Hello World: Rational Functional Tester

Create robust, reusable automated functional tests

Skill Level: Intermediate

Dennis Schultz (dennis.schultz@us.ibm.com)
Marketing Engineer
IBM

24 Oct 2005

Step through a few practical exercises to get a feel for how IBM® Rational® Functional Tester lets you create robust, automated functional tests. Learn how to record automated functional tests, enhance the tests through Java™ scripting, and playback those tests as part of a functional regression test suite. This is the fifth in a series of Hello World tutorials, which provide high-level overviews of the key IBM software products.

Section 1. Before you start

About this series

This series is for novices who want high-level overviews of IBM software products. The modules are designed to introduce the products and draw your interest for further exploration. The exercises only cover the basic concepts, but are enough to get you started.

About this tutorial

This tutorial provides a high-level overview of IBM Rational Functional Tester and gives you the opportunity to practice using it with hands-on exercises. Step-by-step instructions show you how to record an automated functional test, enhance the test through Java scripting, and play back the test as part of a functional regression test suite.

Objectives

After completing this tutorial, you should understand the basic functions of Rational Functional Tester and be able to create automated functional regression tests as demonstrated in the exercises.

Prerequisites

This tutorial is for testers new to test automation. A general familiarity with the Java programming language is helpful, but not required.

System requirements

To run the examples in this tutorial, you'll need to install IBM Rational Functional Tester. You can download and install a free trial version of the product if you don't already own a copy. This product is available for both Microsoft® Windows® and Linux™ on x86-compatible chips; the instructions in this tutorial assume that you will be running Windows, but Linux users should be able to adapt them to that environment without difficulty.

To view the demos included in this tutorial, JavaScript must be enabled in your browser and Macromedia Flash Player 6 or higher must be installed. You can download the latest Flash Player at http://www.macromedia.com/go/getflashplayer/.

Animated demos

If this is your first encounter with a developerWorks tutorial that includes demos, here are a few things you might want to know:

- Demos are an optional way to see the same steps described in the tutorial. To see an animated demo, click the  Show me link. The demo opens in a new browser window.
- Each demo contains a navigation bar at the bottom of the screen. Use the navigation bar to pause, exit, rewind, or fast-forward portions of the demo.
- The demos are 800 by 600 pixels. If this is the maximum resolution of your screen or if your resolution is lower than this, you will have to scroll to see some areas of the demo.
- JavaScript must be enabled in your browser and Macromedia Flash Player 6 or higher must be installed.

Section 2. Getting started

IBM Rational Functional Tester, referred hereafter as *Functional Tester,* is an advanced, automated functional and regression testing tool for testers and GUI developers who need superior control for testing Java, Microsoft Visual Studio .NET, and Web-based applications.

Functional Tester provides two user interface options: Microsoft Visual Studio and Eclipse. If your development team is building applications in Visual Studio and your test teams are more comfortable with Visual Basic .NET as a scripting language, you can install Functional Tester directly into your Visual Studio IDE shell and capture your test scripts in Visual Basic .NET.

The other option you have is to install Functional Tester into the Eclipse platform -- either your own Eclipse installation or one provided by Rational as part of the Functional Tester installation package. With the Eclipse variant of Functional Tester, test scripts are captured in Java code. Either way, you have the power of a full-up environment and the versatility of a commercial standard scripting language.

Functional Tester supports the functional regression testing of many types of applications, including those built with the following technologies:

