivePosition" property as the argument to the getCell() method. To do this, you need to add the following using statement at the top of your class:

using SourceGrid;

By overriding the GetProperty() and GetProperties() methods and using a custom method to acquire the selected cell value, you are able to record an Object Properties Verification Point that contains a “selected cell" property (see [Figure 13.32](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig32)).

**Figure 13.32** Object Properties Verification Point—selected cell property exists



You are also able to obtain the value of this property using the getProperty() and getProperties() TestObject API scripting methods ([Listing 13.8](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex08)).

**Listing 13.8** TestObject API methods

image

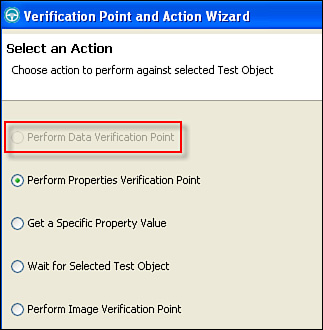
**At This Point**

The basic concept for adding custom property tests to your proxies is overriding two base class methods and creating any necessary method(s) to handle the data acquisition. It is critical that you have an in-depth understanding of the custom control for which you are creating the proxy. You should make sure that you have a copy of the custom control’s API documentation. It behooves you to stay friendly with the development team. This increases your ability to quickly find answers to your questions.

**Verifying Object Data**

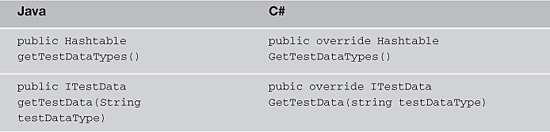
A popular task you might run into while developing your proxy is adding a custom Object Data Verification Point. The control you are working with might not provision its data to Rational Functional Tester in a typical manner. In other words, when you record your script and go to create an Object Data Verification Point, a data type (for example, the type of date test you are looking to use) doesn’t exist for what you are looking to do. You might even run into the instance where the Verification Point Wizard doesn’t even give you the option to test for object data (see [Figure 13.33](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig33)).

**Figure 13.33** No Object Data Verification Point



You can add this functionality into your proxy teaching, Rational Functional Tester how to record an Object Data Verification Point for your control, by overriding the two methods shown in [Table 13.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13tab03).

**Table 13.3** Proxy Development: Object Data Methods to Override



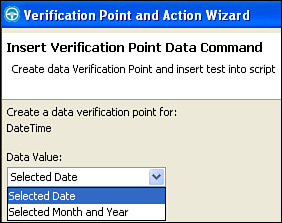
**Java—Using the DateTime Example**

You need to start with the getTestDataTypes() method. This is how you tell Rational Functional Tester what types of object data tests you want to see in the Verification Point Wizard. Using class-level constants in your proxy class enables you to capture and reference the data types you want. You can accomplish this with string variables, making them private, static, and final. This helps you when you need to refer to your test types in other parts of the proxy (that is the two data methods you are going to override). You don’t have to worry about typing the same strings each time. You can simply refer to the variables that you create. Referring back to the DateTime object referenced in the first section of this chapter, you might use the following:



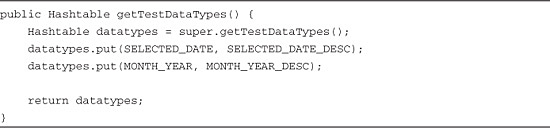
The code listing tells Rational Functional Tester that you want to create two data tests. The first is for the data currently selected in the DateTime control. The second captures the currently selected month and year in the header of control. Note that there are two lines for each test type. The first is the actual type. The second is the description of the type. The description shows up in the Verification Point Wizard as the test selection (see [Figure 13.34](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig34)).

**Figure 13.34** Object Data Verification Point—custom data tests



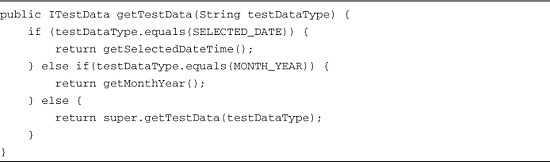
After creating the data types, you now simply override the getTestDataTypes() method. This is accomplished by first making a call to the getTestDataTypes() method that belongs to the base class. This is the proxy that you are extending. In the DateTime example, it is the com.rational.test.ft.domain.java.swt.CompositeProxy class. Calling this method returns any data types that might exist for the base class. These are stored in a Hashtable. The next thing you need to do is add your new data types to the Hashtable. Lastly, you simply return the Hashtable (as required by the getTestDataTypes() signature). [Listing 13.9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex09) shows how this is implemented for the DateTime example.

**Listing 13.9** getTestDataTypes()—overridden method



The next thing you need to do is override the getTestData() method. This method actually captures and returns the data from your control and displays it in the Object Data Verification Point. This method returns an ITestData object. This object contains the values that you want to display in the Object Data Verification Point. Overriding this method is actually simple. It is a matter of creating the conditional logic that figures out the test data type that is selected in the Verification Point Wizard. You can easily handle this with an if...then...else block. [Listing 13.10](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex10) shows this for the DateTime control example with which you have been working.

**Listing 13.10** The getTestData()—overridden method



The previous code listing shows that the conditional logic checks to see which data type was chosen in the Verification Point Wizard. If it is either of your custom date types—SELECTED\_DATE or MONTH\_YEAR, respectively—it calls out to a secondary method to capture and return the requested data. If the selected value was something else (such as you hit the “else” piece of your logic); it calls out to the getTestData() method of the base class, returning its value if it exists. Note the bulk of the work is handled in another method. Overriding the getTestDataTypes() method is just a matter of figuring out which other method you need to write and then invoke to acquire and return the appropriate data.

In the case of the SELECTED\_DATE type chosen, you need to capture and return the currently chosen date with a method similar to [Listing 13.11](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex11).

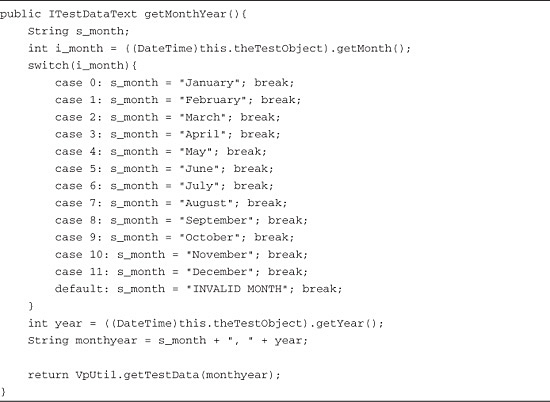
**Listing 13.11** getSelectedDate()—custom method



The getselectedDate() method relies on the getDateAsString() method that was defined in the “[Verifying Object Properties](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13sec1lev4)” section. As a refresher, the getDateAsString() method builds a date in the format of month/day/year, returning it as a string. This value is passed to VpUtil.getTestData, creating and returning an ITestDataText object to the invoking getTestData() method.

You also need a method to handle the MONTH\_YEAR data type as well. This is something like what you can see in [Listing 13.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex12).

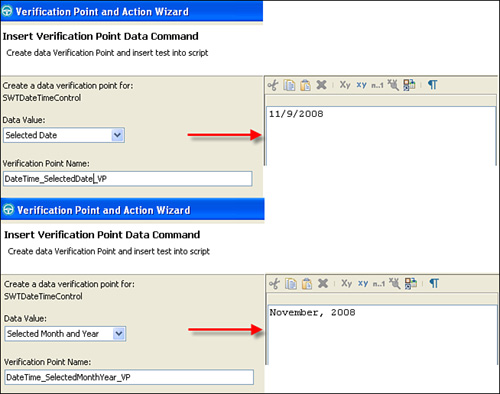
**Listing 13.12** The getMonthYear()—custom method



This method behaves similarly to the getDateAsString() method. It casts the stolen reference to a DateTime reference. To reiterate, the cast occurs so you can access the methods necessary for obtaining certain information from the DateClass GUI Object that Rational Functional Tester is recording against. In this case, you want to capture and return the month and year that are currently displayed in the control’s header. A switch statement is used to convert the month from an integer (returned by the getMonth() method) to a string, starting with 0 to represent January and ending with 11 to represent December.

