Effective test automation techniques for Rational Functional Tester

Level: Intermediate

Michael Kelly (mike@michaeldkelly.com), Independent Consultant, MichaelDKelly.com

19 Jun 2007

Learn tips for dealing with common problems with browsers, verification points, low-level commands, the script helper superclasss, and more.

If you're a regular user of test automation tools, then you are probably familiar with the concept of test automation frameworks. Testers frequently ask for framework recommendations, references, and solutions, but frameworks are only half of what you need to consider. How you structure your test code to best facilitate testing of the applications that you're testing is the *second* step toward effective automation.

This article focuses on the *first* step, which is understanding how to effectively use the tool that you have. This step encompasses several topics:

- Objects and properties
- Common problems with browsers
- Verification points
- Low-level commands
- The script helper superclass

For each of those topics, you'll find links to additional information at the end of this article under Resources.

Note:
The author used this software to write this article:

- IBM® Rational® Functional Tester Version 7.0.0
- Microsoft® Internet Explorer® Version 6.0.2900.2180, SP2
- Microsoft® Windows® XP Professional, SP2

Alternatives to finding objects and their properties

Components such as dialog boxes, command buttons, and labels have associated pieces of information called properties. Properties have names and values. You often need to get to the various properties of the objects that you are testing so that you can perform a verification of some sort, or so that you can programmatically determine what the test script should do next. This section explains objects, properties, and ways that you can interact with them in your scripts.

Querying and setting values of object properties

Have you ever wanted to compare previous versions of a value to the current value dynamically at runtime? Or have you ever wanted to add a branch in your Rational Functional Tester script that is based on the current value of a property contained in an object? You can retrieve the value of a property programmatically by calling the `getProperty` method.

The example in code Listing 1 uses the `getProperty` method to determine whether a label contains a success message. If it does, the **OK** button will be clicked. If it doesn't, the **Cancel** button will be clicked.

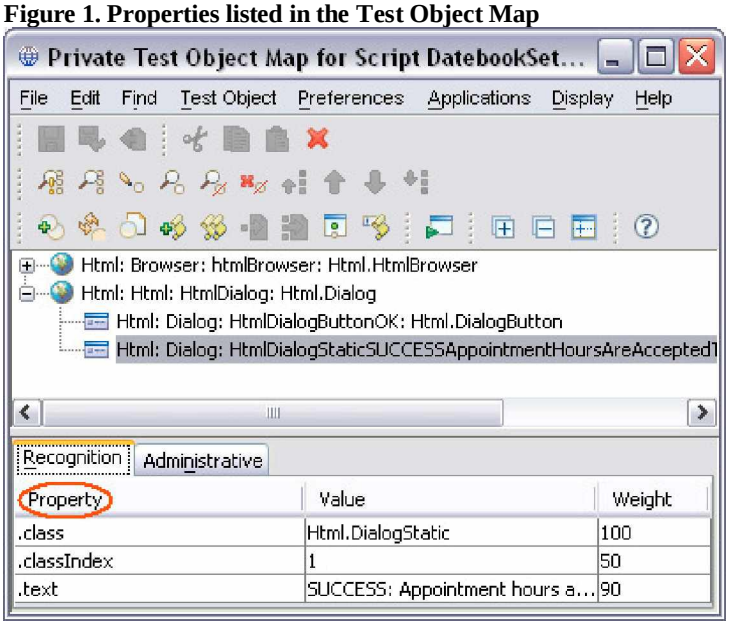
Listing 1: Using the getProperty method

```
if("SUCCESS".equals(dialog_htmlDialogStatic().
getProperty(".text")))
{
dialog_htmlDialogButtonOK().click();
}
else
{
```

```
dialog_htmlDialogButtonCancel().click();
}
```

If you need to figure out what properties an object has, you can open the **Test Object Map** and look at the properties available there. (See Figure 1.)