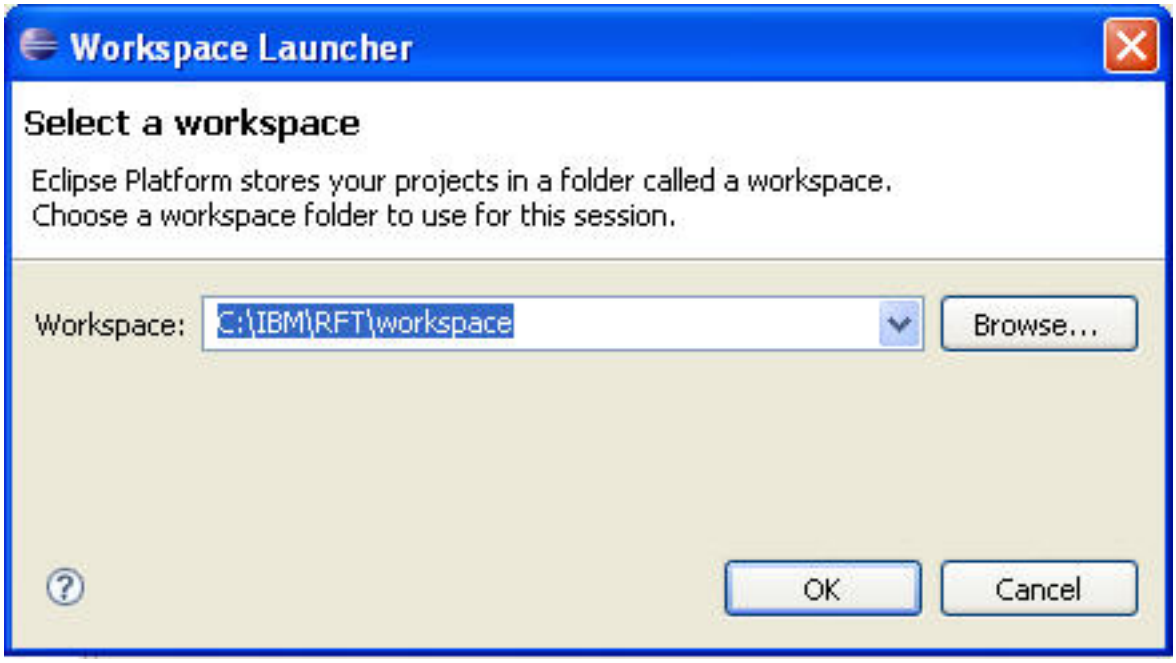
- The Java platform
- Web (HTML)
- Microsoft® .NET
- Siebel
- SAPgui
- Terminal-based applications (3270, 5250, and VT100)

All of these application types can be tested with either the Eclipse-based or Visual Studio-based variant of Functional Tester. Although the Eclipse-based variant will be used for the remainder of this tutorial, everything you do here can also be done in the Visual Studio-based variant as well.

Launching Functional Tester

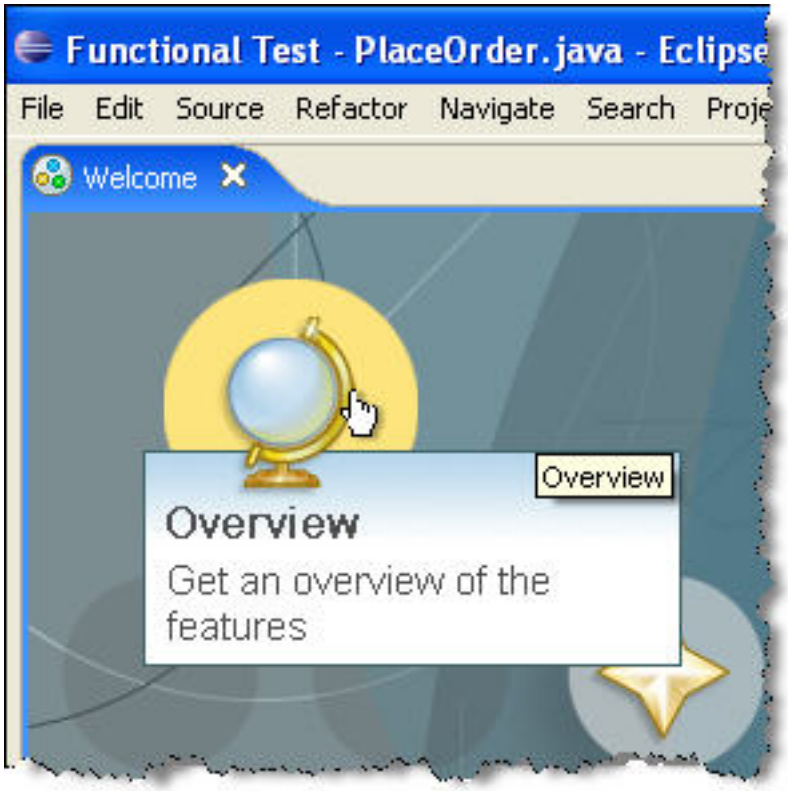
Functional Tester is started from the Windows Start menu by selecting **Programs > IBM Rational Functional Tester > IBM Rational Functional Tester > Java Scripting.** When you first start Functional Tester, you might be asked to select a workspace location, as shown in Figure 1. A *workspace* can be any directory location where your work is stored. If you are asked to choose a workspace when starting Functional Tester, you can choose the default or create a new one of your own.

