IBM PRESS

Software Test Engineering with IBM Rational Functional Tester

The Definitive Resource

Chip Davis, Daniel Chirillo, Daniel Gouveia, Fariz Saracevic, Jeffrey R. Bocarsly, Larry Quesada, Lee B. Thomas, and Marc van Lint

Foreword by Patrick L. Mancini

IBM Press Program Managers: Steven M. Stansel, Ellice Uffer
Cover design: IBM Corporation

Associate Publisher: Greg Wiegand
Marketing Manager: Kourtnaye Sturgeon
Publicist: Heather Fox
Acquisitions Editor: Chris Guzikowski
Development Editor: Sheri Cain
Managing Editor: Kristy Hart
Designer: Alan Clements
Project Editor: Anne Goebel
Copy Editor: Deadline Driven Publishing
Indexer: Brad Herriman
Compositor: Jake McFarland
Proofreader: Leslie Joseph
Manufacturing Buyer: Dan Uhrig

Published by Pearson plc
Publishing as IBM Press

Library of Congress Cataloging-in-Publication Data
Software test engineering with IBM Rational functional tester : the definitive resource / Chip Davis ... [et al.].
p. cm.
Includes index.
ISBN 978-0-13-700066-1 (pbk. : alk. paper) 1. IBM Rational functional tester. 2. Computer software—Testing. 3. Object-oriented methods (Computer science) 4. Software engineering. I. Davis, Chip, 1969-
QA76.76.T48S635 2009
005.1—dc22
2009030295

ISBN-13: 978-0-13-700066-1
ISBN-10: 0-13-700066-9

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing October 2009

Foreword

The American novelist and historian Edward Eggleston once wrote that “persistent people begin their success where others end in failure.” The tremendous accomplishment of writing this book is a perfect reflection of Eggleston’s perspective.

As a long-time practitioner in the automated tools landscape, it is my opinion that the competency and technical capabilities of the authors and the information they provide are *undisputed* in many ways. Having had the privilege of working closely with these individuals (and even on occasion serving as a mentor), I am wholly impressed with the talent they have demonstrated.

Perhaps more impressive, however, is their successful collaborative efforts over the course of many long months, notwithstanding the fact that each was heavily engaged with time-consuming corporate- or customer-facing projects. Having followed the group in their journey from a distance, there were times when the idea of publication seemed a distant goal. In addition to being a technical contributor, it must be said that Daniel Gouveia’s undying determination serviced the project in many exceptional ways; for that, we are most grateful.

This book serves as a unique and useful tool for quality practitioners of all levels who wish to learn, reference, build upon, and implement technical best-in-class automation techniques for functional tests. The real power of this book is the fact that much of the content originates from the needs of real users and real customer situations, where introductory and advanced concepts are defined, constructed, and implemented.

The advancements in automated testing have dramatically changed the landscape in terms of traditional tester roles of the past. Rational Functional Tester has updated, matured, and changed in ways that now attract a wide range of technical and nontechnical testers. In particular, this change presents tremendous opportunity for users who are in traditional business roles to now diversify and pursue the more challenging aspects of test automation. We welcome and embrace this technical and cultural shift in the hopes that other technical disciplines will eventually follow suit.

To the authors, I salute your persistence and determination and congratulate you in a job well done on a book well worth reading.

With much gratitude,

Patrick L. Mancini,
IBM WW Quality Management Practice Lead

Preface

Rational Functional Tester (RFT) is a tool for automated testing of software applications. It mimics the actions and assessments of a human tester. To understand the value of Rational Functional Tester, it is best to know some of the history of software testing.

Brief History of Software Testing

In the earliest days of computing, programs and their results were rigorously tested by hand. Computing resources were so expensive and scarce that there was no margin for error on the part of a programmer. Confirming that a program was correct required a great deal of manual effort.

As computing power became more readily available, the number of programs written and the complexity of programs increased rapidly. The result was that manual testing could no longer keep pace.

Furthermore, testing an application once is an interesting exercise, but testing that same application repeatedly through multiple bug fix cycles is boring and error-prone. Plus, the time to test applications in the early waterfall-style development lifecycle was limited (and was often cut further when the development phase could not deliver on time), adding pressure on the tester, which increased errors.

The only way to avoid these problems was to automate the testing effort. However, most user interfaces were not created with an automation-friendly interface as a requirement. The earliest UI automation testing tools were able to perform an action only at a coordinate-based location, and derive text for verification from a coordinate-based rectangle, leading to highly fragile automation code. This meant that applications could be tested automatically, but only if the application's UI did not change significantly. Unfortunately, application maintenance frequently required *both* business logic changes *and* UI changes. This meant recreating many if not all tests every time the application's UI changed.

With the rise of object-oriented programming, user interfaces began to be constructed from libraries of UI objects. These objects could accept actions in the form of method invocations, and they could return properties including text, font, color, and so on. Automated testing tools followed the development technologies, and were refashioned to operate on an object-recognition basis instead of a screen-coordinate basis. Tool assets (known in Rational Functional Tester as test scripts because they are similar to scripts given to actors in a stage play) could be reused from one build to the next, even when objects were moved within the UI.

However, several problems still remained. A developer might change an object so that the testing tool could no longer recognize it (for example, by changing the value of one of the object's critical properties), causing the script to fail. Or, a programmer might move an object to a different part of the UI hierarchy, where the testing tool might be unable to find it. Many applications were developed using custom objects (such as those for Enterprise Resource Planning [ERP] applications), and not those provided to developers by the operating system. In addition, more complex verification of text was required, particularly in the case where an application displayed data retrieved from or generated by a server. Also, assets such as test scripts were increasingly recognized as programs (in spite of recording capabilities and wizards that reduced the need to create or modify assets by programming), and were worthy of sophisticated editing tools available to developers. Increasingly, developers involved in test automation insisted that coding a test be as straightforward and simple as coding any other asset. Finally, testers began to develop and share open source code, which required the testing tool and its programming language be compatible with sharing.

IBM Rational Functional Tester

Rational Functional Tester was developed specifically to address these latest concerns of testers and developers. Rational Functional Tester is available in two flavors: one built on the popular open source Eclipse framework and the other built on the Microsoft Visual Studio framework, putting testers and developers in the same Integrated Development Environment. Rational Functional Tester uses either the Java programming language or VB.NET, providing access to a large body of code made available on the Internet, giving testers a choice of industry-standard languages, and allowing developers to code in a language that is already familiar to them.

Rational Functional Tester uses advanced, patented technology to find objects, and it provides controls to the tester to permit flexible recognition and control over what happens when an object is not found or only partially recognized. Rational Functional Tester includes advanced object recognition and advanced verification features based on a pattern-matching mechanism.

The combination of modern Integrated Development Environments (IDEs) and programming languages means that Rational Functional Tester is easily extended to support automation of not only ERP environments, but other environments, too. In many cases, these extensions can be performed by the tester directly.

With all of these capabilities, Rational Functional Tester makes test automation an activity that delivers significant return on its investment. Large regression test suites can be created with the assurance that they will provide high value via limited maintenance, common tooling, low barrier to adoption, and adaptability for the future.

Total Quality Solution

By itself, Rational Functional Tester is a powerful tool. It enables test teams to automate large portions of their test suites, enabling them to expedite their test cycles and expand on the amount of testing that is done. However, it is just one arrow in a team's quiver. It lacks the capability to pull metrics beyond what passed and failed. Plugging it into a quality management and defect tracking solution enables them to take advantage of the benefits of automation while capturing valuable test metrics (density, trending, and so on), tracing test results back to requirements, and so on.

IBM Rational Quality Manager is a total quality management solution built on the Jazz platform. It is completely web-based, providing rich reports, dashboards, and useful web 2.0 capabilities (such as RSS feeds). Rational Quality Manager (RQM) allows teams to create robust test plans, tracing their test cases back to requirements. Rational Functional Tester is one of the tools that can be used to automate test cases in RQM. The end result is a complete testing solution that provides end-to-end traceability—requirements to test cases to any defects that might be uncovered—using Rational Functional Tester's rich automation for unattended testing on a selected subset of test cases.

Introductions to Eclipse and to Visual Studio

If you are new to the Rational Functional Tester flavor that you are about to start using, you might benefit from a general introduction to your Rational Functional Tester IDE. Because Eclipse and Visual Studio are so prominent in the computing world, there is a wealth of resources of all different types at all different levels. The following list includes a few introductory-level articles and books, but this list is far from comprehensive. The following suggestions are just to get you started; you are encouraged to pursue the vast range of information that is available.

Eclipse

What is Eclipse, and how do I use it?
www.ibm.com/developerworks/opensource/library/os-eclipse.html

Getting started with the Eclipse Platform.
www.ibm.com/developerworks/opensource/library/os-ecov/?Open&ca=daw-ec-dr

An introduction to Eclipse for Visual Studio users.
www.ibm.com/developerworks/opensource/library/os-eclipse-visualstudio/?ca=dgr-lnxw01Eclipse-VS

***Eclipse: Step by Step*.** Joe Pluta. Lewisville, TX: MC Press Online (2003).

***Eclipse IDE Pocket Guide*.** Ed Burnette. Sebastopol, CA: O'Reilly (2005).

Visual Studio .NET

Introduction to Visual Studio .NET.
http://msdn.microsoft.com/en-us/library/ms973867.aspx

Quick Tour of the Integrated Development Environment. http://msdn.microsoft.com/en-us/library/ms165088(VS.80).aspx

***Visual Basic 2005 in a Nutshell*, Third Edition.** Tim Patrick, Steven Roman, Ron Petrusha, Paul Lomax. Sebastopol, CA: O'Reilly (2006).

***Beginning VB 2008: From Novice to Professional*.** Christian Gross. Berkeley, CA: Apress (2008).

How to Use This Book

This book is a reference for both novice and advanced users. The initial chapters focus on the basics of using Rational Functional Tester, whereas the latter chapters become more advanced. Ideally, you will find that as your automation abilities mature, you will get more value from the specialized content found deeper in the book.

Although you might find value in reading this book cover to cover, it was conceived for use on a topic-by-topic basis. Many chapters are standalone; however, some are related and build on each other. In addition, the initial chapters were crafted specifically for the novice Rational Functional Tester user, whereas the latter chapters were written with the experienced user or power user in mind. The following chapter list "calls out" these relationships.

Chapter 1, Overview of Rational Functional Tester—An introductory chapter for the novice user that gives a broad overview of Rational Functional Tester and its basic features.

Chapter 2, Storyboard Testing—A description of the RFT Simple Scripting Visual Editor, introduced in RFT 8.1, to open up RFT scripting to the nontechnical user and to the novice user.

Chapter 3, General Script Enhancements—This chapter provides a grab-bag of highly useful techniques ranging from basic data capture to script synchronization, to an introduction to `TestObjects`. It is for the intermediate level user.

Chapter 4, XML and Rational Functional Tester—A tour of basic XML programming in Rational Functional Tester to familiarize the reader with the XML libraries in Rational Functional Tester for testing with XML and for dealing with Rational Functional Tester's own XML infrastructure. It is for the intermediate or advanced user.

Chapter 5, Managing Script Data—An in-depth chapter discussing Rational Functional Tester Datapools, database access from Rational Functional Tester, and the use of properties files, XML files, and flat files with Rational Functional Tester. It is for the intermediate level user or the advanced level user.

Chapter 6, Debugging Scripts—Eclipse and Visual Studio have powerful, integrated debuggers. As you build your scripts, the IDE debuggers can provide valuable information for pinpointing the root cause of script problems. This chapter surveys the debugging tools most useful for RFT script development. It is for the intermediate level user or the advanced level user.

Chapter 7, Managing Script Execution—This chapter covers the different aspects of controlling the execution flow in scripts. Topics include manipulating playback settings, using conditional and looping logic to optimize script flow, handling errors that scripts encounter, regression scripts, and executing outside of the Rational Functional Tester environment. It is for the intermediate level user.

Chapter 8, Handling Unsupported Domain Objects—A discussion of the Rational Functional Tester API for dealing with domains that are not supported by Rational Functional Tester. It is for the advanced user.

Chapter 9, Advanced Rational Functional Tester Object Map Topics—A detailed examination of the Rational Functional Tester Object Map features and underpinnings for the advanced user. This chapter pairs with Chapter 10 to supply the background for how `TestObjects` are handled by Rational Functional Tester.

Chapter 10, Advanced Scripting with Rational Functional Tester TestObjects—A discussion of `TestObjects`, how they are used by the Object Map, how they can be used without the Object Map, and how they can be used to manipulate unsupported controls. It is for the advanced user.

Chapter 11, Testing Specialized Applications—An examination of specialized environments, where Rational Functional Tester requires special setup or produces scripts with notable characteristics. It covers scripting for terminal-based (mainframe) applications, SAP, Siebel, and Adobe Flex. It is for the intermediate or advanced user.

Chapter 12, Extend Rational Functional Tester with External Libraries—This chapter shows how to call a range of external libraries from Rational Functional Tester, including log4j and log4net (to create custom logging solutions), JAWIN and Microsoft Interop libraries (to create an Excel reporting utility), and PDFBox and IKVM.Net (to test PDF files). It is for the advanced user.

Chapter 13, Building Support for New Objects with the Proxy SDK—A step-by-step discussion of how to develop Rational Functional Tester support for third-party Java or .NET controls with the Rational Functional Tester Proxy SDK. Chapter 10 provides useful background. It is for the advanced user.

Chapter 14, Developing Scripts in the VB.NET Environment—Rational Functional Tester historically is a creature of Eclipse and Java. More recently, it has entered the world of Visual Studio and VB.NET. This chapter offers tips and tricks for using Rational Functional Tester in the Visual Studio environment. It is for the intermediate to advanced user.

Chapter 15, Using Rational Functional Tester in a Linux Environment—This chapter describes the major similarities and differences between a Rational Functional Tester

Linux environment and the Rational Functional Tester Windows environment. A basic installation procedure is discussed in addition to the creation of a basic script. It is for the novice user.

Chapter 16, Internationalized Testing with Rational Functional Tester—A discussion of how to set up internationalized testing in the Rational Functional Tester framework. Rational Functional Tester has all the core functionality to handle testing of internationalized applications; this chapter shows one way to use basic Rational Functional Tester features to create an internationalized testing framework. Chapters 9 and 10 offer the background for this discussion. It is for the advanced user.

Appendix A, Advanced Logging Techniques—A tour of using advanced logging packages and methods in Rational Functional Tester scripting, including emailing results, XSL transformation of the RFT XML log, and custom logging. It is for the advanced user.

Appendix B, Regular Expressions in Rational Functional Tester—A detailed discussion of the use of regular expressions in Rational Functional Tester scripting. It is for the advanced user.

Finally, as an additional aide, all code listings for the book are available via download as Rational Functional Tester project exports. Readers can access these code samples at http://www.ibm.com/developerworks/rational/library/09/testengineeringrft/index.html.

CHAPTER 3

General Script Enhancements

Chip Davis, Daniel Chirillo, Daniel Gouveia

Record and playback can be a successful technique when automating your test cases. It provides a quick and simple means to create the minute programs that verify pieces of your application. Of course, the degree of success for this technique depends upon the complexity of the application you are testing. The more complex your application becomes (for example, using third-party controls, multiple tiers of communication, and so on), the less successful a record-and-playback approach becomes.

This chapter is meant to serve as a next step to simply recording and playing back scripts. It serves as a basis for you to build core test automation skills. Subsequent chapters expound on some of the topics found in this chapter. However, before you run (for example, debugging your custom scripting, handling custom controls, building your own proxies, and so on), you must learn to walk. Here you learn to take steps toward running with any automation effort. These steps include: learning how to synchronize your scripts with any latency found in your application, manipulating data, using the Clipboard object (think, "cut, copy, and paste"), working with the data found in a test object, creating your own custom verification points, and developing your own custom methods that reuse one or more of these techniques.

Test Script Synchronization

It is critical for script playback to stay synchronized with the application it tests. In Chapter 1, "Overview of Rational Functional Tester," you saw that Rational Functional Tester had playback settings that enabled you to increase the amount of time it would wait for a GUI object to render. The playback settings affect all scripts, not just the ones with which you are experiencing issues.

Further, they are specific to each installation of Rational Functional Tester. You need to be cognizant of this when your team members want to run your scripts from their installations.

This section focuses on building synchronization right into your scripts. You learn how to free yourself from the dependencies of installation-specific, global playback settings. Topics covered include:

- Creating general delays in your scripts using the `sleep()` methods
- Introducing intelligent delays into your scripts using the `waitForExistence()` methods
- Writing delays that are triggered by object properties
- Using timers to understand how long a piece of your script takes to execute

The aim of this section is to help you become familiar with enhancing Rational Functional Tester's default synchronization without being dependent upon global settings. If you build the necessary delays into your scripts, you can take advantage of them from any installation of Rational Functional Tester without the need to adjust playback settings.

Placing a Generic Delay in Your Script

Enhancing Rational Functional Tester's default synchronization can be as simple as adding a generic delay to your script. You can use the `sleep()` method (`Sleep()` function in the .NET version). This is available to you while recording or via coding.

When you are recording and notice that your application is slow to respond, you can turn to the Script Support Functions button on your toolbar. This is shown in Figure 3.1.