Figure 1. Properties listed in the Test Object Map



You can also look at available properties by temporarily recording an Object Properties verification point (VP) or by inserting the commands to extract a property value into a variable through the **VP and Actions** wizard.

Note:
Rational Functional Tester also supports a setProperty method. However, that method comes with a disclaimer: Do not use it unless you are sure of the result. This is because it calls internal methods that may violate the integrity of the application that you're testing.

Now that you know how to get properties from your objects, you might be wondering: "How do I find the object in the first place?" Well, that's good question.

Ways to search for TestObjects

The basic building block of your Rational Functional Tester script is the ubiquitous **`TestObject`** . A `TestObject` represents a connection point between the script that is playing back and the application that you are testing. For every `TestObject` in a generated test script, a corresponding object exists in the application that you recorded against, and that object now also exists in the **Test Object Map**.

Most likely, you normally interact with a `TestObject` by using the object map. However, Rational Functional Tester also supports a means for locating a `TestObject` programmatically. The search is based on name and value pairs representing properties of the `TestObject` or `TestObjects` that you are looking for. The search can be either global or limited to children of a parent `TestObject`.

Rational Functional Tester uses an object called the **`RootTestObject`** to represent a global view of the software under test.

- If you want to search the entire application, you invoke the find method on the `RootTestObject`.
- If you want to search a specific object, then simply invoke find on that `TestObject`. Searching a specific `TestObject` will search only the children of that `TestObject`

Mark Nowacki and Lisa Nodwell write about how to better understand and use the `TestObject.find` method (see the second listing in Resources). I've spared you my code samples so that you can see theirs, instead. They do a great job of explaining and illustrating the principles through practical examples.

Solving ambiguous recognition of objects

Sometimes, Rational Functional Tester can't distinguish between two objects that are alike during playback. It's common for this to occur when two browsers or instances of the same application are open simultaneously. When two Web browsers are open, Rational Functional Tester tries to resolve the ambiguity by using an anchor to help determine the target object. Now that you know how to use the find method, you can identify the browser and application instances by using a `TestObject` reference.

Running more than one instance of an application simultaneously during playback will result in ambiguity about the target of commands, such as `object()` or `click()`. To resolve this, you can use `ProcessTestObject` when you call the `startApp` command. In the Listing 2 example, the `ProcessTestObject` functions as an anchor to locate the desired application.

Listing 2: Using the ProcessTestObject to anchor the application

```
ProcessTestObject pto1 = startApp("ApplicationOne");
ProcessTestObject pto2 = startApp("ApplicationTwo");
object(pto1, DEFAULT).click();
```

Clean house by deleting registration of test objects

The final aspect of using the find method deals with deleting registration of the object. `TestObject.find` returns a new `TestObject` reference (sometimes called a **bound reference**, a *found reference*, or a *non-mapped reference*). These references retain access to the object until you explicitly "unregister" them, which means that they can cause a few problems if you don't take care of them.

Rational Functional Tester unregisters bound references only when the entire playback ends, not when the script ends. As long as a bound reference to the object exists, Rational Functional Tester may prevent the object in the application from being entirely free. For example, while you hold a bound reference to a Java™ object, the Java object is not treated as no longer necessary, thus deleted. Therefore, it's a good idea to explicitly unregister any bound references that you create as soon as you don't need them anymore.

`RationalTestScript` contains several methods that remove references to `TestObjects`:

- `com.rational.test.ft.script`
- `RationalTestScript.unregister`
- `unregisterAll`

Caution:
When dealing directly with `TestObjects`, using them may create instability in your application. Unregister your `TestObjects` as soon as possible.

Ways to solve common problems with browsers

Here are some of the common problems that you may encounter when testing HTML applications and how to deal with them.