Figure 1. Workspace launcher



After you have chosen a workspace, the **Welcome** page displays, as shown in Figure 2. It offers quick links to tutorials and samples. Click on the **Overview** icon to take a guided tour of Functional Tester.

Figure 2. Welcome page overview



Close the Welcome page. The Functional Test perspective displays. A *perspective* in Eclipse is a consolidation of tools and views focused on one particular task. The Functional Test perspective, as the name implies, provides views that are needed by

a developer or quality assurance professional focused on validating the functional operation of a software application.

Section 3. Preparing the test environment

This section walks you through the process of preparing your application environment for testing with Functional Tester. You will:

- Create a repository, or *project,* in which to store your Functional Tester artifacts
- Enable the application environment for testing
- Configure your application under test

Create a new functional test project

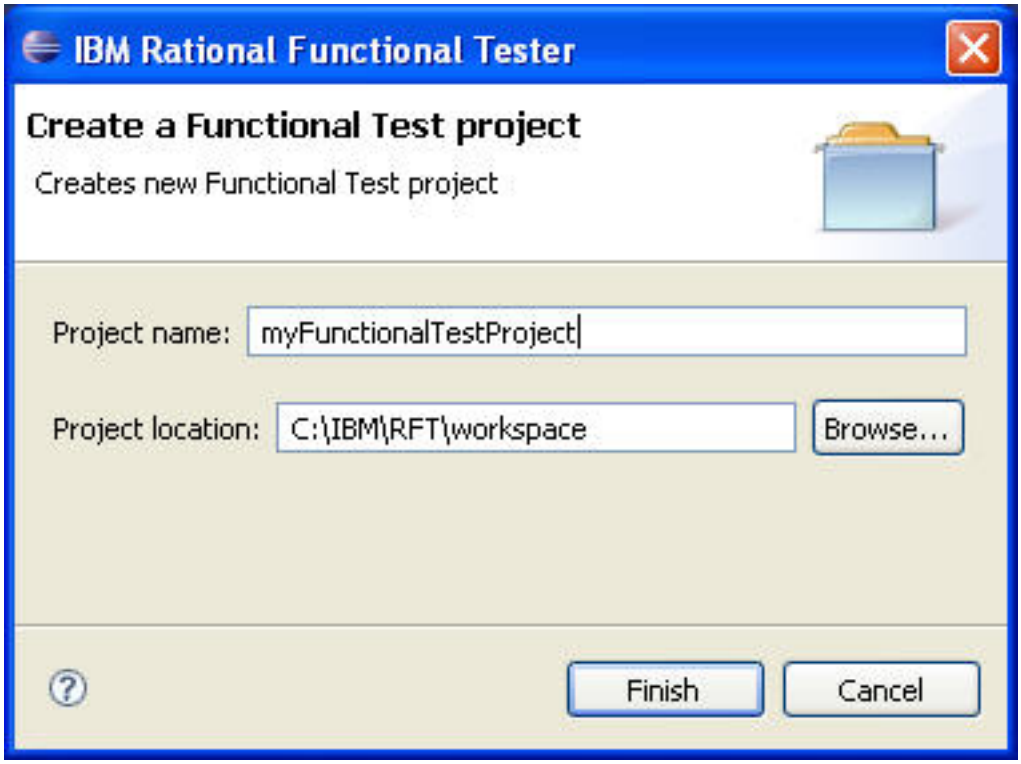
Begin by creating a Functional Test project named `MyFunctionalTestProject` to store your test scripts, datapools, and object maps.

Animated demo
Would you like to see these steps demonstrated for you?