Rational Functional Tester now has two Object Data Verification Points that it can use (see [Figure 13.35](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig35)). One tests for the selected month and year. The other tests for the selected date.

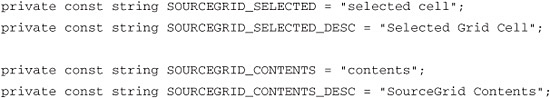
**Figure 13.35** New Object Data Verification Points



**C#—Using the SourceGrid Example**

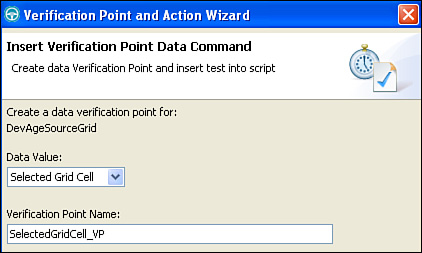
The first method to override is getTestDataTypes(). This method instructs Rational Functional Tester on the types of object data tests to perform. You can use class-level constants in your proxy to capture and reference the data types you want. This is accomplished using string variables with the private and const modifiers. Doing this helps you when you need to refer to your test types in other parts of the proxy (that is the two data methods you are going to override). It is also beneficial if you need to change the string itself. You only have to change it in one location.

The SourceGrid example is missing a data test for the selected cell. You can create four constants to inform Rational Functional Tester that there two new data tests: "selected cell" and "contents".



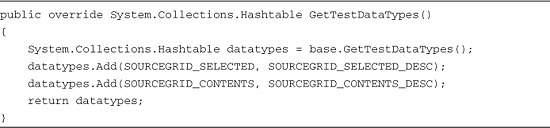
Rational Functional Tester is now aware that there are new data tests for its Object Data Verification Point. Note that there are two lines for each test type. The first is the actual type. The second is the description of the type. The description shows up in the Verification Point Wizard as the test selection (see [Figure 13.36](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig36)).

**Figure 13.36** Object Data Verification Point—custom data test



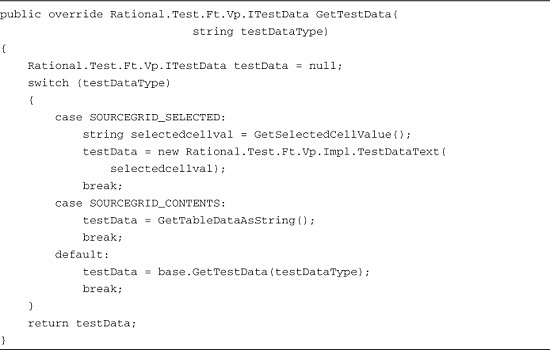
Now that you have the data test specified, you can override the GetTestDataTypes() method. This is accomplished by first making a call to the GetTestDataTypes() method that belongs to the base out-of-the-box proxy class. This is the PanelProxy for this example. Calling this method returns any data types that might exist for the base class. These are stored in a Hashtable, called datatypes. You can place new test types into the datatypes using its Add() method. The last step is to return the datatypes Hashtable. [Listing 13.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex13) represents the steps you just discussed.

**Listing 13.13** GetTestDataTypes()—overridden method



The next step is to override the GetTestData() method. This method captures and returns the data from your control and displays it in the Object Data Verification Point. It returns this data as an ITestData object and contains the values that you want to display in the Object Data Verification Point. Overriding this method is a matter of creating conditional logic that figures out the data test that the Verification Point Wizard requests. You can easily handle this with a switch statement (see [Listing 13.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex14)).

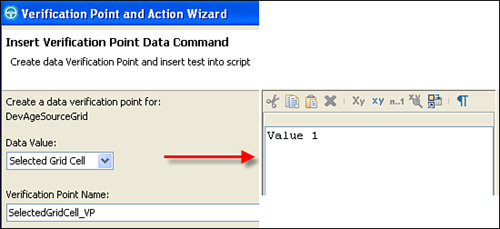
**Listing 13.14** The GetTestData()—overridden method



Your overridden version of the GetTestData() method seeks to collect the string representation of a selected cell value and return it as an ITestData object, called testData. This object is initialized with null. However, it is assigned a new value after execution reaches the switch statement. This statement first checks to see if the testDataType string (the argument passed into the GetTestData() method) matches the SOURCEGRID\_SELECTED constant string (also known as the "selected cell"). If it does match, it calls GetSelectedCellValue() to obtain the value of the selected cell and return it as a string. This value is stored in testData. You head down a similar path if testDataType matches the SOURCEGRID\_CONTENTS constant. The GetTableDataAsString() method is invoked and returns the contents of the SourceGrid control. If testDataType does not match either SOURCEGRID\_SELECTED or SOURCEGRID\_CONTENTS, it calls the out-of-the-box proxy’s GetTestData() method, passing testData Type along to it and storing the value that comes back into testData. Finally, the method returns testData and displays the results in the Object Data Verification Point.

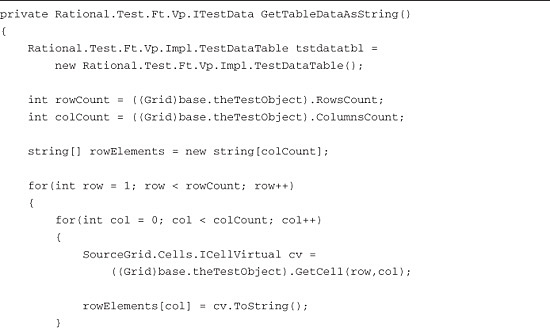
Your overridden GetTestData() method relies on GetSelectedCellValue() to capture and return the desired date. This is the same method that is used in the “[Verifying Object Properties](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13sec1lev4)” section. It is used to populate the custom object property, "selected cell". It can be reused here to capture the value to display in your data verification point. These three methods (see [Figure 13.37](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig37)) enable Rational Functional Tester to invoke your desired data verification.

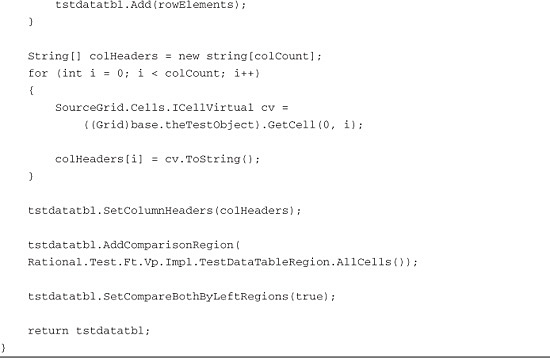
**Figure 13.37** New Object Data Verification Point—selected cell value



GetTestData() also invokes another helper method, GetTableDataAsString(), to obtain the contents of the SourceGrid Grid (see [Listing 13.15](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex15)).

**Listing 13.15** GetTableDataAsString()—custom method





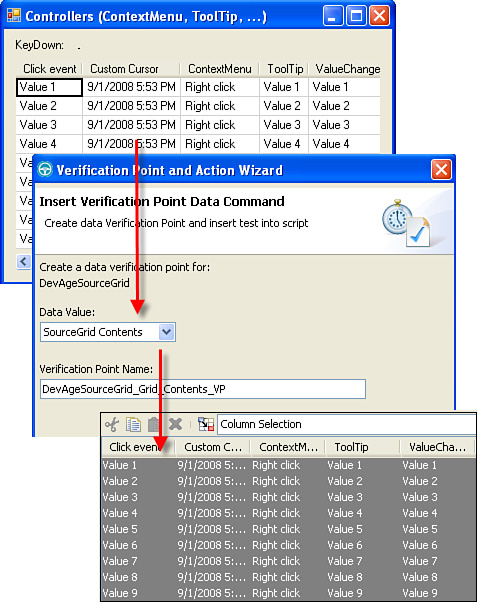
GettestDataTableAsString() starts off by creating a new TestDataTable class called tstdatatbl. This is used to capture and return the contents of the grid. It then captures the grid’s row and column counts and casts the TestObject reference to a SourceGrid Grid class. The column count is used to initialize the number of cells for your rowElements array. This array is used to store the string values of each cell in the grid. A for loop is employed to access the rows of the grid. It is important to note that the loop starts with the second row of the grid (**int row =** 1). The first row contains the column headers. This information is garnered from reviewing the documentation for this control. For each row encountered, another for loop is used to walk through each column of the row, pulling out the string values and storing them in the rowElements array along the way. Every row is added to tstdatatbl. Ultimately, when the for loops end, the grid data is fully represented in this TestDataTable class.