Fix funky behavior with the .readystate method

If your browser or the objects within the browser aren't in the right state, you can get some funky (strange, erratic) behaviors when you interact. with them -- especially if you do a lot of interaction outside of the helper methods. That's why checking the Web browser or an object's readyState is a good habit to get into.

readyState values in Rational Functional Tester

Value	State	Description
0	uninitializedObject	Object is not initialized with data
1	loadingObject	Object is loading data
2	loadedObject	Object has finished loading data
3	interactiveUser	User can interact with the object, even though it is not fully loaded yet
4	completeObject	Object is completely initialized

If you know that you're going to be interacting with a large table, tree (file directory), or HTML document, then it's a good idea to check the state (Listing 3) before starting with your interaction.

Listing 3. Checking the readyState

```
while
(Integer.parseInt(object.getProperty(".readyState").toString())
< 4)
{
sleep(2);
}
```

Improve test execution by using the `waitForExistence` method

You may also have noticed that when you play back your test scripts, the browser is still starting while the playback window is waiting for the first command to run. This is because newer browsers are slow, perhaps because of the extra items, such as tabs, that now have to load. Therefore, it's a good habit to use a `waitForExistence` method when you open a browser. One way to do this is to use a **Wait for Selected TestObject** verification point:

1. When recording, start the application.
2. Click the **Insert Verification Point or Action Command** button on the **Recording** toolbar.
3. On the **Select an Object** page of the **Verification Point and Action Wizard**, click the **Object Finder** icon and drag it over the **HTML page** (not the browser itself).
4. Click **Next**.
5. On the **Select an Action** page of the **Verification Point and Action Wizard**, click the **Wait for Selected TestObject** option.
6. If necessary, clear **Use the defaults** to change the **Maximum Wait Time** and **Check Interval** settings, which are preset to 2 minutes and 2 seconds, respectively.
7. Click **Finish**.

However, you may find it easier just to add the call yourself after the `startApp` command. All you need to do is reference the `BrowserTestObject` and add the call to the `waitForExistence` method.

Handling unexpected active windows

Another common problem with browsers (and some Java applications) is the numerous popup windows. Unexpected active windows are especially common when your scripts run on different machines or with different browsers and browser configurations.

The Rational Functional Tester **Help** files provide a couple of good solutions (one simple, one not so simple), though. One solution is to use the classic try-and-catch (Listing 4), and wait for the message to appear. If it does not appear, you can continue.

Listing 4. Waiting for a popup dialog

```
try
{
// Dialog_HtmlDialogButtonOK().waitForExistence(5,1);
Dialog_HtmlDialogButtonOK().click();
}
catch(ObjectNotFoundException e)
{
}
```

Tip:
You can remove the commented line of code if you want to wait a specified amount of time.

Another solution, which is a bit more involved, is to implement a check similar to this in an onObjectNotFound exception. By putting the implementation in a helper superscript, you can handle the event for any Rational Functional Tester script that extends this helper superclass (more on that here later).

Consider not-so-common verification points

In addition to verification points specified during recording, you can also incorporate new verification points into a Rational Functional Tester script (and not only by using the `Insert at Cursor` command). Scripting manual and dynamic verification points enables you to specify data for comparison against an object that is not found in the Test Object Map.

Manual verification points

Manual verification points are useful when you create the data for the verification point yourself, and you want to compare the data. For example, the data could be the result of a calculation or could come from an external source, such as a spreadsheet, datapool, database, or custom calculation.

Manual verification point objects are constructed by using the `vpManual` method shown in Listing 5.

Listing 5. Using the vpManual method

```
vpManual
("ManualVP_01", "Check something manually.").performTest();
// or
vpManual
("ManualVP_01", "Check something manually.",
"Check something manually.").performTest();
```

The first time that a manual verification point is executed, if the verification point object doesn't exist, the value passed in will become the baseline value (which you can then see and edit in Rational Functional Tester, just like any other verification point). Your log file will say: Created verification point baseline.

From that point forward, any subsequent call to `vpManual` for that verification point will compare the actual point with the point. As Listing 5 shows, you can also call `vpManual` by passing in both the baseline and the actual result. In this case, the verification point object in Rational Functional Tester is ignored.