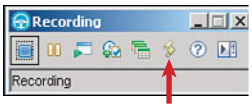


Figure 3.1 The Script Support Functions button on the Recording toolbar

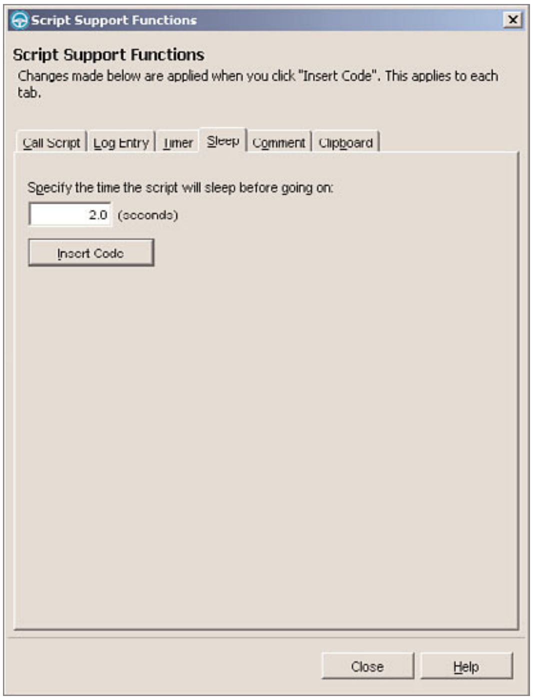
Engaging this button provides you with the option to place a `sleep()` method into your script. Please refer to Figure 3.2 for a visual reference.

To use this function, you specify the number of seconds that you think your script will need to handle any latency issues with your application. You then click the **Insert** button. If you were using the Eclipse version of Rational Functional Tester, you would see a line in your script that looks like the following:

```
sleep(2.0);
```

Otherwise, using the .NET Studio version of Rational Functional Tester, you would see:

```
Sleep(2.0)
```


Script Support Functions

Script Support Functions

Changes made below are applied when you click "Insert Code". This applies to each tab.

Specify the time the script will sleep before going on:

2.0 (seconds)

Figure 3.2 Script Support Functions Sleep tab

Listing 3.1 shows what these lines look like in the context of a test script:

Listing 3.1 Using `sleep() / Sleep()` in a script

Java

```
public void testMain(Object[] args)
{
        startApp("ClassicsJavaA");

        // PAUSE EXECUTION, USING sleep() METHOD, FOR 2 SECONDS
        sleep(2.0);

        // Frame: ClassicsCD
        tree2().performTest(VerifyComposerListVP());
        classicsJava(ANY,MAY_EXIT).close();
}
```

VB.NET

```
Public Function TestMain(ByVal args() As Object) As Object
        StartApp("ClassicsJavaA")

        ' PAUSE EXECUTION, USING sleep() METHOD, FOR 2 SECONDS
        Sleep(2.0)

        ' Frame: ClassicsCD
        Tree2().PerformTest(VerifyComposerListVP())
        ClassicsJava(ANY,MAY_EXIT).Close()
        Return Nothing
End Function
```

If you execute either of the scripts displayed in Listing 3.1, execution pauses when it hits the `sleep()` method. The duration of the pause is determined by the number of seconds that was specified in the argument to the method (the number in the parentheses). In the case of the examples in Listing 3.1, execution pauses for two seconds.

You also have the ability to code these lines directly into your scripts. For instance, you might not have recorded these delays because you didn't experience poor performance with your application while recording. However, upon playback of your scripts, you see periodic latency issues that surpass the default synchronization built into Rational Functional Tester. Using, the playback logs, you can find out what object was unable to be found on playback. This enables you to go into your script and code in the necessary `sleep()` lines. Please remember that you need to provide, as an argument to the method, the number of seconds to pause. This should be a floating number (for example, 2.0, 3.5, and so on).

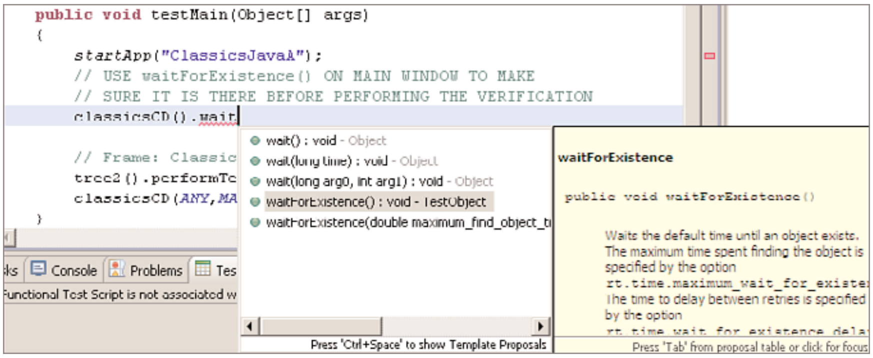
The benefit of using the `sleep()` method is that you can now place extended synchronization capabilities right within your scripts, freeing you from the dependency of global playback settings. The downside of using the `sleep()` method is you are dealing with a static period of time. In other words, your script has to pause for the specified period of time regardless of whether or not the application responds faster than anticipated.

Waiting for Test Objects to Come into Existence

When GUI objects aren't rendered in a timely manner, you simply want your script to wait until they appear. Chapter 1 discussed the pros and cons of using global delay settings (for example, using Rational Functional Tester's playback settings). You also just learned about using the `sleep()` method in the previous section. This is helpful because the delay is specific to a script, releasing your dependencies on the global tool settings. However, it is a static delay. It waits until a specified number of seconds elapse, regardless of whether the GUI object appears sooner or not. This section discusses a happy medium between the global delay settings and the `sleep()` method.

The Rational Functional Java API provides a useful `waitForExistence()` method or `WaitForExistence()` function if you're using VB.NET. In either case, it comes in two variants. One is dependent upon global playback settings. The other is script-specific.

To use this method, you need to call it from a test object. Simply put, you need to tell your script which object to wait for. You can either type the name of the object, followed by opening and closing parentheses, directly into your code and hit the period key, or you can simply place your cursor in the script where you wish the command to go, right-click the test object in the Test Objects folder (within the Script Explorer view), and click **Insert at Cursor** from the displayed menu. Either of these two options displays an IntelliSense drop-down window (some people refer to this as "code complete"). This window provides you with a list of methods and functions that you can call. Typing the word wait shows you the methods and functions that begin with wait. The `waitForExistence()` method is among them. Figures 3.3 and 3.4 show how to call the `waitForExistence()` method, using the Eclipse and .NET Studio environments, respectively.



```
public void testMain(Object[] args)
{
    startApp("ClassicsJavaA");
    // USE waitForExistence() ON MAIN WINDOW TO MAKE
    // SURE IT IS THERE BEFORE PERFORMING THE VERIFICATION
    classicsCD().wait

    // Frame: Classic
    tree2().performTe
    classicsCD(ANY,MA
}
```

```
wait() : void - Object
wait(long time) : void - Object
wait(long arg0, int arg1) : void - Object
waitForExistence() : void - TestObject
waitForExistence(double maximum_find_object_ti
```

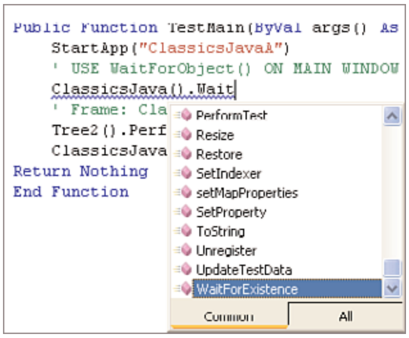
waitForExistence

```
public void waitForExistence()
```

Waits the default time until an object exists. The maximum time spent finding the object is specified by the option `rt.time.maximum_wait_for_existe` The time to delay between retries is specified by the option `rt.time.wait_for_existence_dela`

ks
Console
Problems
Tes
Functional Test Script is not associated w
Press 'Ctrl+Space' to show Template Proposals
Press 'Tab' from proposal table or click for focus

Figure 3.3 Calling the `waitForExistence()` method—Java language



```
Public Function TestMain(ByVal args() As
    StartApp("ClassicsJavaA")
    ' USE WaitForObject() ON MAIN WINDOW
    ClassicsJava().Wait
    ' Frame: Cla
    Tree2().Perf
    ClassicsJava
Return Nothing
End Function
```

```
PerformTest
Resize
Restore
SetIndexer
setMapProperties
SetProperty
ToString
Unregister
UpdateTestData
WaitForExistence
```

Figure 3.4 Calling the `WaitForExistence()` function—VB.NET language

If you use the Java language, you need to append a semicolon at the end of the line. If you use the VB.NET language, you are set and there is nothing further to do. Listing 3.2 shows you how the command looks in the same Rational Functional Tester script, using Java and VB.NET, respectively.

Listing 3.2 Using `waitForExistence() / WaitForExistence()` in a script

Java

```
public void testMain(Object[] args)
{
            startApp("ClassicsJavaA");

      // USE waitForExistence() ON MAIN WINDOW TO MAKE
      // SURE IT IS THERE BEFORE PERFORMING VERIFICATION
      classicsJava().waitForExistence();

      // Frame: ClassicsCD
      tree2().performTest(VerifyComposerListVP());
      classicsJava(ANY,MAY_EXIT).close();
}
```

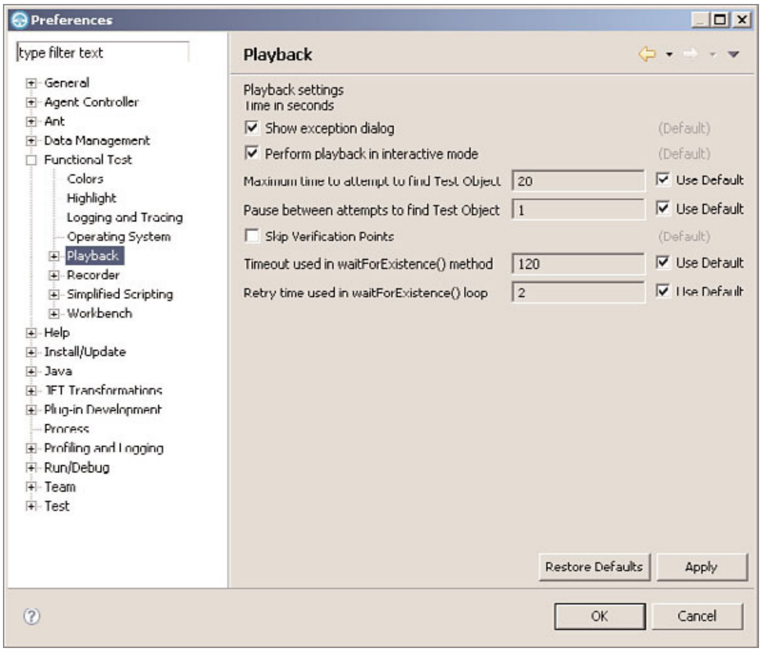
VB.NET

```
Public Function TestMain(ByVal args() As Object) As Object
      StartApp("ClassicsJavaA")

      ' USE WaitForExistence() ON MAIN WINDOW TO MAKE
      ' SURE IT IS THERE BEFORE PERFORMING VERIFICATION
      ClassicsJava().WaitForExistence()

      ' Frame: ClassicsCD
      Tree2().PerformTest(VerifyComposerListVP())
      ClassicsJava(ANY,MAY_EXIT).Close()
      Return Nothing
End Function
```

In Listing 3.2, `ClassicsJava()` is a test object, representing the main window of the application. You call this test object's `waitForExistence()` method to tell your script that it needs to wait until this object appears, prior to executing the next line. The value of this method is that it immediately continues execution of the script after the test object renders on the screen (versus waiting a static period of time). However, this particular variant of the method is dependent upon global delay settings accessed from Rational Functional Tester's primary playback settings. Figures 3.5 and 3.6 show these options for Java and VB.NET.

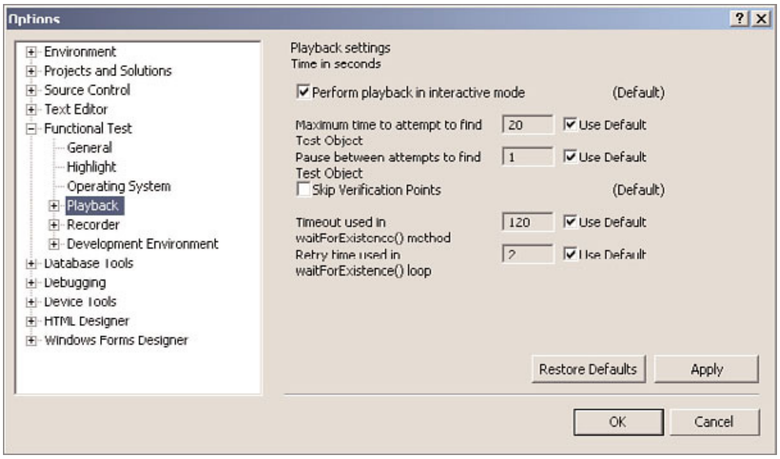


Playback

Playback settings
Time in seconds

☑ Show exception dialog		(Default)
☑ Perform playback in interactive mode		(Default)
Maximum time to attempt to find Test Object	20	☑ Use Default
Pause between attempts to find Test Object	1	☑ Use Default
☐ Skip Verification Points		(Default)
Timeout used in waitForExistence() method	120	☑ Use Default
Retry time used in waitForExistence() loop	2	☑ Use Default

Figure 3.5 Primary playback settings—`waitForExistence()` options in Java



Playback settings
Time in seconds

☑ Perform playback in interactive mode		(Default)
Maximum time to attempt to find Test Object	20	☑ Use Default
Pause between attempts to find Test Object	1	☑ Use Default
☐ Skip Verification Points		(Default)
Timeout used in waitForExistence() method	120	☑ Use Default
Retry time used in waitForExistence() loop	2	☑ Use Default

Figure 3.6 Primary playback settings—`WaitForExistence()` options in VB.NET

The last two settings are specific to the `waitForExistence()/WaitForExistence()` method. If you want to change how long it waits or how often it checks for a test object's existence, you simply override the defaults and supply your own values. You should note that using this variant of the method keeps you dependent upon Rational Functional Tester's global settings.

To become script-specific, you would use the second variant of this method—`waitForExistence(double arg0, double arg1)`. This enables you to provide the maximum amount of time script playback waits for the object to render (that is arg0). You also enter the amount of time it waits between attempts to find the test object (that is arg1). You specify these two values using seconds. Using this version of the method keeps your scripts independent from the values in the global delay settings for the `waitForExistence()` method. Listing 3.3 shows how this would look in both Java and VB.NET.

Listing 3.3 Using the script-specific version of `waitForExistence()/WaitForExistence()` in a script

Java

```
public void testMain(Object[] args)
{
      startApp("ClassicsJavaA");

      // USE waitForExistence() ON MAIN WINDOW TO MAKE
      // SURE IT IS THERE BEFORE PERFORMING VERIFICATION
      // WAIT A MAXIMUM OF 180 SECONDS/3 MINUTES
      // CHECK FOR THE MAIN WINDOW EVERY 2 SECONDS
      classicsJava().waitForExistence(180.0, 2.0);

      // Frame: ClassicsCD
      tree2().performTest(VerifyComposerListVP());
      classicsJava(ANY,MAY_EXIT).close();
}
```

VB.NET

```
Public Function TestMain(ByVal args() As Object) As Object
      StartApp("ClassicsJavaA")

      ' USE WaitForExistence() ON MAIN WINDOW TO MAKE
      ' SURE IT IS THERE BEFORE PERFORMING VERIFICATION
      ' WAIT A MAXIMUM OF 180 SECONDS/3 MINUTES
      ' CHECK FOR THE MAIN WINDOW EVERY 2 SECONDS
      ClassicsJava().WaitForExistence(180.0, 2.0)

      ' Frame: ClassicsCD
      Tree2().PerformTest(VerifyComposerListVP())
      ClassicsJava(ANY,MAY_EXIT).Close()
      Return Nothing
End Function
```

You can see that your script now waits for the main window (for example, `classicsJava`) of the application for a maximum of 180 seconds, searching for it every two seconds.

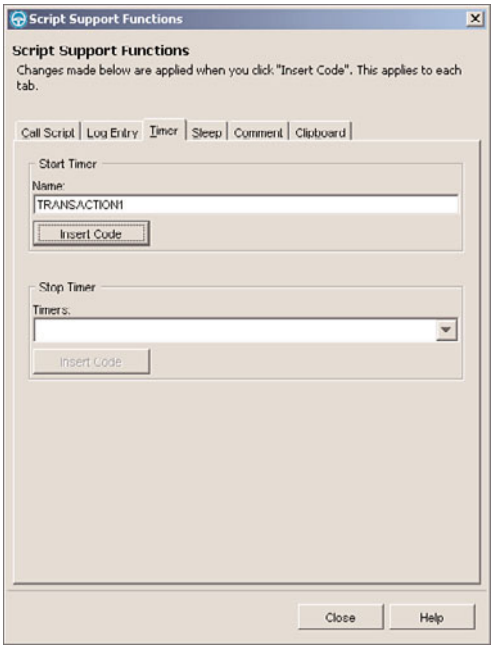
Of the three synchronization topics that you have seen thus far, this is the most desirable. It provides you with the ability to have your scripts wait until a test object appears versus waiting for a static amount of time to elapse. Further, it enables you to become script-specific, allowing any tester on your team to run it without having to adjust global playback delay settings.