The next thing your method does is set the column headers of tstdatatbl. This is done by looping through the first row of the grid. As mentioned in the preceding paragraph, this information is obtained from reviewing the control’s documentation.

After the grid data and column header information is set, you need to set a couple of flags. The first flag says that you want to look at all the cells in the grid. The second flag states that you want to compare the data by using the comparison regions of both left and right values.

The end result of this method, in conjunction with the overridden GetTestDataTypes() and GetTestData() methods, is a verification point that looks like [Figure 13.38](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig38).

**Figure 13.38** New Object Data Verification Point—grid contents



**At This Point**

As you can see, the general idea behind adding custom data tests to your proxies is a matter of overriding two base class methods and creating your own custom method(s) to handle the majority of the data acquisition. A key facet of this is having an understanding of the class to which you are casting the TestObject wrapper class. This provides you with the necessary methods needed to capture the data that satisfies your test. You should make sure that you have a copy of the custom control’s API documentation. Another channel for gaining this knowledge is your development team. Keeping lines of communication open with them greatly enhances your ability to quickly find answers to your questions.

**Executing a Click Using Object Information**

Object-oriented clicking is a primary need for automation tools. It makes an action relative to a piece of the object, for instance, clicking on a particular cell (row, column) of a table, selecting an item in a combobox by its text, and so on. This eliminates the dependencies on screen or top-level object coordinates for navigation. It ultimately enables an automation tool to find an object, regardless of where it has moved to on its parent control (for example, a Windows form, HTML document, and so on).

You usually find that if you need to build a proxy so that Rational Functional Tester can recognize and interact with your control, you need to add object-oriented clicking to your to do list The implementation of this is somewhat similar to the other tasks that you have seen. You need to override a series of methods in the base out-of-the-box proxy to tell Rational Functional Tester how to handle object-oriented clicking for your custom control.

**Java—Using the DateTime Example**

Rational Functional Tester can navigate only the DateTime control via coordinate-based clicking. When you select a particular date, you see a line in your script similar to the following:

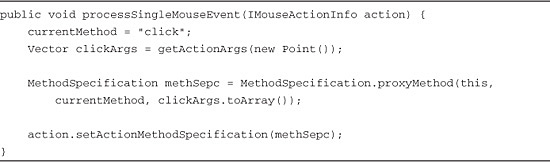
dateTime().click(atPoint(112,81));

This is what is meant by coordinate-based clicking. Rational Functional Tester maps the DateTime control, creating a TestObject class-wrapper for it. Unfortunately, it can’t figure out how to tell you what date it clicked on. You need to add capability to your proxy to help teach Rational Functional Tester how to do this. You need to override methods for both record and playback.

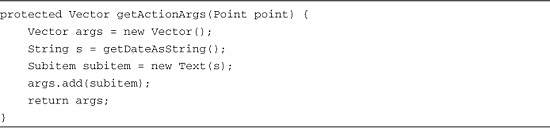
There are different record mouse event methods that you can override. For example, you can choose ProcessMouseEvent(), ProcessSingleMouseEvent(), ProcessHoverMouseEvent(), and so on. The method(s) that you choose are dependent upon the control for which you are writing the proxy. This requires you to have a good understanding of your control. You need to know the different methods that you can use to help provide object-oriented information for a click event.

The purpose of overriding the mouse event methods is to convert a point—clicked on by Rational Functional Tester while recording—to an object-oriented subitem. This can be an index, text string, table cell, and so on. The DateTime doesn’t readily expose methods that enable you to relate a point to it, extracting the object-oriented information that you want. You can spend time researching how to accomplish this task, working with developers, reading through Javadoc, and experimenting. To keep things simple, you can cheat some here using the DateTime’s internal methods for aggregating the day, time, and year pieces of the date that was clicked on. This task is handled by overriding the processSingleMouseEvent() and getActionArgs() methods (see [Listings 13.16](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex16) and [13.17](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex17), respectively).

**Listing 13.16** processSingleMouseEvent()—overridden method



**Listing 13.17** getActionArgs()—overridden method



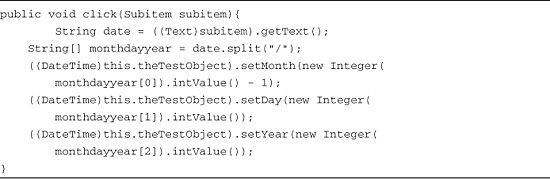
Overriding the processSingleMouseEvent() method involves skipping over the point-to-object-oriented information conversion. Instead, after setting current method to a click, it makes a call to the getActionArgs() method, passing in an arbitrary point. In other words, it invokes the method without a true need for the point. It takes the returned clickArgs vector and uses it along with the currentMethod string ("click") and current proxy test object (this) to create a new method for the recorded event.

The processSingleMouseEvent() method relies on getActionArgs() to provide the object-oriented information for the click event to use (see [Listing 13.17](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex17)). Your proxy bases your click off the selected date.

Overriding this extensible method is quite simple, at least for your purposes. You basically create a new vector and call the getDateAsString() method. The getDateAsString() method was defined in the “[Verifying Object Properties](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13sec1lev4)” section. It stores the selected date in the format month/day/year in a string variable called s. You then create a new Subitem class, aptly named subitem, and store a new Text object in it, based off your string variable s. This is ultimately the atText("month/day/year") subitem that you want to see in your recorded click statements.

Defining a custom, object-oriented means of recording a click requires you to create a custom means of playing it back. You simply work your way backward. This means that for the DateTime control, you break the date string into its individual day, month, and year parts and set the selected calendar date with them. Because you cheated the record methods, you can cheat the playback. You need only develop a custom click() method (see [Listing 13.18](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex18)).

**Listing 13.18** click()—custom method



This method converts the subitem argument into a text string. This is the month/day/year string found in your object-oriented click. You can easily parse this string into its separate day, month, and year pieces using the split() method. This returns an array that contains the individual parts. You use these parts as arguments to the DataTime methods for setting the month, day, and year. This is accomplished by casting the stolen reference to your control object (that is, this.theTestObject) as a DateTime class. Doing this exposes setMonth(), setDay(), and setYear(). Passing in the respective array values as arguments to these methods sets the calendar to the date on which you clicked.

These three methods provide you with a means to eliminate the coordinate-based clicks. You now see the following type of click instead:

swtDateTimeControl().click(atText("11/11/2008"));

**C#—Using the SourceGrid Grid Example**

You most likely need to implement object-oriented clicking in your proxy. You can come to this conclusion based off the simple fact that Rational Functional Tester won’t recognize the SourceGrid Grid control without your proxy. You had to override the ShouldBeMapped() method to tell Rational Functional Tester that it should include this control in its map. If you need to validate your assumption, start recording a script and click on the SourceGrid control. This results in the following line generated in your test script:

DevAgeSourceGridTable().Click(AtPoint(24,104))

You can see that Rational Functional Tester clicks on the control via coordinates. Having confirmed your initial thoughts, you can add object-based clicking in your proxy. For your purposes, you need some out-of-the-box proxy methods. You can break these down into the following two categories:

**Record Methods**

ProcessMouseEvent()

ProcessPreDownMouseEvent()

getActionArgs()

**Playback Methods**

GetSubitemRect

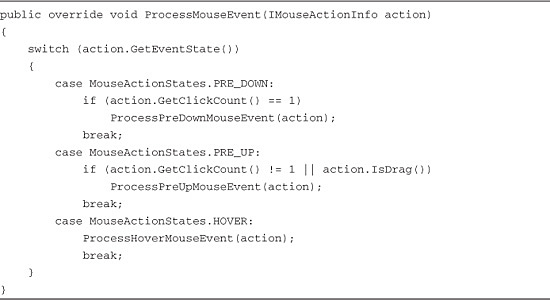
You need also to create a custom Click() method to tell Rational Functional Tester what you want it to do.