Dynamic verification points

Sometimes, you need to perform verification points on test objects that aren't in your object map. To do this, you can use dynamic verification points by using the `vpDynamic`method (Listing 6). Dynamic verification points raise the appropriate user interface the next time that the script is played back. You can insert verification point data tested against an object specified by the script. In this way, you avoid having to run the test manually to the appropriate state before recording the verification point.

If you pass in only the verification point name, the **Recording Verification Point and Action** wizard activates the next time that the script is played back. Using that wizard, you then specify the `TestObject` and the baseline data for subsequent runs to test against.

Listing 6. Using the vpDynamic method

```
vpDynamic
("DynamicVP_01").performTest();
// or
vpDynamic
("DynamicVP_01", TestObjectHere()).performTest();
```

Caution:
The second example shown in Listing 6 requires that you pass in the actual `TestObject`. Although the specified `TestObject` does not have to be from the Test Object Map, it must be the same object consistently for the results to be meaningful. So again, if you are using the find method for `TestObjects`, be careful.

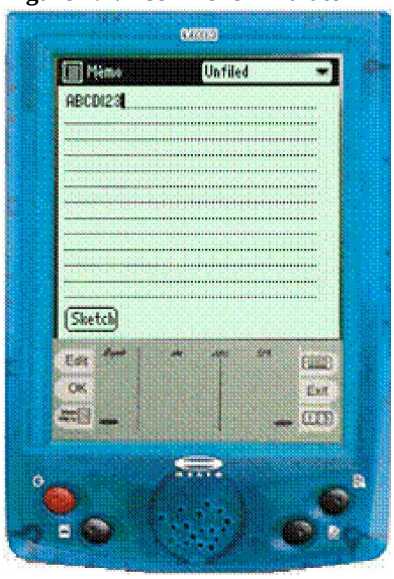
A common error when using these methods is to omit the `performTest` method. This is permissible and compiles without warning, but no interesting action occurs when the script runs.

Details about using low-level commands

Low-level playback emulates the exact mouse movements and keyboard actions that the user performs. This playback can allow for more control for certain user actions, such as drawing programs, PDA emulators, and other nontraditional object-compliant tools that just don't like simple `click` and `input` commands.

For example, suppose that you were testing handwriting recognition in the VTech Helio Emulator that Figure 2 shows.

Figure 2: VTech Helio Emulator



Traditional Rational Functional Tester scripts are powerless against applications such as the Helio Emulator. In fact, as I wrote in a previous article on low-level scripting using IBM® Rational® Robot (see Resources), about the only option that you have for applications like this is low-level recording. By going low-level, you can play back the individual components of a mouse click.

The `RootTestObject` class includes two low-level methods:

- emitLowLevelEvent(LowLevelEvent)
- emitLowEvent(LowLevelEvent[])

Factory methods on `SubitemFactory` for construction of LowLevelEvents include these methods:

- delay(int) **Note:** In milliseconds
- keyDown(string)
- keyUp(string)
- mouseMove(point)
- mouseWheel(int)
- leftMouseButtonDown()
- leftMouseButtonUp()
- rightMouseButtonDown()
- rightMouseButtonUp()
- middleMouseButtonDown()
- middleMouseButtonUp()

The script code in Listing 7 draws on the handwriting recognition pad for the Helio in Microsoft® Paint®.

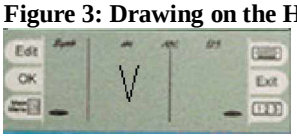
Listing 7. Using low-level scripting in Rational Functional Tester

```
afx10000008window().click(atPoint(100,100));
LowLevelEvent llEvents[] = new LowLevelEvent[7];
llEvents[0] = mouseMove(atPoint(100,100));
llEvents[1] = leftMouseButtonDown();
llEvents[2] = delay(250);
```

```
llEvents[3] = mouseMove(atPoint(105,120));
llEvents[4] = delay(250);
llEvents[5] = mouseMove(atPoint(110,100));
llEvents[6] = leftMouseButtonUp();
getRootTestObject().emitLowLevelEvent(llEvents);
```

The code in Listing 7 produces the letter **V** on the canvas, as Figure 3 shows.