Show me

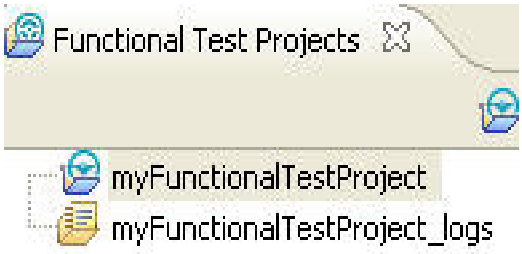
1. Start Functional Tester and select a workspace as shown in the previous section, if you haven't already done so.
2. From the workbench, select **File > New > Functional Test Project.** You should see the screen illustrated in Figure 3.
3. Enter `MyFunctionalTestProject` as the project name and click **Finish.**

Figure 3. Create Functional Test Project wizard



As you can see in Figure 4, two new entries are in your Functional Test Projects view: one for test artifacts and one for results logs.

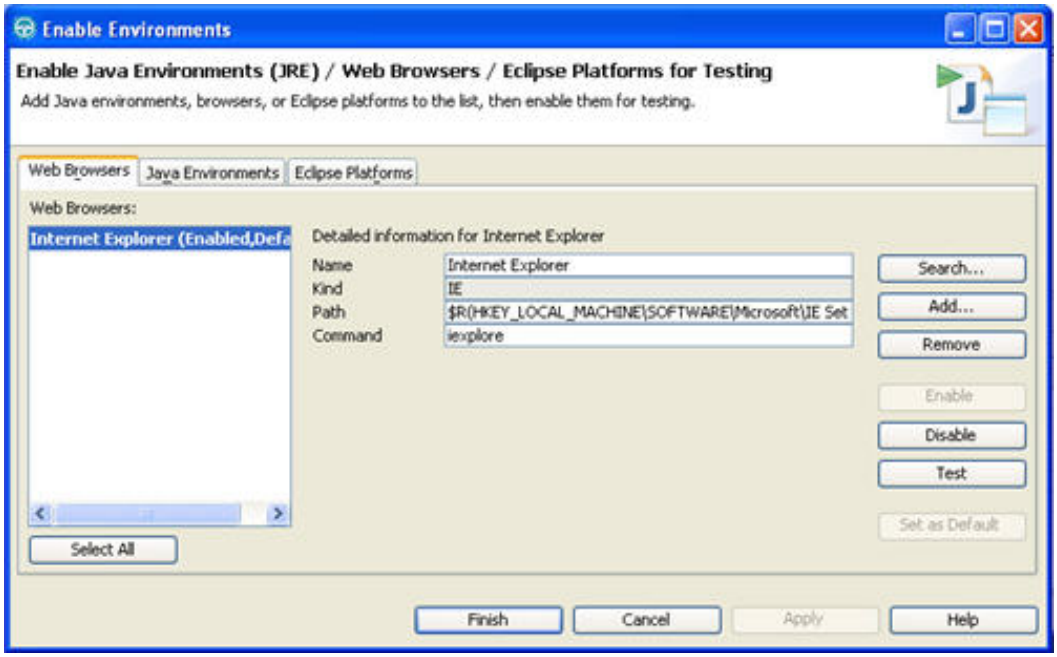
Figure 4. Functional Test Projects view



Enable the application environment

Next, you need to enable the application runtime environment. This allows Functional Tester to see inside the runtime to identify objects within the application under test.

1. Select **Configure > Enable Environments for Testing...**. You will see three tabs in the Enable Environments window, as shown in Figure 5. These are the three classes or domains that might need to be enabled, depending on the type of application you are testing. Most likely, Internet Explorer is the default browser for test playback and that it is enabled.
   Figure 5. Enable Environments window



2. Functional Tester can test Eclipse-based plug-ins and RCP applications as well. If you have additional Eclipse platforms installed, you can enable them for testing on the Eclipse Platforms tab.