The reason you break things down into record and playback methods is due to the fact that you need to construct how you tell Rational Functional Tester to record the click against your custom control and then how to play those clicks back. Let’s start with the record methods.

**Record Methods**

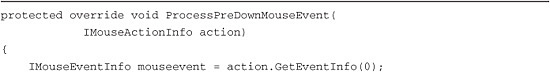
The ProcessMouseEvent() is the first method that gets called when you record a mouse event. At a high level, it just sorts through the type of event that was just recorded. Questions to ask include: Was it a hover event? Was it an event prior to a mouse button being depressed? You can probably guess that the method contains logic to figure this out (see [Listing 13.19](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex19)).

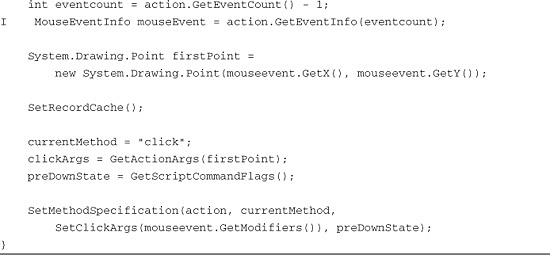
**Listing 13.19** ProcessMouseEvent()



The primary logic is contained in a switch statement. This looks at the event state of the mouse action and matches it to one of three cases. If the event state is a PRE\_DOWN state (prior to the mouse button being fully depressed), the number of clicks is collected and then the ProcessPreDownMouseEvent() is called. A similar series of actions occur if you hit a PRE\_UP state, testing for the click count and even testing if you encounter a drag action prior to the mouse button be released. The ProcessPreUpMouseEvent() methods gets called in this case. The HOVER case simply calls out to its event handling method, ProcessHoverMouseEvent. Because you are concerned with capturing object-related clicks, you can focus on collecting them as the mouse button is depressed. Therefore, you override the ProcessPreDownMouseEvent() method (see [Listing 13.20](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex20)).

**Listing 13.20** ProcessPreDownMouseEvent()

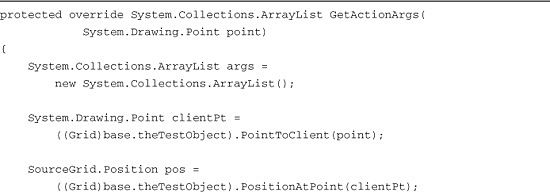


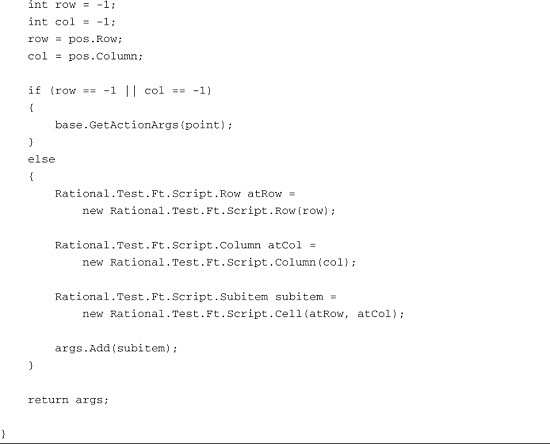


Overriding this method relies on you gathering information from the mouse event. To do this, you instantiate the IMouseEventInfo mouseevent interface and store the mouse action’s event information in it. You then take mouseevent’s X and Y coordinates to create the point—called firstPoint—where you click. This point is passed into the GetActionArgs() method for the purpose of capturing the actual object-based information derived from firstPoint. The object-based click arguments that get returned from GetActionArgs() are used along with the currentMethod variable (set to click) and preDownstate variable (containing the originating script’s command flags) to create a new method specification for Rational Functional Tester, telling it how to use the information your overridden method collected.

The method that actually specifies the object-based click action is the GetActionArgs() method (see [Listing 13.21](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex21)). You need to override this to specify the object-based action arguments you want for your SourceGrid control. This is the row index and column index for which cell in the grid to click on.

**Listing 13.21** GetActionArgs()





Overriding the GetActionArgs() methods essentially translates the point where Rational Functional Tester is clicking while recording. The first thing you need to do is create a new ArrayList, called args. This is used to store object-oriented information for the Click() method. After this is created, you can go about the business of translating the original click point to a row index and column index.

The original click point is passed to GetActionArgs() as an argument. You can use the native SourceGrid Grid control to get some more interesting data. You need to cast the reference to your TestObject (for example, base.theTestObject) as a SourceGrid Grid class. This enables you access to the PointToClient() method. Passing the point argument to this method enables you to capture the click point relative to the grid control itself. In other words, instead of having a point relative to the screen, you have a point relative to the actual SourceGrid control. This new point, called clientPt, is used to capture and return a SourceGrid Position structure called pos. This is where the translation begins. The pos object knows what row and column in the SourceGrid control relates back to clientPt. You can simply call pos.Row and pos.Column to get the integer values that represent a cell in the grid. This is a critical thing to understand. You used the SourceGrid Grid class methods to take the original point from the Rational Functional Tester recorder, turn it into a point, relative to the grid, and then turn that into the row and column indices that represent a cell in the grid.

The last part of this method is simple. You need to create the following three subitems:

• Rational.Test.Ft.Script.Row

• Rational.Test.Ft.Script.Column

• Rational.Test.Ft.Script.Cell

The Row subitem represents the row index in a cell. The Column subitem represents the column portion of the cell. The Cell subitem is the combination of the Row and Column. This is what gets passed back to the ProcessPreDownMouseEvent() as the click arguments.

The result of overriding these three record methods is the following, newly recorded click statement:

image

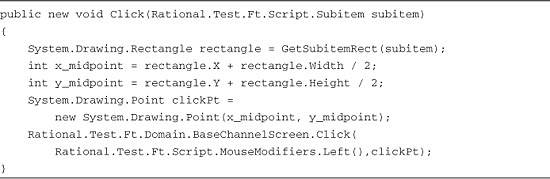
A more industrious version of this method would add the name of the column and perhaps text that is in the cell. This is something that can be addressed by tweaking the GetActionArgs() method to get more data out of the SourceGrid control.

**Playback Methods**

Overriding the methods for telling Rational Functional Tester how to record an object-oriented click is only half of the equation. You need to tell Rational Functional Tester how to play back the Click() action using your newly created object-oriented information. There are a few implementations for handling this. You traverse the path that creates a custom implementation of the Click() method and override the GetSubitemRect() method.

Your Click() implementation is somewhat of an inverse to the record methods you overrode. It takes object-oriented information and turns it into a point to click in the SourceGrid Grid (see [Listing 13.22](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex22)).

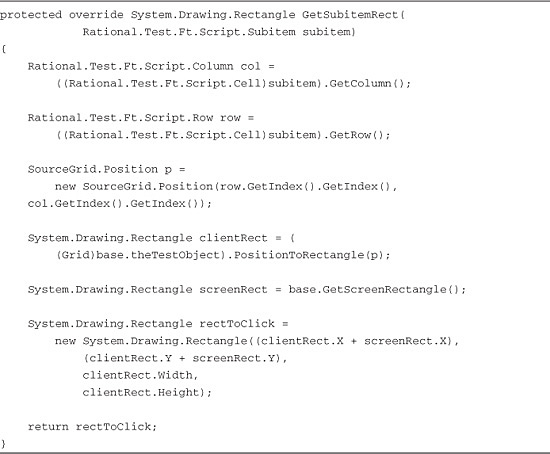
**Listing 13.22** Click()



The first thing your Click() method does is take the object-oriented information, passed in as the subitem argument, and passes it to the GetSubitemRect() method. This returns a System.Drawing.Rectangle class called rectangle. You then calculate the X and Y midpoints of rectangle. These are used to create a new System.Drawing.Point class called clickPt. You have essentially calculated the center point of the cell you want to click, derived from the object-oriented information contained in subitem. The final thing you do is perform the click action, stating that it should be a left click at the point defined by clickPt.

The actual conversion of the object-oriented information to System.Drawing.Rectangle is handled via the GetSubitemRect() method. You need to override the out-of-the-box proxy’s implementation of this method to make it understand how to convert the information you collected in the recording methods of your proxy (see [Listing 13.23](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex23)).