Timing How Long Something Takes to Execute in a Script

Rational Functional Tester provides timers that give an indication of the amount of time that elapsed during script playback. You can use these to estimate how long your scripts take to complete. You can also use these for capturing how long a specific section of your script takes to play back.

The first option that Rational Functional Tester provides is the built-in timer. You can use this option while recording. You can also add it to your scripts later, using two simple lines of code.

While recording, you can define a timer from the GUI recording options, resulting in the appropriate piece of code being written into your script. This option is available from the Script Support Functions button on your Recording toolbar. Figure 3.7 shows this option.


Script Support Functions

Changes made below are applied when you click "Insert Code". This applies to each tab.

Start Timer
Name: TRANSACTION1

Figure 3.7 Script Support Functions—Timer tab

You need to enter only a name for your timer, and then click the **Insert Code** button. This creates the following line in your script:

```
timerStart("TRANSACTION1");
```

If you use Rational Functional Tester .NET, you see the following line instead:

```
TimerStart("TRANSACTION1")
```

You need to create the corresponding command to stop your timer. To do this, you simply perform the following steps:

1. Access your Script Support Functions.
2. Choose the **Timer** tab.
3. Click the name of timer (you would like to stop) from within the **Timers:** combobox.
4. Click the **Insert Code** button.

This will insert either a `timerStop("TRANSACTION1");` line or a `TimerStop ("TRANSACTION1")` line into your script, depending on which version of Rational Functional Tester you are using, Java or .NET, respectively.

These can be added manually, too. You simply type in the necessary `timerStart()` and `timerStop()` methods. For instance, Listing 3.4 shows how to use the `timerStart()` and `timerStop()` methods for timing how long it takes for the main window of the application to render.

Listing 3.4 Using timers in a script

Java
```
public void testMain(Object[] args)
{
      // TIME HOW LONG MAIN WINDOW TAKES TO RENDER
      // AFTER STARTING THE APP
      timerStart("TRANSACTION1");
      startApp("ClassicsJavaA");
      timerStop("TRANSACTION1");

      // Frame: ClassicsCD
      classicsJava(ANY,MAY_EXIT).close();
}
```

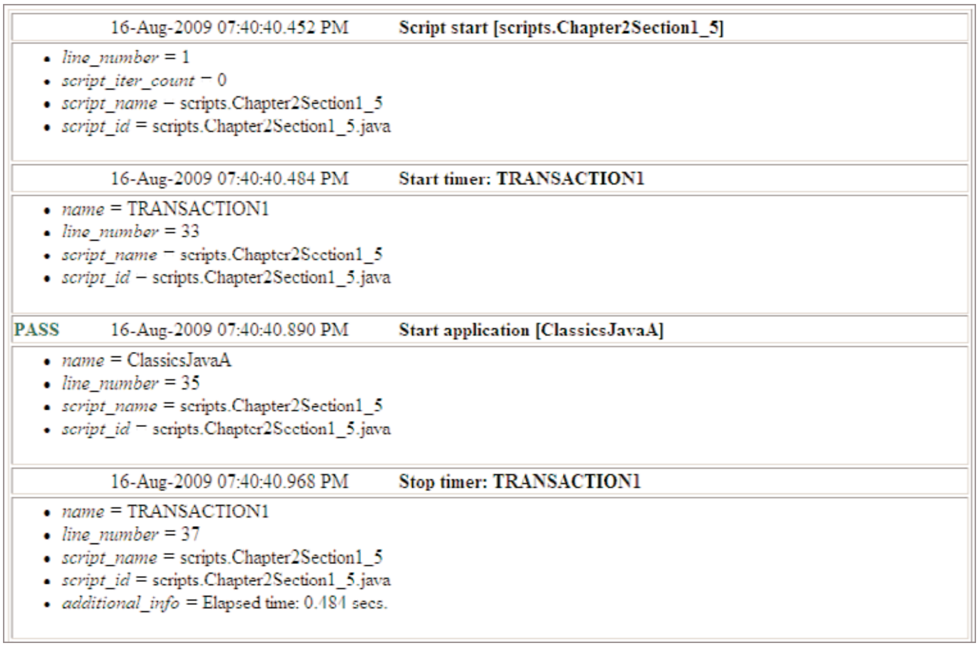
VB.NET
```
Public Function TestMain(ByVal args() As Object) As Object
' TIME HOW LONG MAIN WINDOW TAKES TO RENDER
' AFTER STARTING THE APP
```

```
TimerStart("TRANSACTION1")
StartApp("ClassicsJavaA")
TimerStop("TRANSACTION1")

' Frame: ClassicsCD
ClassicsJava(ANY,MAY_EXIT).Close()

Return Nothing
End Function
```

The prior two scripts result in a couple of events being written into the test log. The first is the timer getting started by the script. The second is the timer stopping. The stop event also displays the amount of time that elapsed. Figure 3.8 shows what these events look like in the HTML log.



```
        16-Aug-2009 07:40:40.452 PM    Script start [scripts.Chapter2Section1_5]
  • line_number = 1
  • script_iter_count = 0
  • script_name = scripts.Chapter2Section1_5
  • script_id = scripts.Chapter2Section1_5.java

        16-Aug-2009 07:40:40.484 PM    Start timer: TRANSACTION1
  • name = TRANSACTION1
  • line_number = 33
  • script_name = scripts.Chapter2Section1_5
  • script_id = scripts.Chapter2Section1_5.java

PASS    16-Aug-2009 07:40:40.890 PM    Start application [ClassicsJavaA]
  • name = ClassicsJavaA
  • line_number = 35
  • script_name = scripts.Chapter2Section1_5
  • script_id = scripts.Chapter2Section1_5.java

        16-Aug-2009 07:40:40.968 PM    Stop timer: TRANSACTION1
  • name = TRANSACTION1
  • line_number = 37
  • script_name = scripts.Chapter2Section1_5
  • script_id = scripts.Chapter2Section1_5.java
  • additional_info = Elapsed time: 0.484 secs.
```

Figure 3.8 Timer events in the HTML log

Note that Rational Functional Tester's Script Assure technology can influence timers. If a test object's property values change and it is wrapped in between timer commands, you can, and often do, see excessive elapsed times in your log files. This is because the time it takes to find a test object gets captured and added into the timer.

One downside of Rational Functional Tester's timers is that they don't return a value for external consumption. For instance, you might want to write the elapsed time out to an external file (for example, if you have a test harness that has its own logging mechanism). This is where you can turn to the native scripting language, Java, or VB.NET.

Java's System class provides the `currentTimeMillis()` method for capturing transaction times. It returns a long data type. It is a native method that directly calls out to the operating system. Therefore, its resolution can vary when you run your tests on different machines running different operating systems. Listing 3.5 revisits the `waitForExistence(double arg0, double arg1)` method. This time the `System.currentTimeMillis()` method is used to capture how long the `waitForExistence()` method actually had to wait.

Listing 3.5 Capturing elapsed times using native Java capabilities

```
public void testMain(Object[] args)
{
      startApp("ClassicsJavaA");

      // START TIMER - first call to System.currentTimeMillis()
      long startTime = System.currentTimeMillis();

      // USE waitForObject() ON MAIN WINDOW TO MAKE
      // SURE IT IS THERE BEFORE PERFORMING VERIFICATION
      classicsJava().waitForExistence(180.0, 2.0);

      // END TIMER - second call to System.currentTimeMillis()
      long stopTime = System.currentTimeMillis();

      // CALCULATE HOW LONG THE waitForExistence METHOD TOOK
      long intervalTime = stopTime - startTime;

      // WRITE INTERVAL TO LOG
      logInfo("Playback needed to wait " + intervalTime + " ms for
application to launch");

      // AND CONSOLE
      System.out.println("Playback needed to wait " + intervalTime + "
ms for application to launch");

      // AND EXTERNAL FILE (TO BE IMPLEMENTED)
      // TODO - ADD CODE TO WRITE TO FILE

      // Frame: ClassicsCD
      tree2().performTest(VerifyComposerListVP());
      classicsJava(ANY,MAY_EXIT).close();
}
```

VB.NET's `System` namespace offers the `DateTime` and `TimeSpan` structures for acquiring elapsed times. To use these, you first obtain the value contained in `DateTime`'s `Now` property to capture the start time. You capture `Now`'s value again when you want to get the stop time. To gather the total elapsed time, you use the `TimeSpan` structure and initialize it with the difference between your stop and start times. Depending upon how you want to view the elapsed time, `TimeSpan` lets you display it in hours, minutes, seconds, milliseconds, and so on. Listing 3.6 shows how this comes together. This builds off of an earlier script that used the `WaitForExistence()` function.

Listing 3.6 Capturing elapsed times using native VB.NET capabilities

```
Public Function TestMain(ByVal args() As Object) As Object
      StartApp("ClassicsJavaA")

' START TIMER - first call to System.DateTime.Now
Dim startTime As System.DateTime = System.DateTime.Now

' USE WaitForObject() ON MAIN WINDOW TO MAKE SURE IT IS THERE BEFORE
PERFORMING VERIFICATION
ClassicsJava().WaitForExistence(180.0, 2.0)

' END TIMER - second call to System.DateTime.Now
Dim stopTime As System.DateTime = System.DateTime.Now

' CALCULATE HOW LONG THE waitForExistence METHOD TOOK
Dim intervalTime As System.TimeSpan = stopTime - startTime

' WRITE INTERVAL TO LOG
LogInfo("Playback needed to wait " & intervalTime.TotalMilliseconds & "
ms for application to launch")
' AND MsgBox
MsgBox("Playback needed to wait " & intervalTime.TotalMilliseconds & "
ms for application to launch")

' AND EXTERNAL FILE (TO BE IMPLEMENTED)
' TODO - ADD CODE TO WRITE TO FILE

' Frame: ClassicsCD
      Tree2().PerformTest(VerifyComposerListVP())
      ClassicsJava(ANY,MAY_EXIT).Close()
      Return Nothing
End Function
```

Synchronization and timing are two critical tools in an automation engineer's toolbox. They help you determine how long a piece of your script takes to execute and, perhaps more importantly, they help you keep your scripts—or pieces of them—in synch with your application's playback.

This section covered the appropriate timing and synchronization methods provided by Rational Functional Tester. You saw how you can use delays to wait a specified number of seconds. You also saw how you can tell your scripts to wait until a desired test object appears. Lastly, you learned how to use timers to capture how long a piece of your script takes to execute.

You should now be able to apply the appropriate techniques and script code to deal with any latency issues that appear in the application you test. You can use timers to determine the duration of the latencies. You can then select the appropriate method for dealing with them.

Working with Test Objects

Chapter 1 discussed the use of test object maps at a high level. It also provided an overview on test objects including adding, modifying, and deleting them. Rational Functional Tester records and uses these objects to interact with an application's user interface. You begin to unleash the power of Rational Functional Tester when you choose to manipulate these objects.

Test objects offer more than what you see when you are done recording. Typically, a recorded script shows you actions against test objects. You see things such as clicks, double-clicks, text-entry commands, and so on. When you dig deeper into these, you find that you can access interesting information. You can programmatically access properties that describe the object and the data that it contains. You also have access to an object's methods that can be invoked from your script. Chapter 10, "Advanced Scripting with Rational Functional Tester TestObjects," covers this topic. For the purposes of this chapter, you examine the basics of accessing test object properties and data using the scripting language.

After you have the basics of programmatically accessing test object data, you get a look at how you can start encapsulating your custom scripting to develop your own methods and functions. This is a great means for other members of your team to take advantage of the work you did, avoiding any redundant custom scripting.

Working with the Clipboard Object

Rational Functional Tester gives you a type of verification point that works with textual data on the Windows Clipboard. You might ask, "Why would I want to use the Windows Clipboard with Rational Functional Tester?" You want to do this for several reasons, but in most cases, it comes down to an easy way to get around object recognition difficulties. Often you need a test script to do something, such as checking a value, and if the normal way to do this (in this example, using a verification point) doesn't immediately work, using the Clipboard might be the fastest way to make it work.

Getting a test script to work reliably with the application interface you are testing often comes down to robust object recognition. Dealing with object recognition is covered in more detail in Chapter 8, "Handling Unsupported Domain Objects," and in Chapter 9, "Advanced Rational Functional Tester Object Map Topics." In some cases, you want to follow that information to fully implement Rational Functional Tester's recognition and interface with the test objects. The reason that you might not follow that approach, but instead use the Clipboard, usually has to do with how quickly you need to get something to work. Do not think of the Clipboard approach as a work-around or interim solution, though, as it is often the most effective and efficient way to automate a test procedure.

You use the Clipboard for handling object recognition with Rational Functional Tester, which specifically means getting, setting, or otherwise using values from the application under test or associated applications. You might do this for any of the following reasons:

- Verifying some functionality or output from the application
- Controlling the timing of your test scripts
- Manipulating the flow of the test procedures
- Logging additional information into the test results

When you use this technique, you record steps to copy and/or paste data to the Clipboard, and then use it as a kind of data source. The Clipboard can be used as an intermediate place to hold data from the application under test, and Rational Functional Tester can get data from the Clipboard or send data to it. You can also test anything that gets onto the Clipboard with a verification point.

There are three general approaches to using the Clipboard with automated testing:

1. Verifying some value from the application
2. Getting some value from the application to use in the test
3. Sending some value from the test to the application

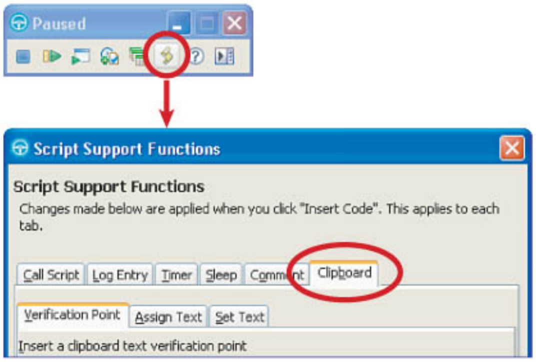
For each of these tactics, you usually record some steps to select and copy data from some GUI, and then add some Clipboard actions into the test script. This chapter describes several examples, with specific steps, for each of the three approaches.

You do not have to manually code the methods previously described; you can simply use the Recorder toolbar to generate the Clipboard code. You add these Clipboard actions much like you would other kinds of verification points or script actions. The wizards for adding Clipboard actions into a test are found in the Script Support Functions on the Recorder toolbar.

Figure 3.9 illustrates how to access the Clipboard actions from the Rational Functional Tester Recorder toolbar.

Like all of Rational Functional Tester's script support functions, you need to click **Insert code** to add the commands to your test script.

The first consideration you have when using the Clipboard is determining exactly how you can select and copy, or perhaps paste, data to and from the Clipboard. In some cases, you might select text using a mouse click and drag, followed by a Ctrl+C keyboard shortcut. In other cases, you might have a menu option in the application under test to copy information to the Clipboard. You might have to use a combination of interactions with the application under test, the operating system, various keyboard inputs, and right-clicking or other menu selections.



Script Support Functions

Changes made below are applied when you click "Insert Code". This applies to each tab.

Insert a clipboard text verification point

Figure 3.9 Recorder toolbar—the Clipboard

The exact method used to get data to and from the Clipboard is of critical importance to the resilience of the automated test script. Some of the steps that you record, such as clicking and dragging, rely on coordinates within an object. These methods are therefore more susceptible to playback failure. You should always strive to record actions that are most likely to replay in the maximum (and reasonable) test environments and playback conditions.

The following looks at several related examples of creating test scripts that use the Clipboard and starts with one possible scenario of creating automated test scripts leveraging the Clipboard. This is not intended to be the most realistic scenario possible, but it illustrates different ways of using the Clipboard with Rational Functional Tester.

In this scenario, you have to test a system that includes an older reporting component. Rational Functional Tester can recognize only parts of this component's interface and doesn't seem to automatically capture a particular value: the current job number. This alphanumeric string has to be checked against expected values and it is used as an input value to another test. Finally, because this is a lower priority test case, you cannot spend too much time developing the test script.

You can figure out how to get Rational Functional Tester to recognize the objects and properties to access the job number value, but that might take time. You realize that the job number field can be easily selected and copied to the Clipboard, so you use the Clipboard to quickly get a working automated test script.

First, you create a Clipboard verification point for the job number value. This differs from other verification points in that you do not directly access the object or properties to get the value. You can record a test script to do this as follows:

1. Record the initial steps of the test script, getting to the point where the job number (the value we want to verify) is displayed.
2. Select the job number text and copy it to the Clipboard. A robust method for doing this could be tabbing to the field if the application GUI is not likely to change.
3. Open the **Script Support Functions** from the Recorder toolbar, and click **Clipboard** > **Verification Point Tab** (see Figure 3.10). You will see the value that was copied in the verification value field.

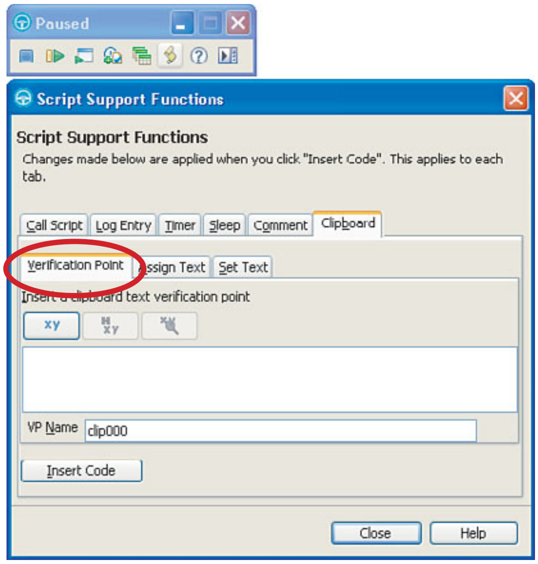


Figure 3.10 Clipboard verification point

NOTE You have the option of creating a regular expression from the Clipboard value. This works the same way as other verification points.