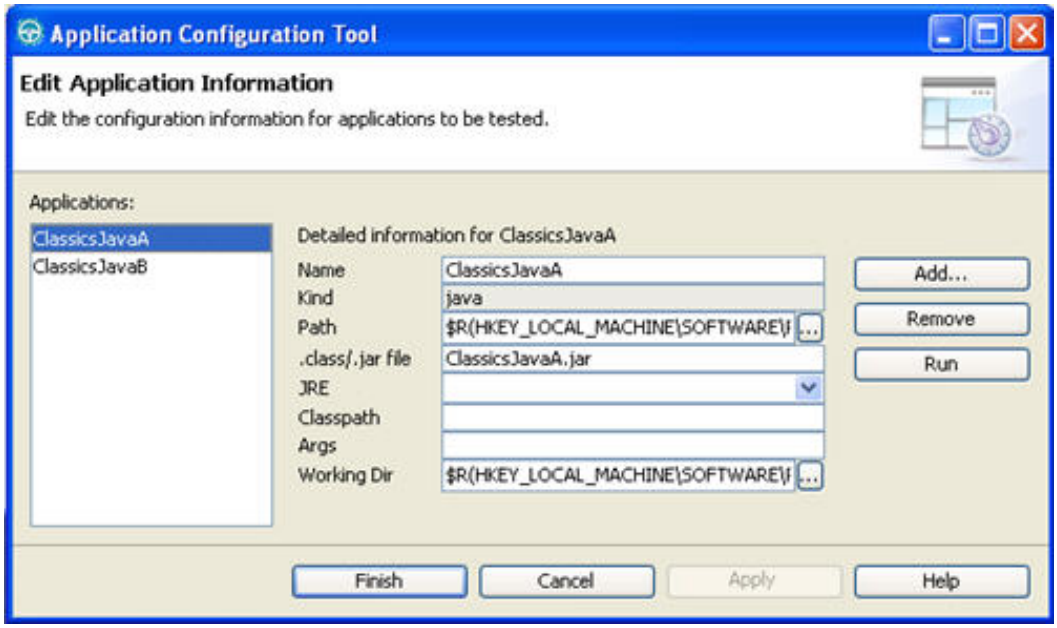
3. Select the Java Environments tab. The IBM SDP Java Runtime Environment (JRE) is the default JRE used to test Java applications and is enabled for testing by Functional Tester. If your application uses a different JRE, enable your JRE through this interface. For this tutorial, use this default JRE for playback, so you don't need to do anything here.

4. Click **Finish** to close the window.

Configure the application under test

Configuring the application under test does not actually affect or change your application in any way; it really just creates a shortcut, or *pointer,* in the Functional Tester environment that makes it easier to launch the application and makes the tests more portable to other test machines.

1. Select **Configure > Configure Applications for Testing...** In the Application Configuration Tool window, shown in Figure 6. A list of all the applications that have been configured for testing with Functional Tester is displayed. Functional Tester comes with a sample Java Swing application called *ClassicsJava.* Two builds of ClassicsJava are provided, and both are defined in Functional Tester automatically when it is installed. You can see here how run parameters such as the JAR file, classpath, JRE, and working directory can be specified.
Figure 6. Application Configuration Tool window



2. Because you will be working with the ClassicsJava sample application in this tutorial, there is nothing more you need to do on this screen at the moment. Click **Finish** to close the window.

Section 4. Recording a test

Tests can be created by hand coding, recording, or a combination of the two. Even if you intend to do some custom coding, it is often easier to record a test first and then modify it than it is to write a test from scratch.

In this tutorial, you will record the act of purchasing a CD within the sample ClassicsJava application.

Start the recorder

Animated demo
Would you like to see these steps demonstrated for you?

Show me

1. Start the recorder by clicking the red record button on the toolbar.
2. In the Record a Functional Test script wizard, enter `PlaceOrder` for the script name and click **Finish**. The test recorder launches, Functional Tester minimizes, and the recording toolbar, illustrated in Figure 7,

appears. The recording toolbar has several features that are useful during recording. Only a few are used in this tutorial.

Figure 7. Recording toolbar

Launch the application to be tested

1. Click the Start Application button to launch the `ClassicsJavaA - java` application shortcut. ClassicsJava build A launches.

2. Keep in mind that anything you do within the ClassicsJava application is now being recorded. Use the Composers tree control to select **Schubert > Symphonies Nos. 5 & 9**.

3. Click **Place Order**.

4. This sample application does not really check logins or passwords. For simplicity, accept the defaults when these are requested; however, check the **Remember Password** check box. This check box will illustrate the

robustness of Functional Tester's test playback engine later in the tutorial.