**Listing 13.23** GetSubitemRect()



As stated previously, you perform the inverse of what you did with the recording methods. You take the argument subitem and parse out the children that it contains. If you recall, you created Row and Column subitems in your overridden GetActionArgs() method. You just pull them out here and use them to calculate their position in the SourceGrid control. You do this by creating a SourceGrid.Position class called p, passing in the integer values contained in the Row and Column subitems. You cast your TestObject reference again to a SourceGrid Grid class. This enables you to call the PositionToRectangle() method, passing p in as the argument. The return value of this method is a System.Drawing.Rectangle class that you named clientRect. This represents the cell’s area on the grid control. You are not quite done yet. You want to pass back a rectangle that represents the cell’s area, relative to the screen. You call the base proxy’s GetScreenRectangle() method to obtain the screen’s rectangle, called screenRect. You then create a third System.Drawing.Rectangle called rectToClick, which is instantiated with the fusion of screenRect and clientRect. You finally return rectToClick back to the invoking Click() method.

**At This Point**

A good way to view this process, be it done in Java or C#, is that you need to undo what you do. You saw that first you need to define the object-oriented information for Rational Functional Tester’s recorder to understand how to click within a custom control. This involved converting the point that was clicked into object-oriented information—a text string, table cell, and so on. You then handled the reverse of that. You took the object-oriented information and converted it back into the necessary point to click. You can view this as pseudo-template for adding more object-oriented click() actions to future proxies.

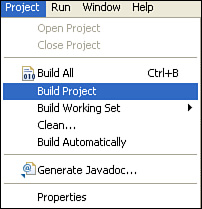
**Building the Proxy**

You need to build and export your proxy after you finish coding it. This results in a .jar file, if your proxy is written in Java or a .dll file, if it was developed in C#. This file is the final product that Rational Functional Tester uses when interacting with the control for which you build the proxy.

**Building the Java Proxy**

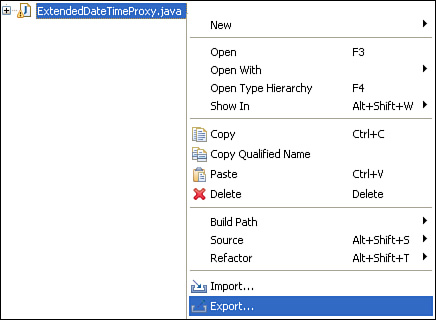
Building your proxy in the Eclipse environment is as simple as selecting **Project** > **Build Project** from the menu (see [Figure 13.39](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig39)).

**Figure 13.39** Build project



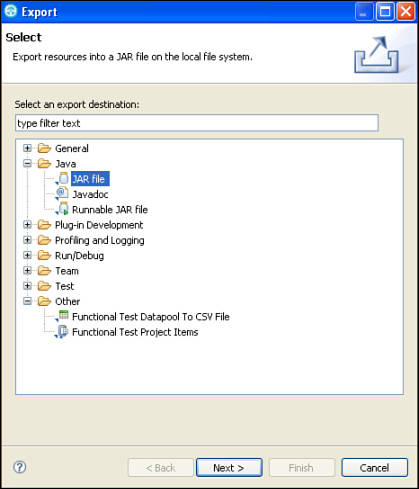
It shouldn’t take long to build your project. The process lets you know if any errors exist in your proxy code. After you have completed building your proxy project, you can export that class into a .jar file. You can right-click on your class file and choose **Export** from the right-click menu. This engages the Export Wizard (see [Figure 13.40](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig40)).

**Figure 13.40** Begin export process



The first page of the wizard asks you to select the export destination. You first need to expand the Java folder. This exposes the list of export options it contains. You want to select the **JAR file** option (see [Figure 13.41](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig41)).

**Figure 13.41** Java folder—JAR file option



The next page of the wizard, the JAR File Specification (see [Figure 13.43](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig43)), shows you what you are exporting. Feel free to expand the selected project in the left side of the Select resources to export list box until you see the package that you created. Selecting the package shows you which class you are exporting (see [Figure 13.42](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig42)).

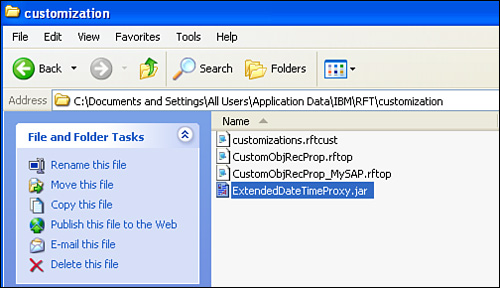
**Figure 13.42** JAR file specification



You need to provide a directory path and file name in the JAR file text field. After doing so, you can click the **Finish** button. This generates the .jar file you specified.

The directory where the resulting file is created is arbitrary. After the .jar file is created, you need to deploy it to the Rational Functional Tester customization directory—C:\Documents and Settings\All Users\Application Data\IBM\RFT\customization for typical installations of Rational Functional Tester (see [Figure 13.43](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig43)).

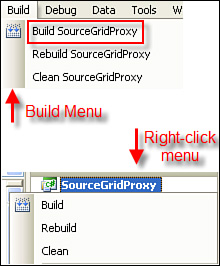
**Figure 13.43** Completed proxy—.jar file in Rational Functional Tester customization directory



**Building the C# Proxy**

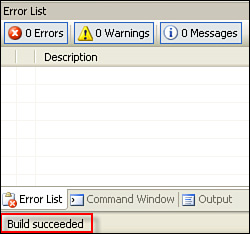
The process to build your proxy is simple. You can either choose **Build** > **Build SourceGridProxy** (or the name of any proxy) or right-click on your project and choose **Build** (see [Figure 13.44](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig44)).

**Figure 13.44** Build options



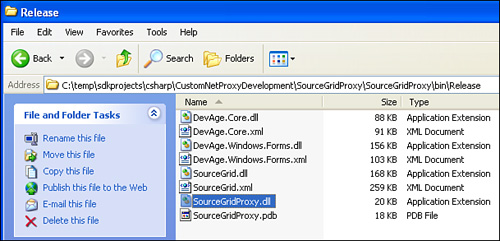
It takes a short amount of time for your proxy to build. You are informed of any errors. You are required to correct all errors prior to your proxy being built. If you do not receive any error messages, your proxy is built. You should see a success message at the bottom of Visual Studio, and you end up with a .dll file (see [Figure 13.45](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig45)).

**Figure 13.45** Build succeeded



If your build is a success, you can enter into your project’s bin directory. The proxy .dll file is in the Release directory (unless you specify a debug build). You should see all the supporting .dll files that you referenced in your project (see [Figure 13.46](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig46)).

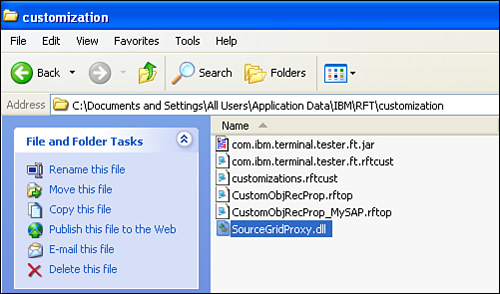
**Figure 13.46** Build Proxy—SourceGridProxy.dll



Notice that the name of the .dll file comes from the name of your project, SourceGridProxy, not the SourceGridControlProxy class. This is the difference between building a C# proxy and building a Java proxy (which exports the class file as a .jar).

You can move the SourceGridProxy.dll to the Rational Functional Tester customization directory. This is C:\Documents and Settings\All Users\Application Data\IBM\RFT\customization for typical installations of Rational Functional Tester (see [Figure 13.47](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig47)).

**Figure 13.47** Completed Proxy—.dll file in Rational Functional Tester Customization directory



**At This Point**

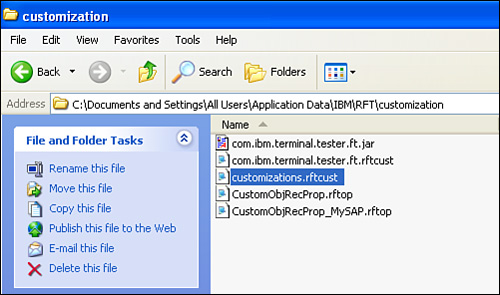
This completes the process for creating a custom proxy. If you have made it to this point without any errors, your proxy should be loaded and engaged by Rational Functional Tester. There is one final step, covered in the next section, to make this happen.