Clearing the Clipboard

Any time you are using the Clipboard as part of a procedure, it is a good practice to clear the Clipboard as an initial step. Copying to the Clipboard normally replaces any prior data, but clearing it first ensures that you are not inadvertently including unwanted data. The exact steps to do this depend on the operating system and version and the applications you use.

4. Enter a descriptive name for the verification point in the VP Name field.
5. Click **Insert Code,** and then click **Close** to exit the Script Support Functions and return to recording mode.

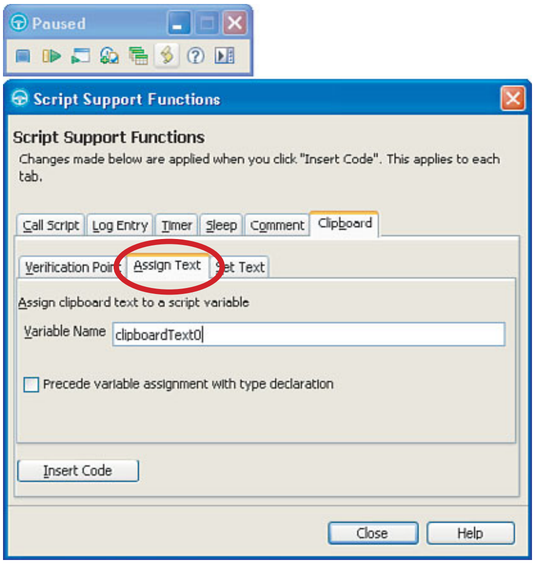
At this point, the script contains the code for a Clipboard verification point, as shown here:

```
getSystemClipboard().performTest(JobNumberVP());
```

You now get the job number value and store it in a local variable within the test script. You can record a test script to do this as follows:

1. Record any initial steps of the test script or continue the previous example, getting to the point where the job number (the value we want to capture) is displayed.

2. Select the job number text and copy it to the Clipboard. If you combine this with the previous verification point example, you can skip directly to the next step.
3. Open the **Script Support Functions**, and click **Clipboard** > **Assign Text Tab** (see Figure 3.11).



Script Support Functions

Changes made below are applied when you click "Insert Code". This applies to each tab.

Assign clipboard text to a script variable

Variable Name: clipboardText0

☐ Precede variable assignment with type declaration

Figure 3.11 Getting a value from the Clipboard

NOTE You do *not* see the value that was copied to the Clipboard here. If you want to check that the right value was copied, you can switch to the Verification Point tab where you will see it, and then switch back to the Assign Text tab.

4. Enter a descriptive name for the script variable in the Variable Name field, and then select **Precede variable assignment with type declaration**.
5. Click **Insert Code**, then click **Close** to exit the Script Support Functions and return to recording mode.

At this point, the script contains the code for a new string variable to hold whatever was copied to the Clipboard, as shown here:

```
String JobNum = getSystemClipboard().getText();
```

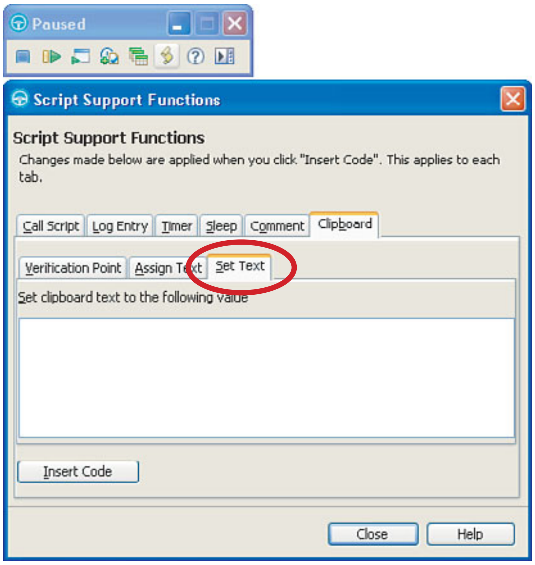
This variable can now be used for any number of reasons within the test.

You also have the ability to send data to the Clipboard. You might want to do this if you need to test your application's paste function. If you set the data in the Clipboard immediately before performing a paste function, you can ensure that you know what should be pasted into your application. Another reason you might do this is for continuity. For example, if one piece of your application provides a record number, you might wish to store it in a variable (for example, `JobNum` in the preceding line of script). You can then use this variable to paste into the piece of your application that does a record lookup. In any case, setting the Clipboard is easy. You simply use the following line of code in your script:

```
    getSystemClipboard().setText(JobNum);
```

Returning to the scenario, you now need to enter a new job number, different from the current one, into the application. You want to paste this value from the Clipboard into the application GUI field. You therefore need to set the value contained in the Clipboard. You might calculate or look up the particular value in several ways, but for this example, you simply show a static string. You can record a test script to do this as follows:

1. Record any initial steps of the test script or continue the previous example, getting to the point where you enter the new value.
2. Open the **Script Support Functions**, and then click **Clipboard** > **Set Text Tab**. The value field on this tab is empty (shown in Figure 3.12).



Script Support Functions

Changes made below are applied when you click "Insert Code". This applies to each tab.

Set clipboard text to the following value

Figure 3.12 Sending a value to the Clipboard

3. Enter the value that you want to send to the Clipboard. You might simply enter a placeholder value when you are recording and then replace it in the script to some calculated or retrieved variable (for example, `JobNum`).

4. Click **Insert Code**, then click **Close** to exit the Script Support Functions and return to recording mode.

At this point, the script now contains the code to set the value of the Clipboard, as shown here:

```
getSystemClipboard().setText("SomeValue");
```

As mentioned in step 3, you might want to replace the hard-coded value entered during recording mode (`IncrementedJobNumber`) with some variable or other value. An incomplete example of this is shown in following code snippet.

```
// retrieve JobNum from the getText() method
String JobNum = getSystemClipboard().getText();
...
//replace recorded setText() value to JobNum variable
getSystemClipboard().setText(JobNum);
```

Instead of using the recorder, you might want to manually add the code for Clipboard interactions. This is easy to do because you would have to add only one line of code for each type of Clipboard action, not counting the lines of script to select, copy, or paste the data. An example of a line to capture something to the Clipboard is:

```
someTextField().inputKeys("{ExtHome}+{ExtEnd}^c");
```

The preceding line of code performs the following steps:

1. Press the **Home** key (that is `{ExtHome}`)—This places the cursor at the beginning of the field.
2. Perform a **Shift+End** keystroke combination (that is `+{ExtEnd}`)—This selects everything in the field (from the beginning to the end of it).
3. Perform the **Ctrl+C** keystroke combination (that is ^c)—This executes the copy function of the application.

   After this, you can manually add the following line to your script for a Clipboard verification point:

   ```
   getSystemClipboard().performTest(myClipVP());
   ```

The Rational Functional Tester API has a public interface `IClipboard` in the `com.rational. test.ft.script` package that defines the set of methods exposed for a Clipboard. The three methods in this are:

- `getText`—Returns a copy of the text from this Clipboard.
- `setText`—Copies the supplied text value to this Clipboard.
- `performTest`:—Captures, compares, and logs active Clipboard text against the supplied baseline data.

NOTE There are four variations of the `performTest` method for various verification point settings such as retry and timeout values. Refer to the Rational Functional Tester API reference in the online Help for more information.

There are several things to consider when using the Clipboard with Rational Functional Tester:

- The Clipboard is often used to verify, get, and set data without mapping and directly accessing objects and properties. This can make script maintenance more difficult because you do not have an object map of the objects containing the values.
- The steps used to capture data to the Clipboard or paste data from it might be less resilient and more susceptible to playback problems. This can be a risk if you have to resort to click and drag or other actions using coordinates.
- There is a risk with the system Clipboard that scripts might not play back the same way in different test environments. Although this should ideally work the same way on Windows and Linux operating systems, it can also fail on the same operating system due to different conditions in the test environment.

Viewing an Object's Properties

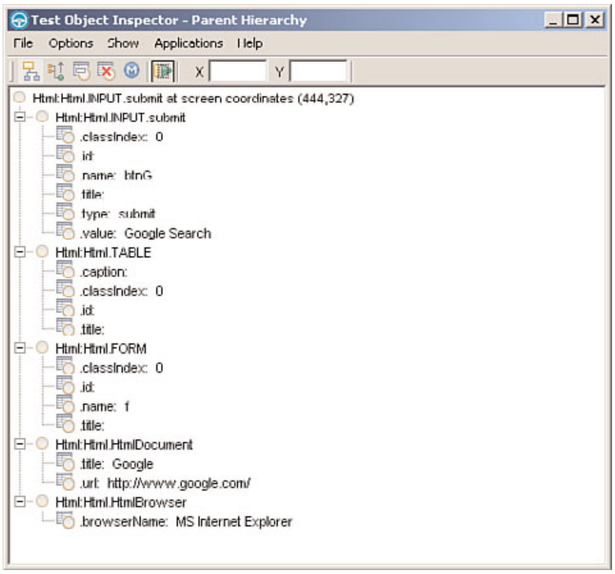
Rational Functional Tester has a useful (and often overlooked) utility called the Test Object Inspector that you can use to examine a wealth of information about test objects (including properties). No coding is required to use the Inspector. It is a design-time tool that can often be used to gather information you need to write your own code. It's powerful because it shows you *almost* everything that you can manipulate in the test object through Rational Functional Tester.

To launch the Inspector, click **Run > Test Object Inspector** (in Eclipse) or **Tools > Test Object Inspector** (in Visual Studio). After the Inspector displays, by default it remains on top of all windows on your Desktop. As you move your mouse over objects, the Inspector captures information and displays it in the Inspector window. To inspect any object on your desktop, simply hover over the object and press the Pause key on your keyboard. This tells the Inspector to freeze with the information currently captured, and enables you to move your mouse and not lose the information currently displayed in the Inspector window.

After you pause it, you can view information about the test object. The Inspector can show five broad categories of information: Parent Hierarchy, Inheritance Hierarchy, Properties, Non-Value Properties, and Method information.

To select what you want the Inspector to display, press the corresponding button on the toolbar or make your selection under the Inspector's Show menu. In addition to these broad categories, there are options that you can turn on and off under the Options menu. The following examines five examples:

- **Parent Hierarchy—**Shows the hierarchical path from the test object to the top-level window. For example, if you inspected the Google Search button on www.google.com (http://www.google.com), the Inspector shows what is displayed in Figure 3.13.



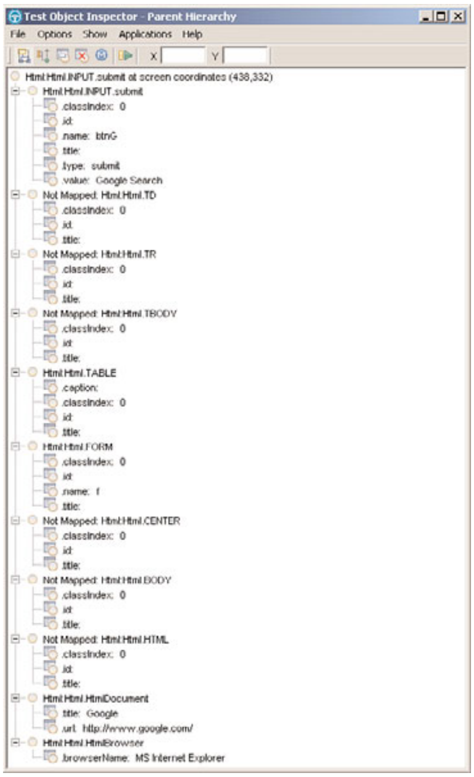
```
Html:Html.INPUT.submit at screen coordinates (444,327)
Html:Html.INPUT.submit
    .classIndex: 0
    .id:
    .name: btnG
    .title:
    .type: submit
    .value: Google Search
Html:Html.TABLE
    .caption:
    .classIndex: 0
    .id:
    .title:
Html:Html.FORM
    .classIndex: 0
    .id:
    .name: f
    .title:
Html:Html.HtmlDocument
    .title: Google
    .url: http://www.google.com/
Html:Html.HtmlBrowser
    .browserName: MS Internet Explorer
```

Figure 3.13 Test Object Inspector—Parent Hierarchy for Google application

The Inspector displays the same recognition information (hierarchy and recognition property names and current values) that the object map would store for the button, except that the tree is upside down (the browser window is the bottom, not at the top, as in the object map) and the hierarchy is flattened (there are no indentations).

You might turn on this option when viewing a test object's parent hierarchy that does a full disclosure of sorts. This option expands what's displayed in the Parent Hierarchy to include *nonmappable* objects. To turn this option on, toggle **Hide not Mappable** under the Options menu. Nonmappable objects are objects that RFT does not think are interesting and does not, therefore, add to the object map (RFT doesn't record any actions against nonmappable objects). However, sometimes you want to know about nonmappable objects. In such situations, turning on this option is useful. Figure 3.14 shows the Parent Hierarchy for the Google Search button.

Without displaying nonmappable objects, the parent of the button is the HTML table; when you include nonmappable objects, you see that the parent (the real parent in a


Test Object Inspector - Parent Hierarchy
File Options Show Applications Help
X Y

```
Html:Html.INPUT.submit at screen coordinates (438,332)
Html:Html.INPUT.submit
    .classIndex: 0
    .id:
    .name: btnG
    .title:
    .type: submit
    .value: Google Search
Not Mapped: Html:Html.TD
    .classIndex: 0
    .id:
    .title:
Not Mapped: Html:Html.TR
    .classIndex: 0
    .id:
    .title:
Not Mapped: Html:Html.TBODY
    .classIndex: 0
    .id:
    .title:
Html:Html.TABLE
    .caption:
    .classIndex: 0
    .id:
    .title:
Html:Html.FORM
    .classIndex: 0
    .id:
    .name: f
    .title:
Not Mapped: Html:Html.CENTER
    .classIndex: 0
    .id:
    .title:
Not Mapped: Html:Html.BODY
    .classIndex: 0
    .id:
    .title:
Not Mapped: Html:Html.HTML
    .classIndex: 0
    .id:
    .title:
Html:Html.HtmlDocument
    .title: Google
    .url: http://www.google.com/
Html:Html.HtmlBrowser
    .browserName: MS Internet Explorer
```

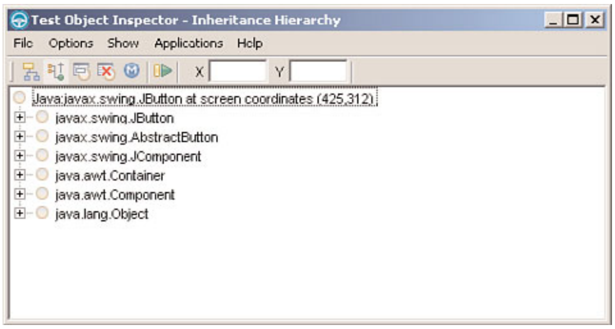
Figure 3.14 Test Object Inspector—the Parent Hierarchy for Google application, nonmappable objects

sense) is the TD HTML element. To differentiate between these two views of test object parentage, you say that the table is the button's *mappable parent* and the TD is its *nonmappable* parent. The RFT API differentiates between mappable and nonmappable parents and children.

- **Inheritance Hierarchy—**Shows class inheritance hierarchy of the test object's class. This option does not apply the HTML test objects. Figure 3.15 shows the inheritance hierarchy for the Place Order button in Classics.

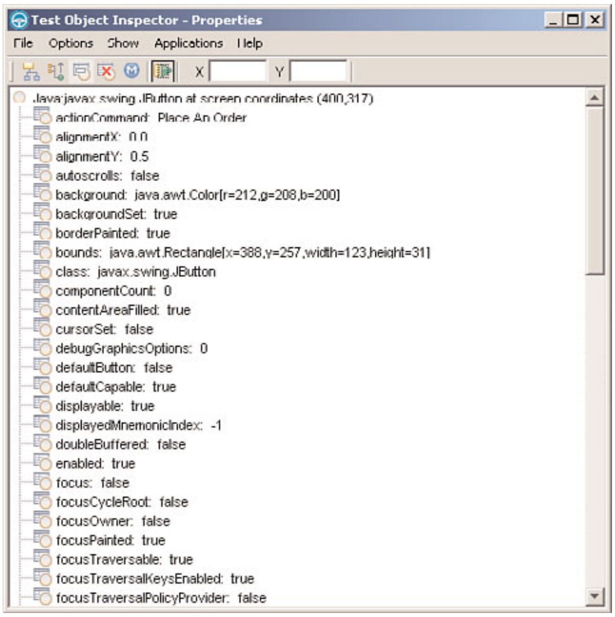
  Here, the Inspector shows that the (Java) class of the button is `javax.swing.JButton`, which inherits from `AbstractButton`, which inherits from `JComponent`, and so on up to `Object`, the root class of all classes in Java.