Figure 3: Drawing on the Helio Emulator in Microsoft Paint



The drawback to such cosmic powers? To the best of my knowledge, unlike with Rational Robot, you need to write all of this code by hand. I don't know of a way to turn on a low-level recorder in Rational Functional Tester. But that's OK. If you were really testing something like Helio, you would create reusable methods for writing the letters anyway. So, most likely, you would need to figure all of them out only once.

Either way, it's a good feature to remember, because there are times when you just can't seem to get the click-drag-type combinations right while recording. In those cases, sometimes low-level playback is the answer.

How to enhance your scripts with the script helper superclass

If you have not read "Creating a super helper class in IBM Rational Functional Tester," by Dennis Schultz, jump down to Resources and do so now. That article is the single best resource that I've seen for understanding how the superclass works. Helper classes allow you to add functionality to your test scripts.

By default, all Rational Functional Tester scripts extend the `RationalTestScript` class, and thereby inherit a number of methods (such as `callScript`). Advanced testers may prefer to create their own helper superclasses to extend `RationalTestScript` and to add methods or override the methods from `RationalTestScript`.

You can specify a helper superclass that Rational Functional Tester will use whenever you create or record a script in your project. This default superclass is specified on the functional test **project properties** page. You can also specify a helper superclass for an individual script in the functional test **script properties** page. After a script has been created, it retains the reference to the default superclass as its own helper superclass.

Helper superclasses are useful in sharing functionality across multiple scripts. A super helper provides a single place for you to put the code that you want each script to access. Any code that you put there will be inherited by every helper class that extends your super helper.

Resources

Learn

- If you aren't yet comfortable using the debugging features in Rational Functional Tester, read "Use the debugging feature of IBM Rational Functional Tester 6.1 to ensure application quality," by Allen Stoker and Michael Kelly (developerWorks, June 2005).
- For more information on using the TestObject.find method, the author recommends "Using IBM Rational Functional Tester: Understanding and Using the TestObject.find Method," by Mark Nowacki and Lisa Nodwell (developerWorks, July 2006).
- For a look at low-level scripting in Rational Robot and for a more detailed example of using low-level scripting, check out "Introduction to Low-Level Scripting" (PDF) by the author of this article.
- If you have not yet experimented with making your own helper classes in Rational Functional Tester, read "Creating a super helper class in IBM Rational Functional Tester," by Dennis Schultz (developerWorks, December 2003).
- Get an Introduction to IBM Rational Functional Tester 6.1. Rational Functional Tester is an automated tool for testing Java, .NET, and Web-based applications. Get familiar with its features and capabilities here.

- Visit the Rational Functional Tester area on developerWorks for introductory to in-depth information.
- Visit the Rational software area on developerWorks for technical resources and best practices for Rational Software Delivery Platform products.
- Subscribe to the IBM developerWorks newsletter, a weekly update on the best of developerWorks tutorials, articles, downloads, community activities, webcasts and events.
- Subscribe to the developerWorks Rational zone newsletter. Keep up with developerWorks Rational content. Every other week, you'll receive updates on the latest technical resources and best practices for the Rational Software Delivery Platform.
- Subscribe to the Rational Edge for articles on the concepts behind effective software development.
- Browse the technology bookstore for books on these and other technical topics.

Get products and technologies

- Try Rational Functional Tester 7.0. The trial download is free but requires registration.
- Download IBM product evaluation versions and get your hands on application development tools and middleware products fromDB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Get involved in the developerWorks Functional and GUI Testing discussion forum. For users of Rational Functional Tester and for the discussion of general testing topics.
- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Mike Kelly is an independent consultant located in the Midwest of the United States. Mike also writes and speaks about topics in software testing. You can find most of his articles and his blog on his Web site, www.MichaelDKelly.com.