**Mapping the Proxy**

You need to create a mapping file for your proxy after you complete the custom coding and building of it. This involves creating an XML file, specifying information about your proxy and the control it was built for. The end result is a file that shares the same name as your proxy file but has a .rftcust extension.

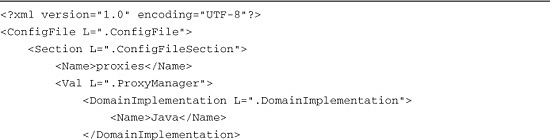
You do not have to create this file from scratch. If you browse to the Rational Functional Tester customization directory, C:\Documents and Settings\All Users\Application Data\IBM\RFT\customization in most install instances, you find a file called customizations.rftcust. This is the starting point of your mapping file (see [Figure 13.48](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig48)).

**Figure 13.48** Rational Functional Tester customization directory—customizations.rftcust



The XML shown in [Listing 13.24](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex24) contains the skeletal XML tags that you can use to get started. It is broken down into multiple sections.

**Listing 13.24** customizations.rftcust





This chapter focuses only on the first section. This is where you list your proxy and the control it supports. It is broken down into subsections. These are identified by the <DomainImplementation> tags. They define the different technology domains with which Rational Functional Tester works. By default, only Java and HTML are listed. You use this framework to initialize your own file. There are four steps for creating your initial mapping file:

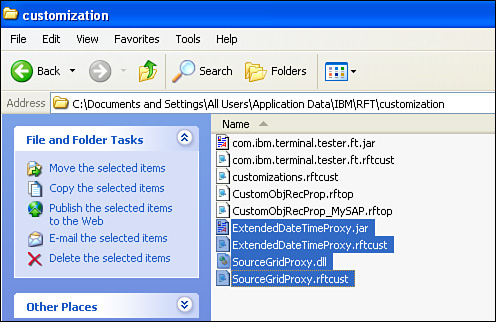
**1.** Create a new text file in the Rational Functional Tester customization directory, provide the same name as your .jar or .dll, and give it an extension of .rftcust. Continuing with the DateTime proxy example, your file would be ExtendedDateTimeProxy.rftcust. Continuing with the SourceGrid Grid example, your file would be SourceGridProxy.rftcust.

**2.** Open both the new file and the customizations.rftcust file. You can use your favorite text editor for this.

**3.** Copy the contents of the customizations.rftcust file into your newly created .rftcust file.

**4.** Save your .rftcust file (close customizations.rftcust because you are done with it). After completing these four steps, you should end up with a new .rftcust file that maps your proxy to the Rational Functional Tester’s recorder (see [Figure 13.49](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig49)).

**Figure 13.49** Rational Functional Tester customization directory—ExtendedDateTimeProxy.rftcust



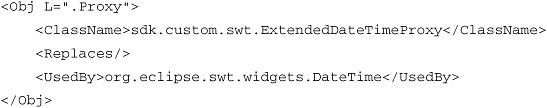
The last thing that you need to do is edit the mapping file to tell Rational Functional Tester to use your custom proxy for the specific control for which you built it. This requires you to make a couple of small edits. These changes only take place in the first section, which is the one that covers the domains. It was mentioned previously that the original customization.rftcust file only contained the Java and HTML domains. If you need to add a new domain, for instance, in .NET, you supply the following lines:

Image

These three lines enable you to list proxies that you have developed for the .NET technology platform, such as the SourceGridproxy. You can place these lines anywhere in the domain section. The location doesn’t matter. They come before the Java domain lines, after them, or even after the HTML domain lines. If you want to get a better feel for the structure of domains and the proxies that are listed for them, you can look at the rational\_ft.rftcust file. This is found in Rational Functional Tester’s bin directory. It shows you the different domains that are supported, listing the base proxies built that support different controls out of the box. You extend these base proxies when you create your own proxies. You need to list your proxy in the appropriate domain.

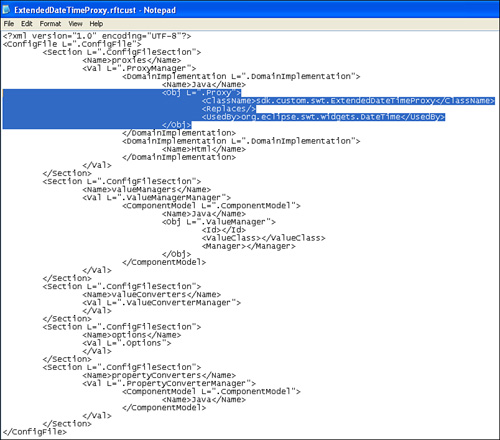
**Mapping for DateTime Proxy**

The DateTime proxy needs to be listed in the Java domain. This is accomplished by providing the XML that defines a proxy object. You need to specify the classname of the proxy and the classname of the control by which your proxy is used. Looking at the example for the DateTime class, you see:



The five lines in the preceding code listing are what wire your proxy to Rational Functional Tester’s recording engine. [Figure 13.50](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig50) shows you what these five lines look like in the context of the Java domain.

**Figure 13.50** Contents of ExtendedDateTimeProxy.rftcust

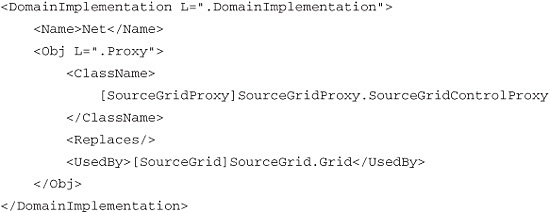


Optional Considerations for the .NET DomainImplementation Section

Note that the assembly prefix for the <UsedBy> element entries is not required. It is optional. This is due to the fact that the control classnames are unique in the world and, therefore, do not need to be prefixed by the name of the .dll files (for example, assemblies) they live in. In the off chance that there are multiple custom controls with the same classname, the assemblies can be prepended to delineate between them.

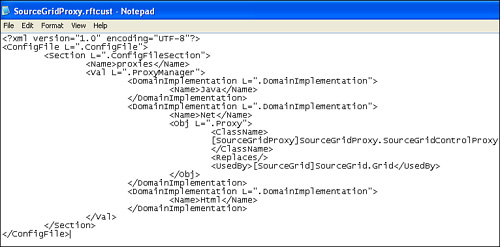
**Mapping for SourceGridProxy**

The SourceGrid proxy needs to be listed in the Net domain. You need to add the appropriate lines (mentioned previously) and then add the XML that defines your proxy object. Similar to the DateTime example, you need to specify the classname of the proxy and the classname of the control that uses the proxy. One slight variation exists. You specify the name of the assemblies (for example, .dll files) that contain key class files (for example, custom control and your proxy classes). This is accomplished by prepending the <ClassName> and <UsedBy> element entries with the assembly name, surrounded by square brackets. The end result of all of this is the following XML:



These lines tell Rational Functional Tester to use your proxy when interacting with the custom control you built it for. The full SourceGrid.rftcust file is displayed in [Figure 13.51](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig51).

**Figure 13.51** Contents of SourceGridProxy.rftcust

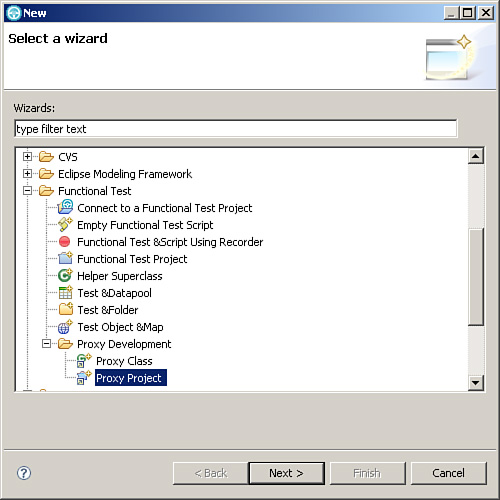


The creation of the .rftcust mapping file is the last step in creating a custom proxy. Whenever you launch rational Functional Tester, it looks in C:\Documents and Settings\All Users\Application Data\IBM\RFT\customization to see if there are any custom proxies to load. It sees your .rftcust file, opens it, and reads the XML, telling it to use the proxy you created. An important thing to note is that after you have your .jar/.dll and .rftcust in place, you need to close and reopen both Rational Functional Tester and the application you were automating for your mapping to take place.