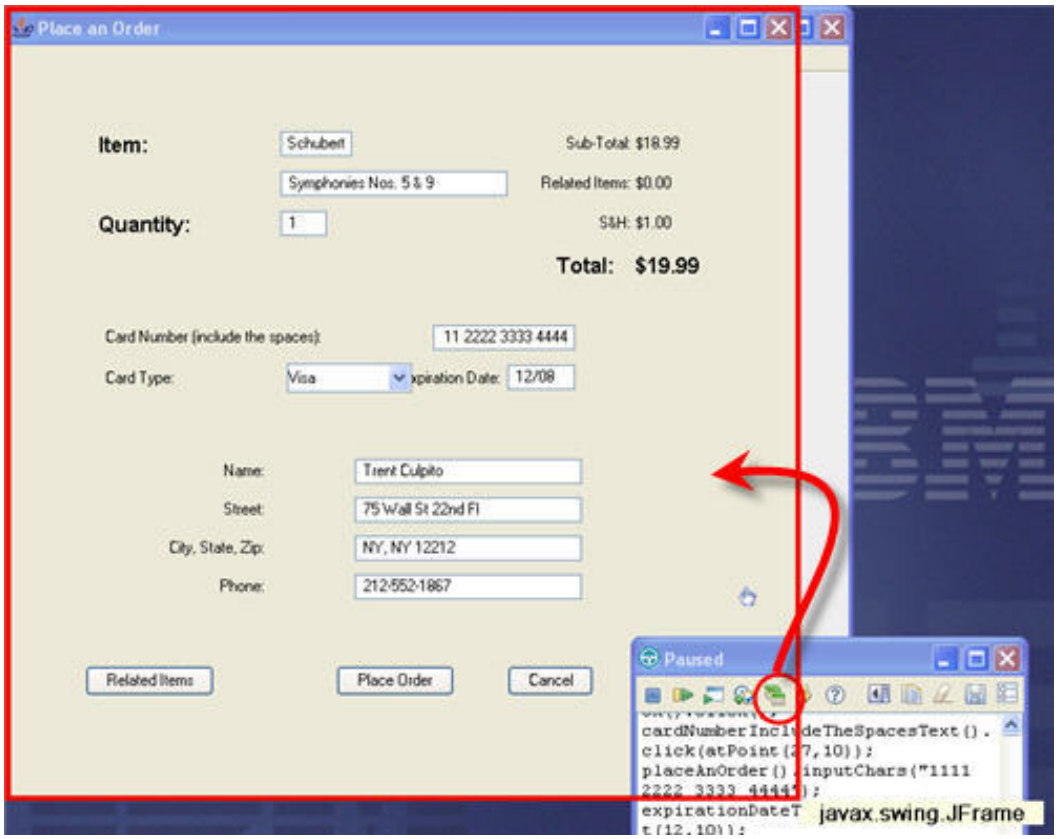
5. Click **OK** to be taken to the Place an Order window.

Create a data-driven order

By default, all keyboard entry is captured in the test script. Functional Tester has a very powerful capability: it can separate the data entered by the user from the procedure and navigation commands of the test. The advantage in doing so is that the same test procedure can be used over and over with different sets of varied data, enabling you to reuse common tests and greatly reducing the time and effort involved in creating repetitive tests.

1. On the Place an Order window, Enter `1111 2222 3333 4444` as the Card Number, leave the Card Type as Visa, and enter `12/08` as the Expiration Date.