- **Properties—**Shows all the test object's value properties (property names and values). Figure 3.16 shows what you get when you examine the properties of the Place Order button in Classics.



```
Java:javax.swing.JButton at screen coordinates (425,312)
  javax.swing.JButton
  javax.swing.AbstractButton
  javax.swing.JComponent
  java.awt.Container
  java.awt.Component
  java.lang.Object
```

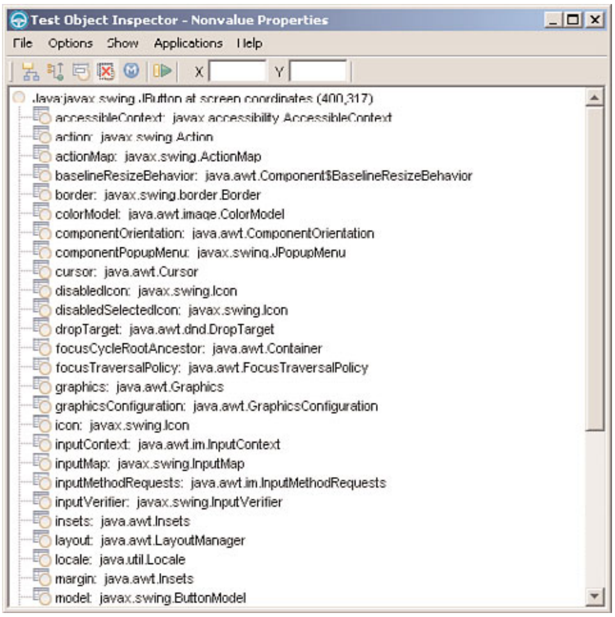
Figure 3.15 Test Object Inspector—Inheritance Hierarchy



```
Java:javax.swing.JButton at screen coordinates (400,317)
  actionCommand:  Place An Order
  alignmentX:  0.0
  alignmentY:  0.5
  autoscrolls:  false
  background:  java.awt.Color[r=212,g=208,b=200]
  backgroundSet:  true
  borderPainted:  true
  bounds:  java.awt.Rectangle[x=388,y=257,width=123,height=31]
  class:  javax.swing.JButton
  componentCount:  0
  contentAreaFilled:  true
  cursorSet:  false
  debugGraphicsOptions:  0
  defaultButton:  false
  defaultCapable:  true
  displayable:  true
  displayedMnemonicIndex:  -1
  doubleBuffered:  false
  enabled:  true
  focus:  false
  focusCycleRoot:  false
  focusOwner:  false
  focusPainted:  true
  focusTraversable:  true
  focusTraversalKeysEnabled:  true
  focusTraversalPolicyProvider:  false
```

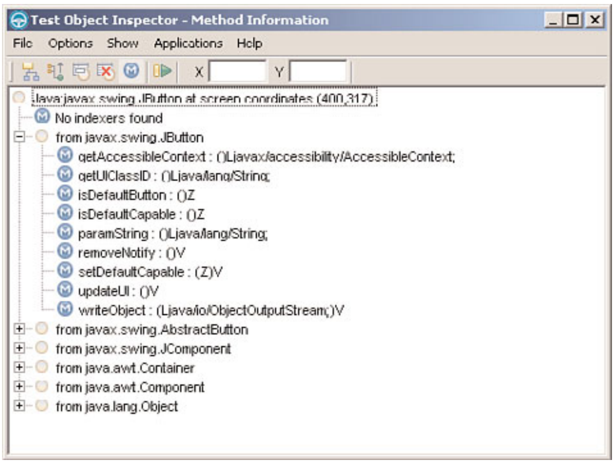
Figure 3.16 Test Object Inspector—properties

- **NonValueProperties—**Shows all the test object's nonvalue property names and each nonvalue property's data type. Using the Place Order button, the Inspector shows the following nonvalue properties (displayed in Figure 3.17).
- **Method Information—**Shows all the methods that can be invoked directly on the test object via RFT's `invoke()` method (which we cover in Chapter 10). The methods are grouped by the test object class' inheritance chain. An example of this is shown in Figure 3.18.



```
Test Object Inspector - Nonvalue Properties
File  Options  Show  Applications  Help
X  Y
Java:javax.swing.JButton at screen coordinates (400,317)
    accessibleContext:  javax.accessibility.AccessibleContext
    action:  javax.swing.Action
    actionMap:  javax.swing.ActionMap
    baselineResizeBehavior:  java.awt.Component$BaselineResizeBehavior
    border:  javax.swing.border.Border
    colorModel:  java.awt.image.ColorModel
    componentOrientation:  java.awt.ComponentOrientation
    componentPopupMenu:  javax.swing.JPopupMenu
    cursor:  java.awt.Cursor
    disabledIcon:  javax.swing.Icon
    disabledSelectedIcon:  javax.swing.Icon
    dropTarget:  java.awt.dnd.DropTarget
    focusCycleRootAncestor:  java.awt.Container
    focusTraversalPolicy:  java.awt.FocusTraversalPolicy
    graphics:  java.awt.Graphics
    graphicsConfiguration:  java.awt.GraphicsConfiguration
    icon:  javax.swing.Icon
    inputContext:  java.awt.im.InputContext
    inputMap:  javax.swing.InputMap
    inputMethodRequests:  java.awt.im.InputMethodRequests
    inputVerifier:  javax.swing.InputVerifier
    insets:  java.awt.Insets
    layout:  java.awt.LayoutManager
    locale:  java.util.Locale
    margin:  java.awt.Insets
    model:  javax.swing.ButtonModel
```

Figure 3.17 Test Object Inspector—nonvalue properties



```
Test Object Inspector - Method Information
File  Options  Show  Applications  Help
X  Y
Java:javax.swing.JButton at screen coordinates (400,317)
    No indexers found
  ⊟ from javax.swing.JButton
        getAccessibleContext : ()Ljavax/accessibility/AccessibleContext;
        getUIClassID : ()Ljava/lang/String;
        isDefaultButton : ()Z
        isDefaultCapable : ()Z
        paramString : ()Ljava/lang/String;
        removeNotify : ()V
        setDefaultCapable : (Z)V
        updateUI : ()V
        writeObject : (Ljava/io/ObjectOutputStream;)V
  ⊞ from javax.swing.AbstractButton
  ⊞ from javax.swing.JComponent
  ⊞ from java.awt.Container
  ⊞ from java.awt.Component
  ⊞ from java.lang.Object
```

Figure 3.18 Test Object Inspector—Method Information

To the right of each method name, the Inspector shows the method's JNI signature (argument list and return type).

What is the difference between a value property and a nonvalue property? A *value property* is a property whose value Rational Functional Tester can reconstruct in its own JVM and store and compare it to other values in another context. There is a practical application of this: When you perform a Properties verification point, for example, Rational Functional Tester captures and must then store (as a baseline) the values of the properties that are being verified. When the verification point is executed, Rational Functional Tester must know how to compare the baseline values with the actual values. The properties Rational Functional Tester can manipulate in this manner are referred to collectively as *value properties*. Properties whose values Rational Functional Tester does not know how to store and compare are called *nonvalue properties*.

If you look at the list of value properties, you can see that the value of each property is either a `String` (actionCommand), `boolean` (enabled), `numeric` (alignment), or a Java class that consists of numeric fields (`Rectangle` and `Color`). These values can easily be restructured, stored, and compared in another context (RFT's JVM or a baseline VP file).

The nonvalue properties, on the other hand, are different in nature. In the nonvalue property list, you see properties such as `graphicsConfiguration`, icon, and cursor. It's not difficult to appreciate that it is difficult (at best) for RFT to store the value of a `Cursor` object or any of the other nonvalue properties. Both value and nonvalue property values are available to you through coding.

Retrieving All Properties of a Test Object

The two sets of properties discussed in the "Viewing an Object's Properties" section—value and nonvalue properties—can be retrieved from a test object in a script by invoking `getProperties()` and `getNonValueProperties()`, respectively. Each of these methods returns a Hashtable of property names and values. For nonvalue properties, the values are references to the actual objects in the target application (remote JVM). Listing 3.7 shows three methods: one that retrieves value properties, one that retrieves nonvalue properties, and one that calls both methods.

Listing 3.7 Methods for retrieving all properties of a test object

Java

```java
public void printValueProperties(TestObject testObject) {

      Hashtable valueProperties = testObject.getProperties();
      Enumeration valuePropNames = valueProperties.keys();
      Enumeration valuePropValues = valueProperties.elements();
      while (valuePropNames.hasMoreElements()) {
            System.out.println(valuePropNames.nextElement() + ":"
            + valuePropValues.nextElement());
      }
}

public void printNonValueProperties(TestObject testObject) {

      Hashtable nonValueProperties = testObject.getNonValueProperties();
      Enumeration nonValuePropNames = nonValueProperties.keys();
      Enumeration nonValuePropValues = nonValueProperties.elements();
      while (nonValuePropNames.hasMoreElements()) {
            System.out.println(nonValuePropNames.nextElement() + ":"
            + nonValuePropValues.nextElement());
      }
}

public void printAllProperties(TestObject testObject) {

      printValueProperties(testObject);
      printNonValueProperties(testObject);
}
```

VB.NET

```
Public Sub printAllProperties(ByVal testObject As TestObject)
   printValueProperties(testObject)
   printNonValueProperties(testObject)
End Sub

Public Sub printValueProperties(ByVal testobject As TestObject)
   Dim valueProperties As Hashtable = testobject.GetProperties
   Dim valuePropNames As ICollection = valueProperties.Keys
   Dim valuePropValues As ICollection = valueProperties.Values

   Dim enumPropNames As IEnumerator = valuePropNames.GetEnumerator()
   Dim enumPropVals As IEnumerator = valuePropValues.GetEnumerator()

   Do While enumPropNames.MoveNext And enumPropVals.MoveNext
      Dim currentProp As String = enumPropNames.Current
      Dim currentPropVal As Object = enumPropVals.Current
         If TypeOf currentPropVal Is Object Then
            If Not (TypeOf currentPropVal Is String) Then
               currentPropVal = currentPropVal.ToString
            End If
         End If
         Console.WriteLine(currentProp + ":" + currentPropVal)
   Loop
End Sub


Public Sub printNonValueProperties(ByVal testobject As TestObject)
```

```
   Dim valueProperties As Hashtable = testobject.GetNonValueProperties
   Dim valuePropNames As ICollection = valueProperties.Keys
   Dim valuePropValues As ICollection = valueProperties.Values

   Dim enumPropNames As IEnumerator = valuePropNames.GetEnumerator()
   Dim enumPropVals As IEnumerator = valuePropValues.GetEnumerator()

   Do While enumPropNames.MoveNext And enumPropVals.MoveNext
      Console.WriteLine(enumPropNames.Current + ": " +
            enumPropVals.Current)
   Loop
End Sub
```

Retrieving the Value of a Single Property

To retrieve the value of a single property, use the `getProperty()` method in the `TestObject` class. You can use `getProperty()` with both value and nonvalue properties. In this section, the discussion is limited to value properties. The signature of `getProperty()` for Java is:

```
Object getProperty( String propertyName )
```

The signature of `getProperty()` for VB.NET is:

```
Function GetProperty( propertyName as String) as Object
```

The argument is the name of the property whose value you want to retrieve. Because `getProperty` can be used with any value or nonvalue property, the return type is generic `Object`. You typically want to cast to a specific type (for example, `String`).

For value properties, you can even use Rational Functional Tester to generate the code for you.

1. Place your cursor at the line in your script where you want the code to generate.
2. Insert recording.
3. Launch the Verification Point and Action Wizard.
4. Click the desired test object.
5. In the Select an Action window, click **Get a Specific Property Value**.
6. The wizard will then display all the test object's value property names and values. Click the property you want, and then click **Next**.
7. In the Variable Name window, enter a variable name to hold the returned value (RFT generates a variable name for you but you will typically always want to change the variable name), and click **Finish**.

If you selected the label property, Rational Functional Tester generates the following code.

In VB.NET:

```
Dim buttonLabel As String = PlaceOrder().GetProperty( "label" )
```

In Java:

```
String buttonLabel = (String)placeOrder().getProperty( "label" );
```

In Java, you need to explicitly cast to the correct type. For example, if you retrieve a non-String property value, such as the `Background` property, you see:

```
java.awt.Color buttonBackground =
(java.awt.Color)placeOrder().getProperty( "background" );
```

If you pass a property name that does not exist in the object, Rational Functional Tester throws a `PropertyNotFoundException`.

As you become more comfortable with Rational Functional Tester, you can rely less on the wizard to generate code for you.

Programmatically Retrieving Data from an Object in My Application

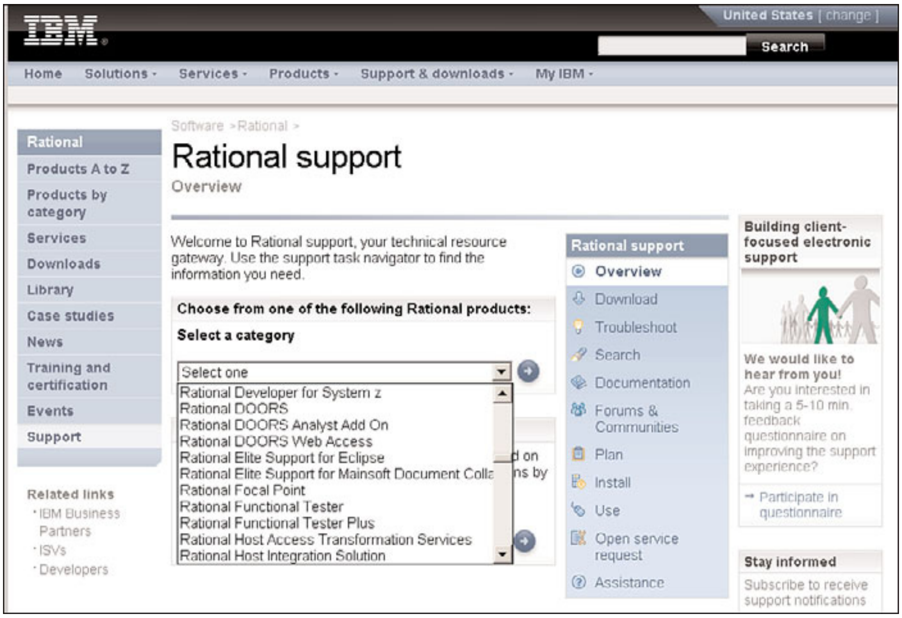
Retrieving data from objects in your application is a critical and fundamental task. You might need to get the data so that you can use it at a later point in your test; you might get it because you're creating your own verification point. No matter what your motivation is for pulling data from an object, the RFT API makes extracting data easy.

For the purposes of this chapter, the world of test objects is divided into two broad categories: those in which the RFT Verification Point Wizard sees data and those in which the RFT Verification Point Wizard doesn't see data. The TestObject techniques discussed in this chapter (with the exception of the "Working with the Clipboard Object" section) are limited to the former category of test objects, which is covered in other chapters.

You use one of two methods to get data from a test object: `getProperty()` or `getTestData().getProperty()` is appropriate if the data is available as a property of the test object and the format of the data as a property value suites your needs. For example, if you want to get the visual text on a button (such as the Place Order button in Classics or the Search button at www.google.com), `getProperty()` would work fine. If you examine these objects using the Test Object Inspector (see the "Viewing an Object's Properties" section), you can see that in Classics, you need to get the value of the label property; the Google button would require that you get the value property.

Things become more interesting when the data you're interested in is available as a property, but not in the desired format. For example, on the Rational Support home page, there is a list of products (see Figure 3.19).

Imagine you wanted to get the list values. Examining the list under the Test Object Inspector microscope reveals a `.text` property whose value is the contents of the list. There's a problem, though: It's one big string—you don't know where the breaks between items are. Sometimes it is not a critical issue. For instance, you might just want to know if "Functional Tester" appears



Welcome to Rational support, your technical resource gateway. Use the support task navigator to find the information you need.

Choose from one of the following Rational products:

Select a category

Select one
Rational Developer for System z
Rational DOORS
Rational DOORS Analyst Add On
Rational DOORS Web Access
Rational Elite Support for Eclipse
Rational Elite Support for Mainsoft Document Colla
Rational Focal Point
Rational Functional Tester
Rational Functional Tester Plus
Rational Host Access Transformation Services
Rational Host Integration Solution

We would like to hear from you! Are you interested in taking a 5-10 min. feedback questionnaire on improving the support experience?

Subscribe to receive support notifications

Figure 3.19 Rational Support site

anywhere in the list. If that is the case, you can simply use `getProperty()` to capture the `.text` property and be done with it. However, what if you want to know how many items are in the list? To achieve this, `getProperty()` does not suit your needs. In this instance, the usage of `getTestData()` would aid your efforts.

If you run Rational Functional Tester's Verification Point and Action Wizard and click **Data Verification Point**, you see that Rational Functional Tester is able to identify each list item discretely. Because Rational Functional Tester's Data Verification Point Wizard is able to capture the data, you can capture the data in code. In fact, any data that the Rational Functional Tester Data Verification Point Wizard is able to capture can be retrieved programmatically in a script.