**Taking an Easier Path**

Rational Functional Tester v8.1 introduces two wizards to facilitate proxy development. They are presently available only for the Eclipse flavor of Rational Functional Tester. The wizards enable you to create the base project structure and the skeleton for the code that you need to complete. Further, they create a template for the .rftcust file needed for mapping the proxy to Rational Functional Tester’s recorder. The first wizard you execute is for the project setup (see [Figure 13.52](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig52)). The second is for creating the skeletons for any proxy classes you need to build. Ultimately, if you are using Java to create a proxy, you still need to follow the steps in the prior sections in this chapter *minus* the steps in the “[Creating a Proxy Project](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13sec1lev2)” section in this chapter.

**Figure 13.52** New Proxy Project wizard



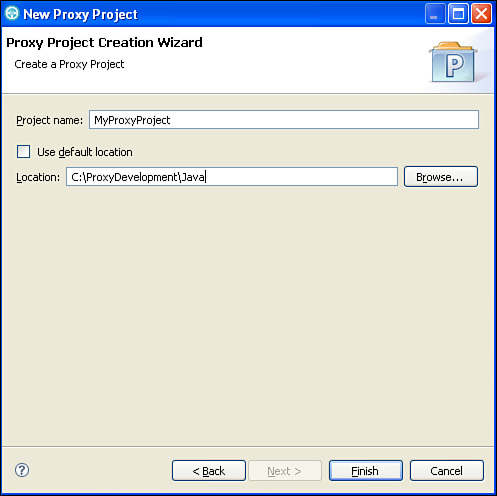
The first step to creating proxies using the wizards is to create the project itself. You initiate the wizard with the following steps:

**1.** Select **File** > **New** > **Other** (this opens the New dialog).

**2.** On this dialog, simply select **Functional Test** > **Proxy Project** (shown in [Figure 13.52](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig52)).

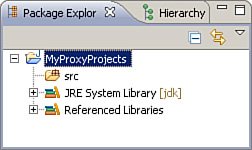
These steps invoke the basic New Proxy Project wizard. You have to provide only a project name here. Optionally, you can provide a new location in which to create the project. By default, the project is created in your workspace (see [Figure 13.53](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig53)).

**Figure 13.53** New Proxy Project Wizard—name and location provided



The end result of providing this data and clicking the **Finish** button is a new project, ready for your proxy development. You need to access the Java Perspective’s Package Explorer View to see your project (see [Figure 13.54](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig54)).

**Figure 13.54** Java Perspective’s Package Explorer View—New Proxy Project



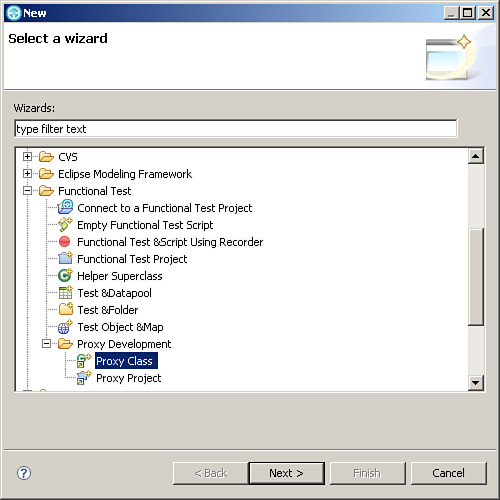
If you expand the src folder in your project structure, you find that a .rftcust file is created for you. This gives you the template needed to create the mapping discussed in the previous section, “[Mapping the Proxy](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13sec1lev8).” You can come here to fill in the necessary data for any proxy class that you develop in this project.

You are able to engage the second wizard when you create a new proxy project. Follow these steps:

**1.** Select **File** > **New** > **Other** (this opens the New dialog box).

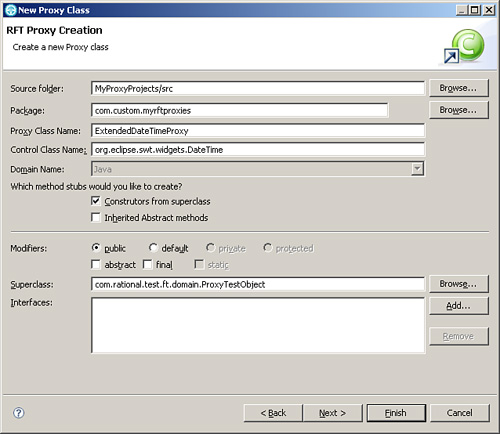
**2.** In this dialog box, simply select **Functional Test** > **Proxy Class** (shown in [Figure 13.55](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig55)).

**Figure 13.55** New Proxy Class wizard



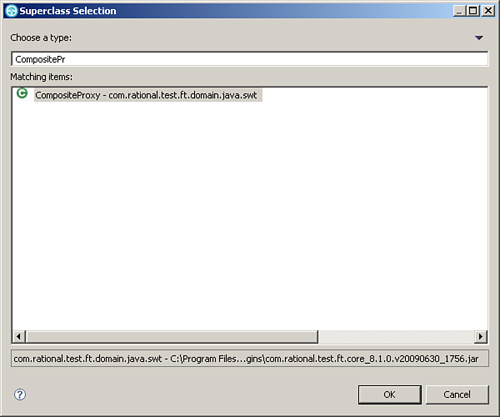
**3.** You need to fill out the **Source folder**, **Package**, **Proxy Class Name**, and **Control Class Name** fields. The Proxy Class Name is what you call your proxy. The Control Class Name is the actual control that you want to build the proxy for (for example, in this example—org.eclipse.swt.widgets.DateTime). See [Figure 13.56](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig56) for an example of these fields filled out. You can click the **Browse** button and select an existing package or type in the name of a new package (for example, com.custom.myrftproxies) to store your proxy class in.

**Figure 13.56** New Proxy Class Wizard—fields filled in



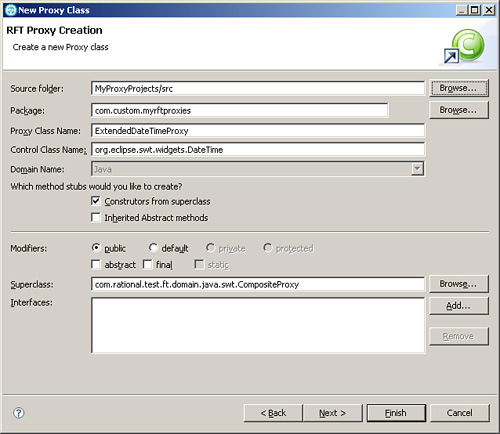
**4.** The next thing that you need to do is extend the proxy class that you researched earlier. For example, the SWT example from the “[Figuring Out Which Proxy to Extend](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13sec1lev1)” section in this chapter is the CompositeProxy. Extending this class is done by clicking the **Browse** button for the Superclass field. This launches the Superclass Selection dialog box. You can find the proxy class you want to extend by typing the name of it in the Choose a type field (see [Figure 13.57](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig57)).

**Figure 13.57** New Proxy Class Wizard—extending the proxy class



**5.** Verify that the Superclass field is populated with the proxy class you selected in Step 4. This is shown in [Figure 13.58](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig58).

**Figure 13.58** New Proxy Class Wizard—Superclass field populated



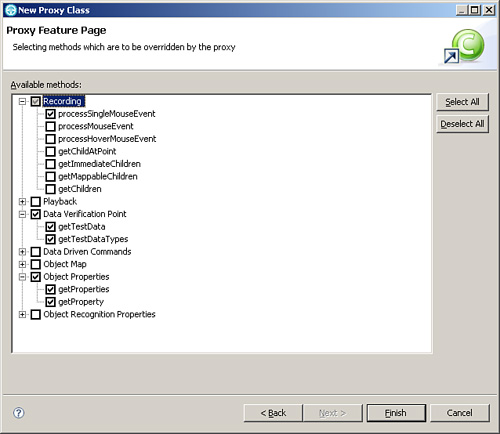
**6.** Click the wizard’s **Next** button. This takes you to the Proxy Feature Page.