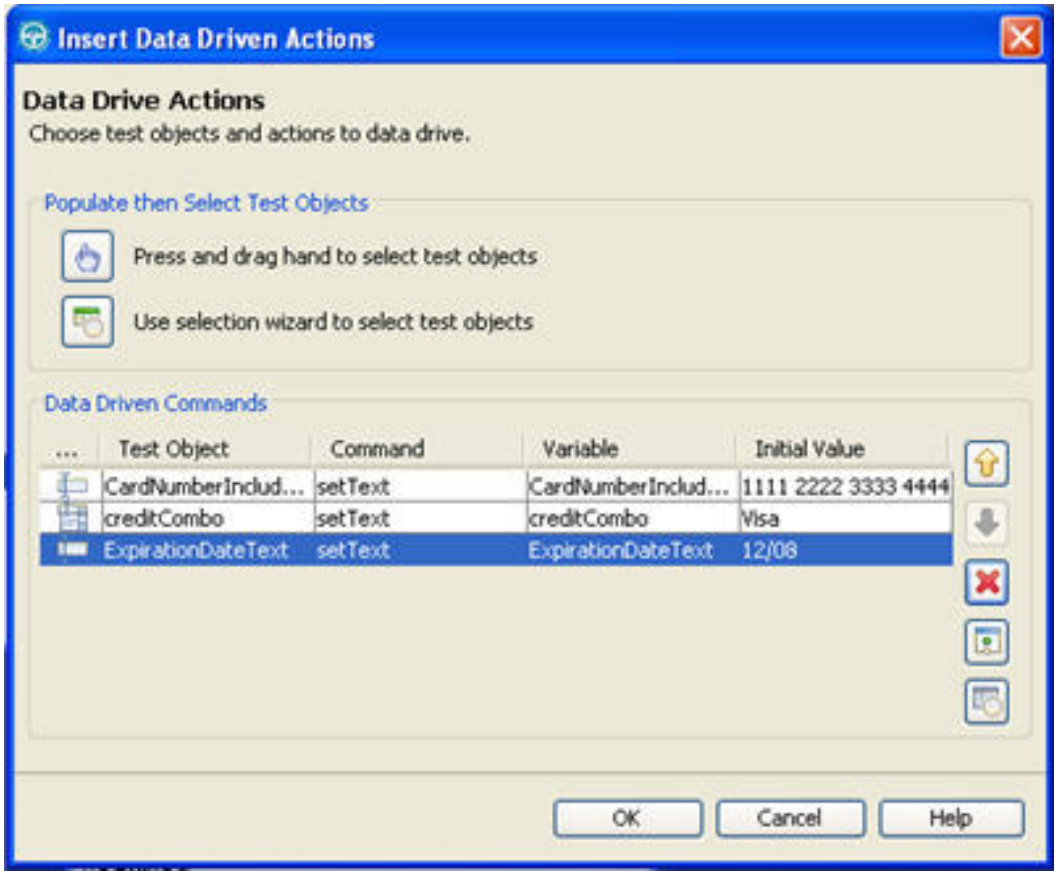
2. From the recording toolbar, drag the Insert Data Driven Commands icon over the order form, so that entire form is encased in a red square, as shown in Figure 8.
Figure 8. Select fields for data-driven access



3. The Insert Data Driven Actions window shown in Figure 9 appears. By default, Functional Tester assumes that you want to data drive all test objects or fields on the panel. In this case, however, you want to data

drive the credit card information only. That enables you to use this same script to run tests with multiple credit cards. To delete the unnecessary test objects, highlight each one and click on the red X icon to delete. Do this for all values except *CardNumberIncludeTheSpaces, creditCombo*, and *ExpirationDateText*.

Figure 9. Insert Data Driven Actions window



4. Click **OK** to confirm your choices.

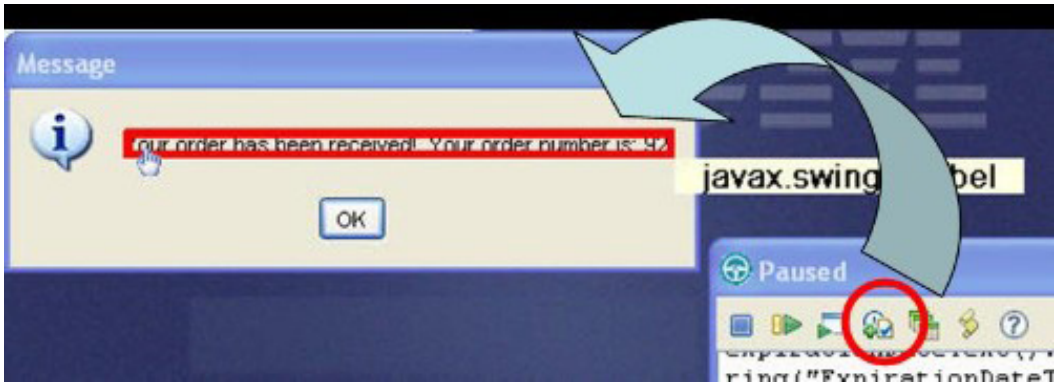
5. Click the Place Order button on the Place an Order window. A confirmation window appears. Verify the message in this window as shown in the next section.