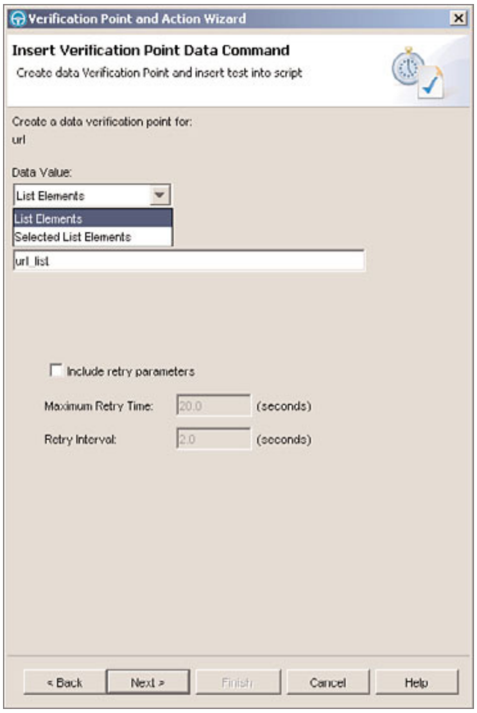
Your script can retrieve any data from a test object that is accessible to the Data Verification Point Wizard by invoking the `getTestData()` method on the test object. Here is the method signature:

```
ITestData getTestData( String testDataType )
```

`getTestData()` does precisely what its name suggests: It gets and returns data from a test object. That's it. It does not verify anything. It requires a String argument—the *test data type*—to capture from the test object. The notion of a test data type requires some explanation.

Test Data Types

Recall that when you use Rational Functional Tester's Verification Point Wizard, you have to indicate to the wizard which test data to verify. For example, if you've selected a list object (as in the case of the Rational Support site example), you have to express if you want Rational Functional Tester to verify all the values in the list or just the selected elements. You can see this in Figure 3.20.


Verification Point and Action Wizard

Insert Verification Point Data Command

Create data Verification Point and insert test into script

Create a data verification point for:
url

Data Value:
List Elements

List Elements
Selected List Elements

url_list

Include retry parameters

Maximum Retry Time: 20.0 (seconds)

Retry Interval: 2.0 (seconds)

Figure 3.20 Verification Point and Action Wizard

The argument passed to `getTestData()` indicates to `getTestData()` that same detail that you express to the Verification Point Wizard when you select a value in the Data Value list. In the case of a recorded verification point, it tells Rational Functional Tester which data to retrieve and verify; in the case of `getTestData()`, it indicates which data to retrieve and return to the caller of `getTestData()`.

ITestData

`getTestData()` returns the data as a Java interface type: `ITestData`. The `ITestData` interface is inherited by several subinterfaces, each of which has methods to store and manipulate specific types of data, for example, `ItestDataList` (for data in lists), `ITestDataTable` (for data in tables), and `ITestDataTree` (for data in trees). Each of these interfaces has methods to query and extract data in the manner appropriate for that data structure. For example, if you retrieved the data in a table, you might want to know how many columns and rows of data there are. You might also want to get all the values in row 3 or column 2. `ITestDataTable` has methods that make this possible in an intuitive fashion. When working with lists, you might want to know how many items are in the list and get all the values. `ITestDataList` has methods to do this.

Even though the signature of the return type of `getTestData()` is `ITestData` (the base interface type), what will actually be returned is a reference to a subinterface of `ITestData`, for example, `ITestDataTable`. Even though you can always treat any of the returned data as base `ITestData` (that is you can upcast the returned reference), you almost always assign to a variable of the specific subinterface type so that you can call the specific methods available in the subtype. In the case of Java, you explicitly downcast; in VB.NET, you can use an implicit downcast. For example (don't worry here about the argument to `getTestData()`; the focus here is on the downcasting), Java:

```
ITestDataTable iData = (ITestDataTable)someTable().getTestData(
"contents" );
```

In VB. Net:

```
Dim iData As ITestDataTable = someTable.GetTestData( "contents" )
```

Using `getTestData()` successfully requires that you:

- Know the appropriate test data type argument for the test object in question.
- Know which methods to use in the returned `ITestData` to access the data elements.

In the next sections, we look at how to acquire this knowledge.

Determining a Test Object's Valid Test Data Type Arguments to getTestData()

You can use two techniques to determine a test object's valid data type arguments:

1. Call `getTestDataTypes()` on the test object you need to get data from. `getTestDataTypes()` is a method in the TestObject class that returns a `Hashtable`, which holds the same information that the Rational Functional Tester Verification Point Wizard displays (and then some).
2. Right-click the test object in the Script Explorer, and click **Interface Summary**. This displays a page in the Rational Functional Tester documentation that summarizes how it works with the selected test object, including the valid data types it can use.

For example, if you invoke `getTestDataTypes()` on the Orders table in the Classics sample application (**Admin > Orders**) and print the returned hashtable to the console (Output window in Visual Studio), you have the following:

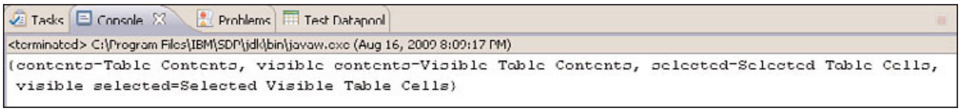
In Java:

```
System.out.println( orderTable().getTestDataTypes() );
```

In VB.NET:

```
Console.WriteLine(someTable.GetTestDataTypes)
```

You see something similar to what is displayed in Figure 3.21.



```
<terminated> C:\Program Files\IBM\SDP\jdk\bin\javaw.exe (Aug 16, 2009 8:09:17 PM)
{contents=Table Contents, visible contents=Visible Table Contents, selected=Selected Table Cells,
 visible selected=Selected Visible Table Cells}
```

Figure 3.21 Console output of `orderTable().getTestDataTypes()`

The keys in the hashtable (contents, visible contents, selected, and selected and visible selected) are the valid test data type arguments to `getTestData()` for the test object in question; the value of each hashtable key is a description (for you, not Rational Functional Tester) of the test data type. Based on that output, the following are all valid calls to make on the Orders table:

```
orderTable().getTestData( "contents" );
orderTable().getTestData( "selected" );
orderTable().getTestData( "visible contents" );
orderTable().getTestData( "visible selected" );
```

Using getTestData to Extract Data from a Text Field

The first example of using `getTestData`() is one that might not initially seem interesting. You can use `getTestData()` to retrieve data from text fields. To do so, pass `getTestData()` an argument of "text;" it returns the text as `ITestDataText`, which is the simplest subinterface in the `ITestData` family to work with. It's simple because you need to know only one method to get the data into a usable string: `getText()`. As an example, look at these two text fields:

- The Search field at www.google.com <http://www.google.com> (HTML domain)
- The Name field on Classics' Place an Order window (Java domain)

If you fire up and point the Inspector (see the "Viewing an Object's Properties" section in this chapter) at each of these objects, you can see that in the case of the HTML text field, you need to get the `value` property. In the case of the Java Swing field, you need to get the `text` property. If you use `getTestData()`, in both cases, you pass an argument of text. This is useful if you want to create a generic method that retrieves text (expressed as a single string) from any type of test object. Listing 3.8 shows both a Java and VB.NET example of using `getTestData()`.

Listing 3.8 Getting data from a text field

Java

```
ITestDataText iData = (ITestDataText)textField().getTestData( "text" );
String value = iData.getText();
```

VB.NET

```
Dim iData as ItestDataText  = textField().getTestData( "text" )
dim value as String = iData.getText()
```

Notice that in Java, you must explicitly cast to `ITestDataText`.

Using getTestData to Extract Data from a List

Lists are relatively easy objects to start with, though there are two twists that follow. The first issue is the test data types:

First twist: If you retrieve the selected element from a Swing `JList`, the data is returned as `ITestDataText` (in this case, you then call `getText()` and you're done); for any other type of list, the data is returned as `ITestDataList`.

Second twist (relates only to `ITestDataList`): Getting the individual data values out of the `ITestDataList` requires a little more work than you might like. You work with three interface types: `ITestDataList`, `ITestDataElementList`, and `ITestDataElement`.

Tables 3.1 and 3.2 show the most commonly used methods in each of these interfaces.

Table 3.1 List Object Data Types

Data Type	Return Type	Comments
List	`ITestDataList`	Retrieves all list elements.
Selected	`ITestDataList` `ITestDataText`	Retrieves the selected items. The selected elements in HTML and .Net lists are returned as `ITestDataList;` selected data in Swing lists `(javx.swing.JList)` is returned as `ITest DataText.`

Table 3.2 List Object Interface Methods

Interface	Methods
`ITestDataList`	`int getElementCount() ITestDataElementList getElements()`
`ITestDataElementList`	`ITestDataElement getElement( int )`
`ItestDataElement`	`Object getElement()`

At a high level, this is the algorithm to get list data:

1. Call `getTestData()` on your list test object. The data will be returned as `ITestDataList`.
2. On the returned `ITestDataList`, call `getElementCount()` on the returned `ITestDataList` to find out how many elements are in the `ITestDataList`.
3. Call `getElements()` on the returned `ITestDataList`. This returns another interface: `ITestDataElementList`.
4. From the returned `ITestDataElementList,` you can now get any individual element by calling `getElement(int)`. The integer argument passed to `getElement()` is the index of the element you want (you start counting at zero). `getElement()` returns yet another interface type: `ITestDataElement`.
5. After you have an individual list item as an `ITestDataElement`, you can call `getElement()` to get actual data value. `getElement()` returns the data as an `Object`, so if you're working in Java, you need an explicit cast to a `String`.

Listing 3.9 offers two examples: one that retrieves all elements and one that retrieves selected elements. In both examples, the data is returned as an array of `Strings`.

Listing 3.9 Getting all elements from a list

Java

```
public String[] getAllListElements(TestObject testObject) {
      String[] all = null;
      ITestDataList iList = (ITestDataList)
                                        testObject.getTestData("list");
      int count = iList.getElementCount();
      all = new String[count];
      ITestDataElementList iElementList = iList.getElements();
      for (int i = 0; i < count; i++) {
         ITestDataElement iElement = iElementList.getElement(i);
         String value = iElement.getElement().toString();
         all[i] = value;
      }
      return all;
}
```

VB.NET
```
Public Function getAllListElements(ByVal testObject As TestObject) _
  As String()
    Dim iData As ITestDataList = testObject.GetTestData("list")
    Dim count As Integer = iData.GetElementCount()
    Dim all(count - 1) As String
    Dim iElementList As ITestDataElementList = iData.GetElements()

    For i As Integer = 0 To count - 1
       Dim iElement As ITestDataElement = iElementList.GetElement(i)
       Dim value As String = iElement.GetElement().ToString()
       all(i) = value
    Next

    Return all
End Function
```

Getting selected elements from a list

Java
```
public String[] getSelectedElements(TestObject testObject) {
      String[] selected = null;
      ITestData iData = testObject.getTestData("selected");
      if (iData instanceof ITestDataList) {
        ITestDataList iList = (ITestDataList) iData;
        int count = iList.getElementCount();
        selected = new String[count];
        ITestDataElementList iElementList = iList.getElements();
        for (int i = 0; i < count; i++) {
          ITestDataElement iElement = iElementList.getElement(i);
          String value = iElement.getElement().toString();
          selected[i] = value;
        }
       } else if (iData instanceof ITestDataText) {
           ITestDataText iText = (ITestDataText) iData;
           selected = new String[0];
           selected[0] = iText.getText();
      }
return selected;
}
```

VB.NET

```
Public Function getSelectedElements(ByVal TestObject As TestObject) As String()
   Dim selected() As String = Nothing
   Dim iData As ITestData = TestObject.GetTestData("selected")

   If (TypeOf iData Is ITestDataList) Then
     Dim iList As ITestDataList = iData
     Dim count As Integer = iList.GetElementCount()
     ReDim selected(count - 1)
     Dim iElementList As ITestDataElementList = iList.GetElements()
     For i As Integer = 0 To count - 1
       Dim iElement As ITestDataElement = iElementList.GetElement(i)
       Dim value As String = iElement.GetElement()
       selected(i) = value
     Next

   ElseIf (TypeOf iData Is ITestDataText) Then
     Dim iText As ITestDataText = iData
     ReDim selected(1)
     Dim value As String = iText.GetText
     selected(0) = value
   End If

   getSelectedElements = selected
End Function
```

The last two script examples in Listing 3.9 illustrate a technique for handling the special case of `getTestData()` returning `ITestDataText` and not `ITestDataList`.

Using getTestData() to Read Data in a Table

As opposed to lists, which require several method calls (and three interfaces) to get list values, tabular table is straightforward; you use only one interface. The initial learning challenge is the test data type arguments. There are quite a few; some are test object domain-specific. (See Tables 3.3, 3.4, and 3.5.)

Table 3.3 HTML Tables

Test Data Type	Return Type	Comments	Provides Access to Column and Row Header Data?
Contents	`ITestData Table`	Returns all table data, including hidden rows/columns.	Yes
Grid	`ITestData Table`	Returns all table data, including hidden rows/columns.	No
visiblecontents	`ITestData Table`	Returns all data displayed in the table.	Yes
visiblegrid	`ITestData Table`	Returns all data displayed in the table.	No
Text	`ItestData Text`	Returns the entire contents of the tables as a single string.	No

Table 3.4 Swing Tables

Test Data Type	Return Type	Comments	Provides Access to Colum and Row Header Data?
contents	`ITestData Table`	Returns all data, including hidden rows/columns.	Yes
visible contents	`ITestData Table`	Returns all data displayed in the table.	Yes
selected	`ITestData Table`	Returns all selected data, includ-ing hidden rows/columns.	Yes
visible selected	`ITestData Table`	Returns all displayed selected data.	Yes

Table 3.6 lists the most commonly used methods in `ITestDataTable`.

Table 3.5 Microsoft .NET Grids

Test Data	Return Type	Comments	Provides Access to Colum and Row Header Data?
viewcontents	`ITestDataTable`	Returns all data displayed in the table.	Yes
Sourcecontents	`ITestDataTable`	Returns all data in the table source, i.e., includes rows/columns not displayed in the table.	Yes
Viewselectedrow	`ITestDataTable`	Returns all data displayed in the selected row(s).	Yes
Sourceselectedrow	`ITestDataTable`	Returns all data in table source, i.e., includes rows/columns not displayed in the table.	Yes
Viewcurrentrow	`ITestDataTable`	Returns all data displayed in the currently active row. i.e., the row with focus.	Yes
Sourcecurrentrow	`ITestDataTable`	Returns all data in the source of the currently active row.	Yes

Table 3.6 Most Commonly Used `ITestDataTable` Methods

Method	Comments
`int getRowCount()`	Returns the number of rows in the data returned.
`int getColumnCount()`	Returns the number of columns in the data returned.
`Object getColumnHeader(int)`	Returns the value of the column title at the specific index. Because the data is returned as Object, you typically cast to a String.
`Object getRowHeader(int)`	Returns the value of the row header at the specific index. Because the data is returned as Object, you typically cast to a String.
`Object getCell (int, int)`	Returns the data in the row, column pair passed. Because the data is returned as Object, you typically cast to a String.

Listing 3.10 provides two examples that print to the console (Output window in Visual Studio) all the test data types available in the (table) test object passed. This method (function in VB.NET) can be used at design time to compare what is returned in each case and to help you determine the appropriate test data type argument for a table you are coding against.

Listing 3.10 Printing all of a table's test data types

Java

```
public void printTableData(TestObject table) {

   Enumeration<String> testDataTypes = table.getTestDataTypes()
                                                               .keys();
   while (testDataTypes.hasMoreElements()) {
      String testDataType = testDataTypes.nextElement();
      System.out.println(testDataType);
      ITestData iData = table.getTestData(testDataType);
      if (iData instanceof ITestDataTable) {
         ITestDataTable iTableData = (ITestDataTable) table
                                   .getTestData(testDataType);
         int rows = iTableData.getRowCount();
         int cols = iTableData.getColumnCount();
         for (int col = 0; col < cols; col++) {
            System.out.print(iTableData.getColumnHeader(col));
            System.out.print("\t\t");
         }
         System.out.print("\n");
         for (int row = 0; row < rows; row++) {
            for (int col = 0; col < cols; col++) {
               System.out.print(iTableData.getCell(row, col));
               System.out.print("\t\t");
            }
            System.out.print("\n\n");
         }
         System.out.print("\n");
      } else if ( iData instanceof ITestDataText ) {
          ITestDataText iText = (ITestDataText) iData;
          String text = iText.getText();
    System.out.println(text + "\n\n" );
 }
    }
  }
```

VB.NET

```
Public Sub printTableData(ByVal table As TestObject)
   Dim testDataTypes As System.Collections.ICollection = _
                                          table.GetTestDataTypes.Keys
   For Each testDataType As String In testDataTypes
      Dim iData As ITestData = table.GetTestData(testDataType)
      Console.WriteLine(testDataType)
      If TypeOf iData Is ITestDataTable Then
         Dim iTableData As ITestDataTable = CType(iData, ITestDataTable)
         Dim rows As Integer = iTableData.GetRowCount()
         Dim cols As Integer = iTableData.GetColumnCount()

         ' First print column headers if available
         Try
            For col As Integer = 0 To cols
               Console.Write(iTableData.GetColumnHeader(col))
               Console.Write(Chr(9) + Chr(9))
            Next
            Console.Write(Chr(13))
         Catch ex As Exception
            Console.WriteLine("COLUMN HEADERS NOT AVAILABLE")
         End Try
         For row As Integer = 0 To rows
            For col As Integer = 0 To cols
               Console.Write(iTableData.GetCell(row, col))
               Console.Write(Chr(9) + Chr(9))
            Next
            Console.Write(Chr(13))
         Next
         Console.Write(Chr(13))
      ElseIf (TypeOf iData Is ITestDataText) Then
         Dim iText As ITestDataText = CType(iData, ITestDataText)
         Dim text As String = iText.GetText()
         Console.WriteLine( text + chr(13) + chr(13) )
      End If
   Next
End Sub
```

Referring to the previous code samples, note the following. First, you want to note that the target (control) of the code is a Swing table. To get all test data types applicable to the table, you call `getTestDataTypes`. The `ITestDataTable` object (returned by the call to `getTestData()`) can be used to determine the number of rows and columns in the table by calling `getRowCount()` and `getColumnCount()`. If column headers are not available, an exception can be thrown; you therefore use a try block to prevent the script from failing. To retrieve the value in any particular cell, call `getCell(row, col)`, passing in the row and column index of the cell you're interested in (start counting at 0). `getCell()` returns the value as an object, which you almost always downcast to a string. In Swing tables, the column and row headers are separate Swing objects from the table object. The data in these headers, however, can be retrieved via methods in the `ITestDataTable` object returned by calling `getTestData()` on the table. In other words, your scripts do not need to worry about having (or getting) separate column or row header `TestObjects`. After you have the `ITestDataTable` in the table, call `getColumnHeader()` and `getRowHeader()` methods, respectively.