**7.** You need to select the methods you want to override.

Note

You might not see every method you need. For example, the CompositeProxy class provides only a subset of the methods that you need to override (see [Figure 13.59](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig59)).

**Figure 13.59** New Proxy Class Wizard—extending the proxy class



The previous sections regarding the Java SWT proxy example showed that you need to have the following methods: processSingleMouseEvent(), getTestData(), getTestDataTypes(), getProperties(), getProperty(), getDescriptiveName(), getRole(), getTestObjectClassName(), and getActionArgs(). [Figure 13.59](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig59) shows that the methods to override for the CompositeProxy class are only a subset of what is needed (for example, only the first five methods, in the preceding list, are available). Therefore, you still need to include the other methods in your proxy class, using the means that were detailed in the previous sections.

**8.** After selecting the methods to override, click the **Finish** button. This completes the New Proxy Class Wizard and creates the skeleton of the proxy you need. This is displayed in [Figure 13.60](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig60).

**Figure 13.60** New Proxy Class Wizard—completed proxy class skeleton



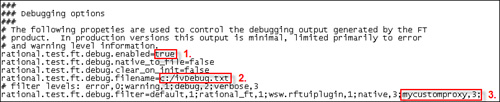
As you can see, the wizards help to expedite proxy development. They provide you with the pieces to quickly get you started in your efforts. However, as nice as the wizards are, you still need to fill in most of the coding—the content that accomplishes the actual work.

**Debugging a Proxy**

You most likely need to debug your proxy at least once. Rational Functional Tester provides a means to do this through a debug log. To use this log for your purposes, you need to specify a few options in the ivory.properties file.

The ivory.properties file is found in the ..\FunctionalTester\bin directory. Opening it reveals a myriad of options. You want to focus on the Debugging Options section (see [Figure 13.61](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig61)).

**Figure 13.61** The ivory.properties file—debugging options



The numbered boxes highlight the three properties with which you need to work. First, you want to ensure that the debugging is actually enabled. Therefore, you need to make sure that the rational.test.ft.debug.enabled property is set to a value of true. Secondly, you control what to name your debug log and where to write it. You can do this by setting the rational.test.ft.debug.filename property to your desired value. By default, it is set to c:/ivDebug.txt. Finally, you want to make sure that you can easily identify your proxy messages. You can add in a custom filter for your proxy by adding it to the rational.test.ft.debug.filter property. [Figure 13.61](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig61) shows this in the third box, highlighting the mycustomproxy filter. The number 3, that follows the filter, tells Rational Functional Tester to log all the messages coded into the proxy, not just the warnings and errors it encounters. After making the appropriate edits to the ivory.properties file, you can save the changes and close it. You have now enabled logging for your proxy.

To write messages from your proxy to the log, you first need to add the following line to the beginning of your proxy class:

protected static FtDebug debug = new FtDebug("mycustomproxy");

This line is the same whether or not you use C# or Java to create your proxy. However, you need to ensure that you have imported the appropriate package into your proxy class. Using Java, your import looks like the following:

import com.rational.test.ft.util.FtDebug;

If you create a proxy in C#, you need:

using Rational.Test.Ft.Util;

Essentially, you create a new FtDebug object that writes your desired messages to the mycustomproxy filter. You write to the log using one of the following messages types:

• trace

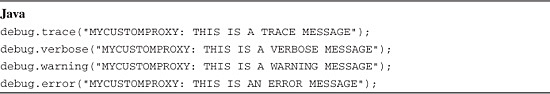
• verbose

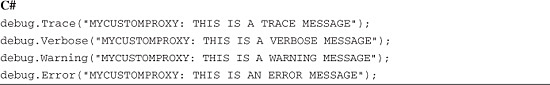
• warning

• error

It behooves you to prefix any of your messages with some sort of identifier. For instance, you might use the name of your proxy filter (see [Listing 13.25](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13ex25)).

**Listing 13.25** Debug messages

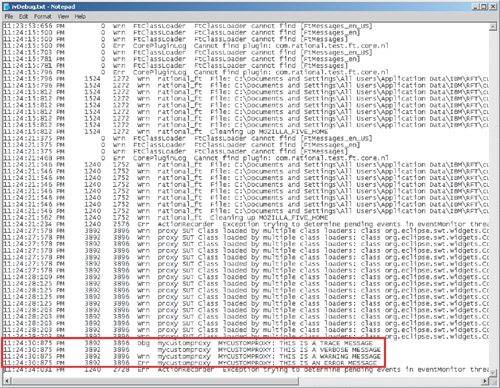




A lot of information outside of the proxy messages is recorded in the log file. Using a prefix, in conjunction with the filter name, helps you quickly locate your proxy messages. This helps you find and solve your issues faster.

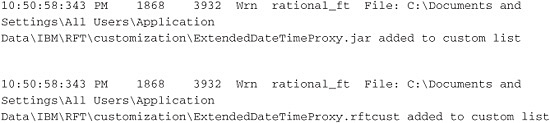
As you can see in [Figure 13.62](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13fig62), the custom proxy messages are identified by their filter name and prefix (contained in the red outline). Depending upon the type of debug message you are writing, a code might be associated with it. For instance, a trace message has a code of Dbg, a warning message has a code of Wrn, and an error message has a code of Err. A verbose message does not have a code.

**Figure 13.62** ivDebug.txt file



You can use these to help you identify the types of messages you want to place in your proxy code. You can place error messages in catch statements, use verbose messages to identify the start and completion of methods, and so on. The idea is to thread your messages throughout your code to have the most robust means to debug your proxy issues.

If you don’t see your debug messages in the log file, Rational Functional Tester might not have successfully loaded your proxy. To check for this, you want to look for lines that are similar to the following:



These two lines show that the .jar and .rftcust files are engaged by Rational Functional Tester. If you do not see these, you need to check your .rftcust file for syntax errors. It is not uncommon to misspell the name of your class. You also want to make sure the name of your .rftcust file matches the name of your .jar file. In the previous example, they are both called ExtendedDateTimeProxy. If they do not match or you have a typo inside of your .rftcust file, Rational Functional Tester does not engage your proxy.

**Summary**

This chapter took a high-level view of Rational Functional Tester’s Proxy SDK. It provided samples to give you an idea of what is involved with creating your own custom proxies and the tools available in Rational Functional Tester to help you build proxies. You should have a feel for what level of effort is involved with this type of project. You should also assume that your project shares some similarities with the examples here but also differs in many aspects. Your proxy development is dictated by the GUI control(s) you automate.

[Prev](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch12.html)

[Chapter 12. Extending Rational Functional Tester with External Libraries](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch12.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch14.html)

[Chapter 14. Developing Scripts in the VB.NET Environment](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch14.html)

**Welcome to Safari**. Remember, your free trial will end on March 7, 2015, but you can [subscribe at any time](https://www.safaribooksonline.com/subscribe/)

* [Recommended](https://www.safaribooksonline.com/r/)
* [Queue](https://www.safaribooksonline.com/s/)
* [Recent](https://www.safaribooksonline.com/recent/)
* [Topics](https://www.safaribooksonline.com/t/)
* [Settings](https://www.safaribooksonline.com/u/)
* [Blog](http://blog.safaribooksonline.com/)
* [Support](http://support.safaribooksonline.com/?prod=flow)
* [Support](http://msupport.safaribooksonline.com/?prod=flow)
* [Feedback](mailto:feedback@safaribooksonline.com)
* [Sign Out](https://www.safaribooksonline.com/accounts/logout/)

© 2015 [Safari](http://www.safaribooksonline.com/).   
[Terms of Service](https://www.safaribooksonline.com/terms/) / [Membership Agreement](https://www.safaribooksonline.com/membership-agreement/) / [Privacy Policy](https://www.safaribooksonline.com/privacy/)

[**Make font larger Make font smaller**](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html)