Using getTestData() to Extract Data from a Tree

From a test data type perspective, trees are simple. There are only two test data types: tree and selected.

Table 3.7 Tree Object Data Types

Data Type	Return Type	Comments
`Tree`	`ItestDataTree`	Retrieves the entire tree hierarchy.
`selected`	`ItestDataTree`	Retrieves the selected tree nodes.

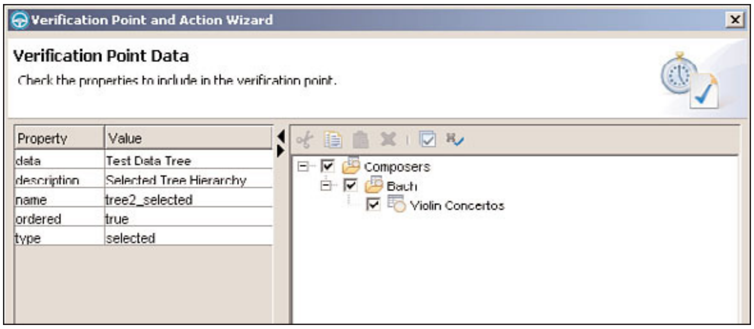
Tree Talk

There's a special vocabulary used with trees with which you might not be familiar. A tree consists of elements referred to as *nodes*. Each node can have a parent, siblings, and children. A node that has no parent is called a *root node*. A tree can have one root node (the Classics music tree) or multiple root nodes (the tree in the Rational Functional Tester Preferences dialog box). A node that has no children is called a *leaf* or *terminal node.*

In the Classics music tree, Composers is the root node. Composers has no siblings. Its children are Schubert, Haydn, Bach, Beethoven, and Mozart. Mozart has three children. Symphony No. 34 has no children and is therefore a leaf node.

Before continuing, you need to clarify what is returned with respect to *selected* tree nodes. If you click Bach's Violin Concertos in Classics, it is the selected node. You might think that in this scenario if you get selected data from the tree, just Violin Concertos will be returned. This is not the case. To see what `getTestData()` returns, insert a Data verification point on the tree with Bach's Violin Concertos selected and select **Selected tree hierarchy**. You see something similar to what is displayed in Figure 3.22.

Notice that Rational Functional Tester captures the selected element along with its ancestor nodes all the way up to the root node. If multiple nodes are selected (hold down the Ctrl key and


Verification Point and Action Wizard

Verification Point Data

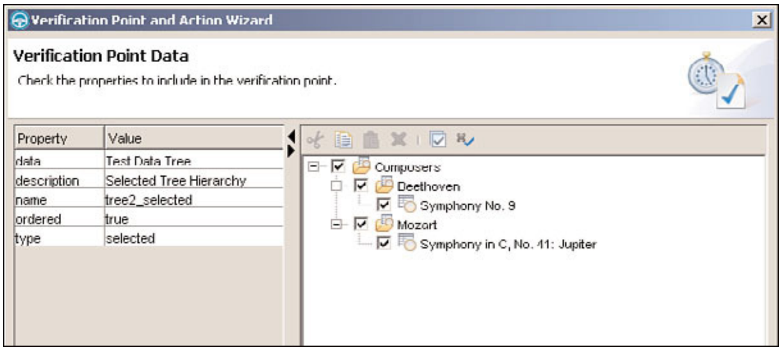
Check the properties to include in the verification point.

Property	Value
data	Test Data Tree
description	Selected Tree Hierarchy
name	tree2_selected
ordered	true
type	selected

```
☑ Composers
  ☑ Bach
    ☑ Violin Concertos
```

Figure 3.22 Data verification point—Selected tree hierarchy (single node selected)

select Symphony No. 9 and Mozart's Symphony in C, No. 41: Jupiter), the selected tree hierarchy would look like Figure 3.23.


Verification Point and Action Wizard

Verification Point Data

Check the properties to include in the verification point.

Property	Value
data	Test Data Tree
description	Selected Tree Hierarchy
name	tree2_selected
ordered	true
type	selected

```
☑ Composers
  ☑ Beethoven
    ☑ Symphony No. 9
  ☑ Mozart
    ☑ Symphony in C, No. 41: Jupiter
```

Figure 3.23 Data verification point—Selected tree hierarchy (multiple nodes selected)

Table 3.8 lists the interfaces for accessing tree data and the commonly used methods in each interface.

Table 3.8 Interfaces and Their Methods Used for Accessing Trees

Interface	Methods
`ITestDataTree`	`ITestDataTreeNodes getTreeNodes()`
`ITestDataTree Nodes`	`int getNodeCount() int getRootNodeCount() ITestDataTree Node[] getRootNodes()`
`ITestDataTree Node`	`int getChildCount() ItestDataTreeNode[] getChildren() ItestDataTreeNode getParent() Object getNode()`

The following steps can be used as a generic template for acquiring the root nodes of a tree control.

1. Call the tree control's `getTestData()` method. The data will be returned as an `ITestDataTree interface`.
2. On the `ITestDataTree` reference, call `getTreeNodes()`. This returns the data as an `ITestDataTreeNodes` interface.

   Note that if you want to get to the selected leaf nodes in the tree, you need to start at the top of the tree and traverse down to them (such as using some sort of loop). To get to the top of a tree, call `getRootNodes()`, which returns an array of individual `ITestDataTreeNodes`. It returns an array because depending on the tree, there can be more than one root node.

   Also note that after you have the data contained in an `ITestDataTreeNodes` interface, you can find out how many nodes are in the data returned by calling `getNodeCount()`. It's important to note that `getNodeCount()` doesn't necessarily return the number of nodes in the test object; it returns the number of nodes in the data returned by the initial call to `getTestData()`. If you pass `getTestData()` an argument of tree, it returns the total number of nodes in the tree (for example, the Composer tree in Classics returns 20 nodes). If you pass an argument of selected, it returns the number of nodes in the selected tree hierarchy. If Bach Violin Concertos is selected, it returns three (because the nodes returned are **Composers > Bach > Violin Concertos**).
3. As listed in Table 3.8, there are three key methods you can invoke on an `ITestDataTreeNode`. `getNode()` returns the node data as an Object. Thus, if you call `getNode()` on the first (and only) root node and cast that to a string, you can print it out (or more likely store it in a variable for further processing).

Listing 3.11 provides two examples—one Java and one VB.NET—that exemplify the previous template steps.

Listing 3.11 Printing a tree's root nodes

Java
```
public void printRootNodes( TestObject tree ) {
   String selectedNode = null;

   ITestDataTree iTreeData =
(ITestDataTree)tree.getTestData("selected");
   ITestDataTreeNodes iNodes = iTreeData.getTreeNodes();
   ITestDataTreeNode[] rootNodes = iNodes.getRootNodes();
   for(int i = 0; i < rootNodes.length; i++) {
        String nodeData = rootNodes[i].getNode().toString();
        System.out.println( nodeData );
    }
}
```

VB.NET

```
Public Sub printRootNodes(ByVal tree As TestObject)
    Dim selectedNode As String = Nothing
    Dim iTreeData As ITestDataTree = tree.GetTestData("tree")
    Dim iNodes As ITestDataTreeNodes = iTreeData.GetTreeNodes()

    Dim rootNodes As ITestDataTreeNode() = iNodes.GetRootNodes()
    For i As Integer = 0 To rootNodes.length - 1
        Dim nodeData As String = rootNodes(i).GetNode.ToString
        Console.WriteLine( nodeData )
    Next
End Sub
```

The next code sample in Listing 3.12 builds off the prior steps. In particular, after you acquire the root node as an `ITestDataTreeNode`, you can start traversing it. To descend one level, call `getChildren()`. This returns the `ITestDataTreeNode`'s children (in the returned data, not the test object) as an array of `ITestDataTreeNodes`. Note that the data type of the root nodes is the same as any other node in a tree, namely `ITestDataTreeNode`. After you have the array of child `ITestDataTreeNodes`, you can acquire their data, using the `getNode()` method and casting it to a string—similar to what was accomplished in Listing 3.11.

Listing 3.12 Get the text of the selected node in a tree

Java

```
public String getSelectedTreeNode( TestObject tree ) {
    String selectedNode = null;

    ITestDataTree iTreeData =
(ITestDataTree)tree.getTestData("selected");
    ITestDataTreeNodes iNodes = iTreeData.getTreeNodes();
    int nodeCount = iNodes.getNodeCount();
    System.out.println("node count = " + nodeCount);
    if( nodeCount != 0) {
        ITestDataTreeNode[] node = iNodes.getRootNodes();
        for(int i = 0; i < nodeCount - 1; i++) {
            ITestDataTreeNode[] children = node[0].getChildren();
            node = children;
        }
        selectedNode = node[0].getNode().toString();
    }
    return selectedNode;
}
```

VB.NET

```
Public Function getSelectedTreeNode(ByVal tree As TestObject) As String
    Dim selectedNode As String = Nothing
    Dim iTreeData As ITestDataTree = tree.GetTestData("selected")
    Dim iNodes As ITestDataTreeNodes = iTreeData.GetTreeNodes()
    Dim nodeCount As Integer = iNodes.GetNodeCount()

    Console.WriteLine("Nodecount = " + nodeCount.ToString)
    If (nodeCount <> 0) Then
        Dim node As ITestDataTreeNode() = iNodes.GetRootNodes()
        For i As Integer = 1 To nodeCount - 1
            Dim children As ITestDataTreeNode() = node(0).GetChildren()
            node = children
        Next
        selectedNode = node(0).GetNode().ToString()
    End If

    Return selectedNode
End Function
```

Obtaining Data from a Test Object That the Rational Functional Tester Verification Point Wizard Does Not Capture

This large topic is discussed here and explored in more detail in other sections. Several techniques are used to deal with this challenge, the simplest of which from a coding perspective is to try to get the data into the Clipboard. If this can *reliably* be done, you can get the data into a script variable (though this technique might not be your first choice).

As discussed in the "Working with the Clipboard Object" section in this chapter, Rational Functional Tester gives you access to the system Clipboard. This provides access to the two methods used for manipulating the Clipboard: `setText()` and `getText()`. These enable you to work with test objects that Rational Functional Tester has difficulty extracting data from.

The following reference example is an open document in Microsoft Word. If you point the Rational Functional Tester Verification Point Wizard at a Microsoft Word document, it does not see the contents of the document. You can, however, get the data into a variable in your script by using the keyboard and a little code.

First, you click the document to place keyboard focus on it. Then you use the keyboard to put the data into the Clipboard (versus trying to manipulate the Word menu). The Rational Functional Tester API has two methods to work with the keyboard: `inputChars()` and `inputKeys()`. Both take string arguments, but there's a key difference between them: `inputChars()` treats every character it has passed literally, and `inputKeys()` assigns special meaning to certain characters, as shown in Table 3.9.

Table 3.9 Characters with Special Meanings in Scripts

^	Apply Ctrl to the next character.
%	Apply Alt to the next character.
+	Apply Shift to the next character.
~	Enter key.

Passing `inputKeys` ^a^c causes Rational Functional Tester to press Ctrl+A (select all) and then Ctrl+C (copy), copying the contents of the entire document to the Clipboard. After the data is copied to the Clipboard, you are home free. To get the data in the Clipboard, you call `getSystemClipboard().getText()`. This is detailed in Listing 3.13.

Listing 3.13 Using the Clipboard to get data from a test object

Java
```
microsoftWordDocumentwindow().click();
((TopLevelSubitemTestObject) microsoftWordDocumentwindow()
                        .getTopMappableParent()).inputKeys("^a^c");

String text = getSystemClipboard().getText();
System.out.println(text);
```

Creating a Custom Verification Point

In addition to the flexibility of being able to use datapool references in verification points created with the Rational Functional Tester Verification Point Wizard, you can create your own dynamic verification points in code. `RationalTestScript` (the root of all script classes) has a method, `vpManual()`, which you can use to create verification points.

`vpManual()` is used when you want your script to do all the work of verifying data. That work consists of capturing expected data, capturing actual data, comparing actual data with expected data, and logging the results of the comparison. Think of *manual* as referring to manual coding.

This discussion begins with the first signature (which you will likely use most often). In `vpManual`'s three-argument version, you supply a name for your verification point along with the baseline and actual data associated with the verification point. `vpManual()` then creates and returns a reference to an object; specifically, one that implements Rational Functional Tester's `IFtVerificationPoint` interface. The verification point metadata (name, expected, and actual data) are stored in the returned `IFtVerificationPoint`.

```
IFtVerificationPoint myVP = vpManual( "FirstName", "Sasha", "Pasha");
Dim myVP as IFtVerificationPoint = vpManual( "FirstName", "Sasha", "Pasha" )
```

There are a couple of items to note about using `vpManual()`:

- **vpName—**The verification point name must be a script-unique valid Java (or .net) method name and be less than 30 (.net) or 75 (Java) characters.
- **Baseline and Actual data—**The compiler accepts a reference to anything that inherits from `Object` (which means that in .NET, any argument you pass is acceptable; Java allows anything other than primitives, such as int, bool, and so on); however, you need to satisfy more than the compiler. To automate the comparison of baseline with actual data, you need to pass data types that Rational Functional Tester knows how to compare (you don't want to have to build your own compare method). This limits you to passing value classes. Some examples of legal value classes are: any data returned by `getTestData()`, strings, primitives, wrapper classes (`Integer`, `Boolean`, and so on), common classes that consist of value class fields (for example, `Background`, `Color`, `Bounds`, `ITestData`, meaning, anything returned by `getTestData()`), and arrays (one and two-dimensional), vectors, and hashtables that contain value class elements.

What do you do with the `IFtVerificationPoint` that `vpManual()` returns? In the simplest case, you call `performTest()` and get on with things. `performTest()` compares the baseline with the actual and logs the results (`boolean`) of the comparison. See Listing 3.14.

Listing 3.14 A simple comparison

Java

```
IFtVerificationPoint myVP = vpManual( "FirstName", "Sasha", "Pasha");
boolean passed = myVP.performTest();
```

VB.NET

```
Dim myVP As IFtVerificationPoint = VpManual("FirstName", "Sasha", _ "Pasha")
Dim passed As Boolean = myVP.PerformTest
```

In two lines of code, you have done quite a bit. You created a verification point and compared and logged the results of comparing the baseline to the actual data. It's common to combine these two statements into one:

```
vpManual( "FirstName", "Minsk", "Pinsk").performTest();
```

You use this style when the only method you need to invoke on the `IFtVerificationPoint` returned by `vpManual()` is `performTest()`.

It's important to note that the three-argument version of `vpManual()` does not persist baseline data to the file system for future runs. It's also important to stress the importance of the uniqueness of the name in the script.

To illustrate how Rational Functional Tester behaves when a verification point is not unique, consider the simple example where `vpManual` is called in a loop (demonstrated in Listing 3.15). The loop in each code sample simply compares two numbers. To introduce some variety, you force the actual value to equal the baseline value only when the baseline value is even.

Listing 3.15 Consequences of a nonunique verification point name

Java

```
for(int baseline = 1; baseline <= 10; baseline++ ) {
   int actual = baseline % 2 == 0 ? baseline : baseline + 1;
   vpManual("CompareNumbers", baseline, actual).performTest();
}
```

VB.NET

```
For baseline As Integer = 1 To 10
   Dim actual As Integer
   If (baseline Mod 2 = 0) Then
      actual = actual
   Else
      actual = baseline + 1
   End If
   VpManual("CompareNumbers", baseline, actual).PerformTest()

Next
```

If you execute this code, you see two interesting results in the log:

- The pass/fail status for each verification point is what's expected (half pass, half fail).
- The comparator shows the correct actual values for each verification point, but a baseline value of 1 for every verification point. The reason for this is that after an `IFtVerificationPoint` has been created, the baseline cannot be updated.

The common technique to deal with this issue (in a looping context) is to append a counter to the verification point name, guaranteeing a unique name per iteration. This is shown in Listing 3.16.

Listing 3.16 Guaranteeing a unique verification point name

Java

```
for(int baseline = 1; baseline <= 10; baseline++ ) {
   int actual = baseline % 2 == 0 ? baseline : baseline + 1;
   vpManual("CompareNumbers_" + baseline, baseline,
                                                   actual).performTest();
}
```

VB.NET

```
For baseline As Integer = 1 To 10
   Dim actual As Integer
   If (baseline Mod 2 = 0) Then
      actual = actual
   Else
      actual = baseline + 1
   End If
   VpManual("CompareNumbers_" & baseline, _
                               baseline, actual).PerformTest()

Next
```

Persisting Baseline Data

In addition to the three-argument version of `vpManual()`, there is a two-argument version of `vpManual()`:

```
IFtVerificationPoint vpManual( String vpName, Object data )
```

The two-argument version is used when you want to persist the baseline data to the Rational Functional Tester project. Here's how it works. The first time `performTest()` is called on an `IFtVerificationPoint` with a given name (the name passed to `vpManual()`), no comparison is done. The baseline data is written to the RFT project and a verification point displays in the Script Explorer (and an informational message is written to the log). With each *subsequent* execution of `performTest()` on an `IFtVerificationPoint` with the same name, the data argument passed to `vpManual()` is treated as actual data, and `performTest()` executes the comparison, logging the result.

Changing the Value of a Test Object's Property

Properties cannot only be read using `getProperty()`, but they can be changed using `setProperty()`:

```
public void setProperty( String propertyName, Object propertyValue )
```

Although you most likely will not use `setProperty()` nearly as often as you use `getProperty()`, it is a method worth knowing about. `SetProperty()` takes two arguments: the property to change and the value to change the property to.

The reference example is one that involves setting data field values in test objects (for example, text fields). In general, you should use `inputKeys()` or `inputChars()` to enter data into the SUT. With some cases, however, this becomes challenging. One such context is internationalization testing. `inputKeys()` and `inputChars()` can enter characters only in the current keyboard's character set. If the current keyboard is set to English, for example, RFT throws a `StringNotInCodePageException` if your script attempts to enter any nonEnglish characters.

One potentially viable solution is to use `setProperty()` instead of `inputKeys()` to set the field value. The first step is to determine the property you need to set. Manually set a value, and then examine the test object using either the Inspector or the Verification Point and Action Wizard. Search for a property whose value is the data value you entered. If you enter a search term of Pasta Norma in a Google search field and examine the field with the Inspector, you see two properties whose values are Pasta Norma: `value` and `.value`. This is not uncommon: It's possible that the data value is represented by more than one property. It's a good idea to note all these property names because some might be read-only. If you try to set a property value that's read-only, Rational Functional Tester throws an exception.

If you had a datapool with different search strings in different character sets, you can manipulate the scripts to perform multiple searches, as shown in Listing 3.17.

Listing 3.17 Using `SetProperty()` to set data in a test object

Java

```
while (!dpDone()) {
   text_q().setProperty(".value", dpString("SearchItem"));
   // Do what we need to do
   dpNext();
}
```

VB.NET

```
Do until(dpDone())
   text_q.SetProperty(".value", dpString("SearchItem"))
   ' Do what we need to do
   dpNext()
loop
```

Why Is InputKeys() Preferred?

To illustrate why `inputKeys()` is the preferred method to enter data into objects, test what happens if you set the quantity of CDs to buy in the Classics sample application:

```
quantityText().setProperty("Text", "4");
```

You see something odd happen (or, *not happen* in this case): The total amount is not updated to reflect the new value. The reason for this is that the total amount is updated when the `inputKeys` event is fired. `setProperty()` does not cause this event to fire and is therefore not a possible technique to set the quantity field.

Evolving Custom Scripting into Reusable Methods

Up to this point, you have learned how to make different types of enhancements to your scripts, making them more robust. You saw how to add delays, introduce timers, work with the Windows Clipboard, manipulate different objects you recorded (such as test objects and verification points), and so on. Rational Functional Tester enables you to expand these techniques by turning them into generic, reusable methods.

Methods that are reusable and, ideally generic, enable you to apply them to more than one object or script. You can often use them across your entire test project. This enables you to create methods that benefit everybody on your team, saving time on your overall test automation effort.

The first thing that is involved with creating custom methods is setting up the signature. Basically, you specify the type of data your method needs to receive and the type of data it needs to send back when it's done executing. You can work with the typical data types and objects that the Java and VB.NET languages work with. For example, you might have an add method with a signature that looks like the following:

```
public int add(int i, int j)
```

The VB.NET equivalent is:

```
Public Function add(ByVal i As Integer, ByVal j As Integer) As Integer
```

In both instances, you are merely passing two integers into the method (function in VB.NET), returning the sum of them as an integer.

Rational Functional Tester enables you to pass and return test objects. This gives you the ability to create custom methods for the objects that you capture in your test object map. A simple instance of this might be a method to test if an object is enabled or not. The method signature would like the following Java example:

```
public Boolean isEnabled(TestObject myGuiObject)
```

In VB.NET, you see:

```
Public Function isEnabled(ByVal myGuiObject As TestObject) As Boolean
```

If you use a signature, such as the preceding examples, you can pass any test object that contains enabled property to your custom method. You can test if a button is enabled before you click it, a tree is enabled before you navigate it, a text box is enabled before typing in it, and so on.

After you have the signature defined for your method, it is a matter of providing the code that accomplishes your desired task. You might write reusable methods that wait for test object properties to change, that acquire data from test objects, and set properties for test objects. Regardless of the methods that you create, you can organize and package them into *Helper Superclasses*, calling them from your scripts.

Helper Superclasses (also called Helper Base Classes in the VB.NET version of the tool) are an excellent way for you to create a simple, project-wide means for storing your reusable methods. You can easily create a Helper Superclass in Rational Functional Tester by performing the following steps:

1. Click **File > New > Helper Superclass** (this launches the Create Script Helper Superclass Wizard).
2. Select the location to store your Helper Superclass.
3. Provide a name for your Helper Superclass.
4. Click the **Finish** button.

If you are using the VB.NET version, the steps are:

1. Click **File > New > Add Helper Base Class** (this will launch the Add New Item Wizard, preselecting the Script Helper Base Class template).
2. Provide a name for your Helper Base Class.
3. Provide a location for your Helper Base Class.
4. Click the **Add** button.

In either case, you end up with an empty class, prebuilt to extend the default capabilities of Rational Functional Tester's **RationalTestScript** class. This is where you start constructing your custom, reusable methods. The following code samples, contained in Listing 3.18, show a Java and a .NET helper class. These contain the following sample methods from prior sections:

- `printValueProperties()`—"Retrieving All Properties of a Test Object" section
- `getAllListElements()`—"Using getTestData to Extract Data from a List" section
- `printRootNodes()`—"Using getTestData to Extract Data from a Tree" section

Listing 3.18 Helper Classes

Java

```
package HelperSuperclasses;

import java.util.Enumeration;
import java.util.Hashtable;

import com.rational.test.ft.object.interfaces.TestObject;
import com.rational.test.ft.script.RationalTestScript;
import com.rational.test.ft.vp.ITestDataElement;
import com.rational.test.ft.vp.ITestDataElementList;
import com.rational.test.ft.vp.ITestDataList;
import com.rational.test.ft.vp.ITestDataTree;
import com.rational.test.ft.vp.ITestDataTreeNode;
import com.rational.test.ft.vp.ITestDataTreeNodes;

public abstract class MyHelperSuperclass extends RationalTestScript
{
      public void printValueProperties(TestObject testObject) {

            Hashtable valueProperties = testObject.getProperties();
            Enumeration valuePropNames = valueProperties.keys();
            Enumeration valuePropValues = valueProperties.elements();
            while (valuePropNames.hasMoreElements()) {
                  System.out.println(valuePropNames.nextElement() + ":"
                              + valuePropValues.nextElement());
            }
      }

      public String[] getAllListElements(TestObject testObject) {
            String[] all = null;
            ITestDataList iList = (ITestDataList)
            testObject.getTestData("list");
            int count = iList.getElementCount();
            all = new String[count];
            ITestDataElementList iElementList = iList.getElements();
            for (int i = 0; i < count; i++) {
                  ITestDataElement iElement = iElementList.getElement(i);
                  String value = iElement.getElement().toString();
                  all[i] = value;
            }
            return all;
      }

      public void printRootNodes( TestObject tree ) {
            String selectedNode = null;

            ITestDataTree iTreeData =
                  (ITestDataTree)tree.getTestData("selected");
            ITestDataTreeNodes iNodes = iTreeData.getTreeNodes();
            ITestDataTreeNode[] rootNodes = iNodes.getRootNodes();
            for(int i = 0; i < rootNodes.length; i++) {
                  String nodeData = rootNodes[i].getNode().toString();
                  System.out.println( nodeData );
            }
      }
}
```

VB.NET

```
Imports Rational.Test.Ft.Script
Imports Rational.Test.Ft.Vp
Imports System
Imports System.Collections
Imports Rational.Test.Ft.Object.Interfaces

Namespace ScriptHelperBaseClasses

    Public MustInherit Class MyScriptHelperBaseClass
        Inherits RationalTestScript

        Public Sub printValueProperties(ByVal testobject As TestObject)
           Dim valueProperties As Hashtable = testobject.GetProperties
           Dim valuePropNames As ICollection = valueProperties.Keys
           Dim valuePropValues As ICollection = valueProperties.Values

           Dim enumPropNames As IEnumerator = valuePropNames.GetEnumerator()
           Dim enumPropVals As IEnumerator = valuePropValues.GetEnumerator()

           Do While enumPropNames.MoveNext And enumPropVals.MoveNext
               Dim currentProp As String = enumPropNames.Current
               Dim currentPropVal As Object = enumPropVals.Current
               If TypeOf currentPropVal Is Object Then
                   If Not (TypeOf currentPropVal Is String) Then
                       currentPropVal = currentPropVal.ToString
```

```
                End If
            End If
            Console.WriteLine(currentProp + ":" + currentPropVal)
        Loop
    End Sub

    Public Function getAllListElements(ByVal testObject As TestObject) As
String()
        Dim iData As ITestDataList = testObject.GetTestData("list")
        Dim count As Integer = iData.GetElementCount()
        Dim all(count - 1) As String
        Dim iElementList As ITestDataElementList = iData.GetElements()

        For i As Integer = 0 To count - 1
            Dim iElement As ITestDataElement = iElementList.GetElement(i)
            Dim value As String = iElement.GetElement().ToString()
            all(i) = value
        Next

        Return all
    End Function

    Public Sub printRootNodes(ByVal tree As TestObject)
        Dim selectedNode As String = Nothing
        Dim iTreeData As ITestDataTree = tree.GetTestData("tree")
        Dim iNodes As ITestDataTreeNodes = iTreeData.GetTreeNodes()

        Dim rootNodes As ITestDataTreeNode() = iNodes.GetRootNodes()
        For i As Integer = 0 To rootNodes.length - 1
            Dim nodeData As String = rootNodes(i).GetNode.ToString
            Console.WriteLine(nodeData)
        Next
    End Sub

  End Class
End Namespace
```

Please note that you might need to import certain packages and namespaces (depending on whether you are using Java or VB.NET). You receive errors from the compiler that let you know that certain classes, structures, objects, and so on do not exist. In the Java version of the tool, you can usually perform a **<Ctrl> + O** keystroke combination. This imports the necessary packages. There might be instances where you have to review the error messages to get an idea of what package needs to be imported. The VB.NET version of the tool requires you to import the necessary namespace.

After you have your Helper Superclass set up, creating the desired reusable methods, you need to perform two more steps. The first is to tell your test scripts to use the new class. In other words, you want to tell your scripts about the new methods that you created. The second step is to actually call your custom methods from your script.

You can associate your new Helper Superclass either at the individual script level or at the project level. Associating your Helper Superclass at the project level causes any new script to be aware of your custom methods. If you already have scripts created, you need to associate your Helper Superclass to each of them. To associate your Superclass to your project, you do the following:

1. Right-click your project node in the Functional Test Projects view (Solution Explorer, if you're using Rational Functional .NET).
2. Click **Properties**.
3. Click **Functional Test Project** (on the left-hand side of the Properties window).
4. Click the **Browse** button next to the Script Helper Superclass property (Script Helper Base Class property if you are using Rational Functional Tester .NET).
5. Select your created Helper Superclass (Helper Base Class).

   Note: You might need to type the first few letters of the name of your Super Helper class to get it to display in the list when using the Java version of Rational Functional Tester.
6. Click the **OK** button.

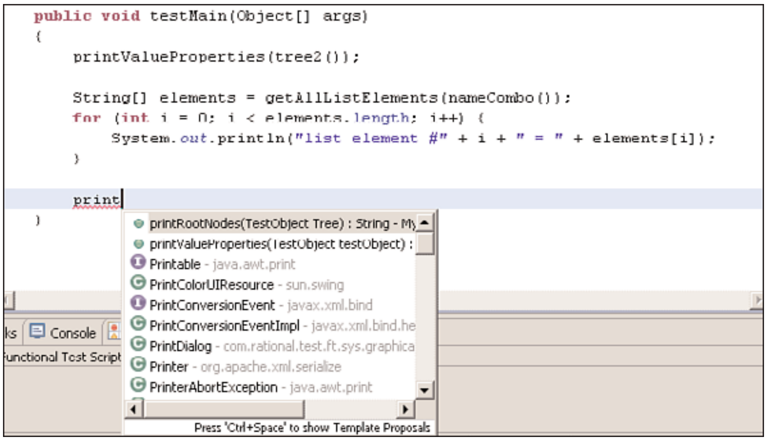
Associating your Helper Superclass to individual scripts is a similar process. You need only to perform the following steps:

1. Right-click an existing script in your project node in the Functional Test Projects view (Solution Explorer, if you are using Rational Functional .NET).
2. Click **Properties**.
3. Click **Functional Test Script** (on the left-hand side of the "Properties" window).
4. Click the **Browse** button next to the Helper Superclass property.
5. Click the created Helper Superclass (Helper Base Class).

   Note: You might need to type the first few letters of the name of your Super Helper class to get it to appear in the list when using the Java version of Rational Functional Tester.
6. Click the **OK** button.

After you make the necessary Helper Superclass associations, the last thing you need to do is to actually use the custom work that you created. Referring to the methods and functions in Listings 3.17 and 3.18, you simply need to call the specific custom method(s) that you need.

Figures 3.24 and 3.25 show scripts that make calls to the custom methods contained in the Helper Superclass. This should give you an idea of how your custom work can save time for the other members on your team. They do not have to reinvent the wheel you already created. They can simply reuse the code that you built and use the method(s) that you placed in a Helper Superclass.



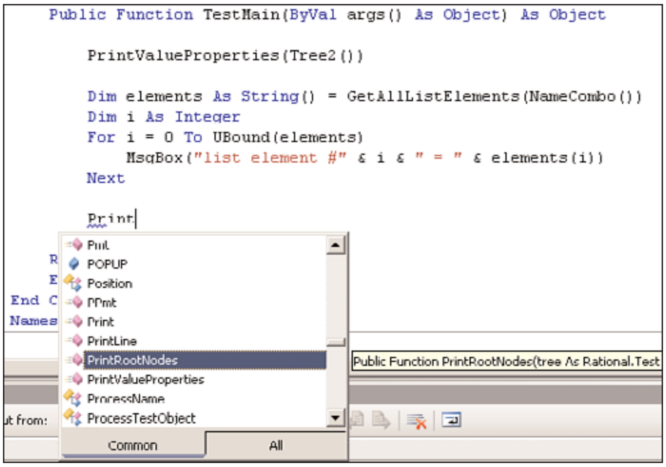
```
public void testMain(Object[] args)
{
    printValueProperties(tree2());

    String[] elements = getAllListElements(nameCombo());
    for (int i = 0; i < elements.length; i++) {
        System.out.println("list element #" + i + " = " + elements[i]);
    }

    print
}
```

```
printRootNodes(TestObject Tree) : String - My
printValueProperties(TestObject testObject) :
Printable - java.awt.print
PrintColorUIResource - sun.swing
PrintConversionEvent - javax.xml.bind
PrintConversionEventImpl - javax.xml.bind.he
PrintDialog - com.rational.test.ft.sys.graphica
Printer - org.apache.xml.serialize
PrinterAbortException - java.awt.print
Press 'Ctrl+Space' to show Template Proposals
```

Figure 3.24 Calling custom methods in Helper Superclass—Java



```
Public Function TestMain(ByVal args() As Object) As Object

    PrintValueProperties(Tree2())

    Dim elements As String() = GetAllListElements(NameCombo())
    Dim i As Integer
    For i = 0 To UBound(elements)
        MsgBox("list element #" & i & " = " & elements(i))
    Next

    Print
```

```
Pnl
POPUP
Position
PPmt
Print
PrintLine
PrintRootNodes
PrintValueProperties
ProcessName
ProcessTestObject
```

Public Function PrintRootNodes(tree As Rational.Test

Figure 3.25 Calling custom functions in Helper Base Class—VB.NET

Summary

This chapter provides you with the knowledge necessary to expand your skill set outside of the realm of record and playback test automation. You should now have the knowledge necessary to handle such tasks as synchronizing your script playback with any latency issues your application might face, acquiring data from the Windows Clipboard, manipulating data in test objects, and so on. Further, you should now possess the ability to turn your custom scripting into methods that can be reused across your project.

You should also note that this chapter provides some building blocks for more advanced code-oriented testing. The ability to create reusable methods is a commonplace task for many scripting activities you might find yourself performing in Rational Functional Tester. Further, you might find that creating custom verification points is more valuable, at times, than using the built-in ones. In any case, you should now have the level of comfort to move onto more advanced scripting tasks.

Index