Verify dynamic data

Regular expressions

Functional Tester supports the use of all the standard *regular expressions* with which you might already be familiar. In this tutorial, you'll see them used with data verification points; in addition, they can be used in property verification point values as well as object recognition properties. To learn more about regular expressions, see the Functional Tester on-line help. First, select **Help > Help contents;** then, within the help menu, choose **Recording functional test scripts > Working with verification points > Regular expressions.**

Applications often respond to input with data and information that you cannot fully predict. For example, the confirmation message you'll receive in ClassicsJava contains a two-digit confirmation number. You might be able to predict that this will be a two-digit number in future orders, but the value of that number will be different each time you run the application. You need a way to verify the *pattern* of the message while being flexible about the actual value of the number. Functional Tester verification points have this flexibility, as you'll learn in this section.

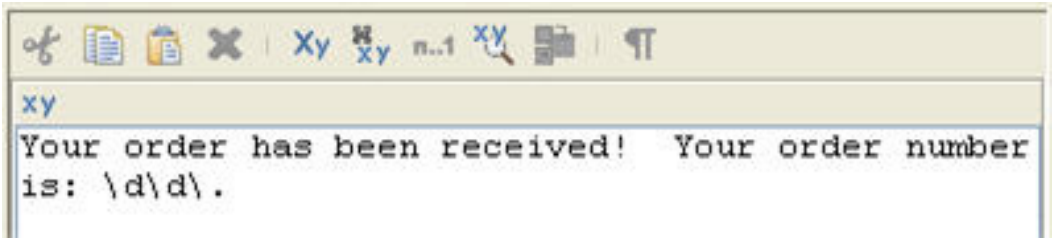
1. Drag the **Insert Verification Point or Action** button from the recording toolbar over the text in the **Message** window, shown in Figure 10. When the red box highlights *only* the text beginning with "Your order has been received...", release the mouse button.
   Figure 10. Select label for verification
   

2. There are various types of verification points. Here, you will add a simple data verification point. Click **Next** twice to accept the defaults.

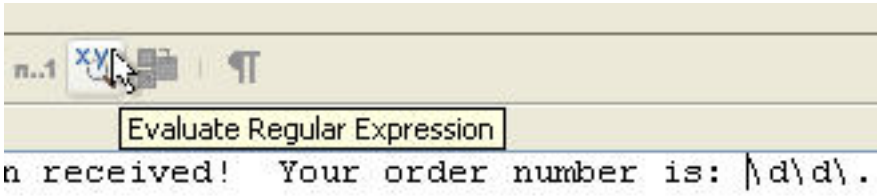
3. On the Verification Point Data screen of the wizard, click the Convert Value to Regular Expression button. Regular expressions are a form of pattern matching language with which you might already be familiar (see the sidebar to learn where you can find out more).

4. Edit the data in the window to erase the two-digit number.

5. Right-click in the spot where the number had previously been. The context menu provides a list of the most commonly used regular expression patterns. Use the context menu to insert `\d\d` where the two-digit number had been. This pattern indicates that the response value should be a two-digit number, but does not specify the exact digits. The test will fail if the number is one digit or three digits, or if it contains something other than 0-9 in each digit location. Your data should look like Figure 11.
   Figure 11. Verification point data with regular expression



6. Functional Tester also provides a way for you to verify that your regular expression does what you need it to do. Click **Evaluate Regular Expression** on the toolbar, illustrated in Figure 12.
Figure 12. Evaluate Regular Expression button



7. In the Regular Expression Evaluator, you can try various patterns in the Match Against Value box. Change the value and then press **Evaluate**. You will see whether or not the values you enter match the regular expression.

8. When you are comfortable that your pattern does what you need it to do, click **OK** on the Regular Expression Evaluator.

9. Click **Finish** on the Verification Point wizard. This puts you back into record mode.

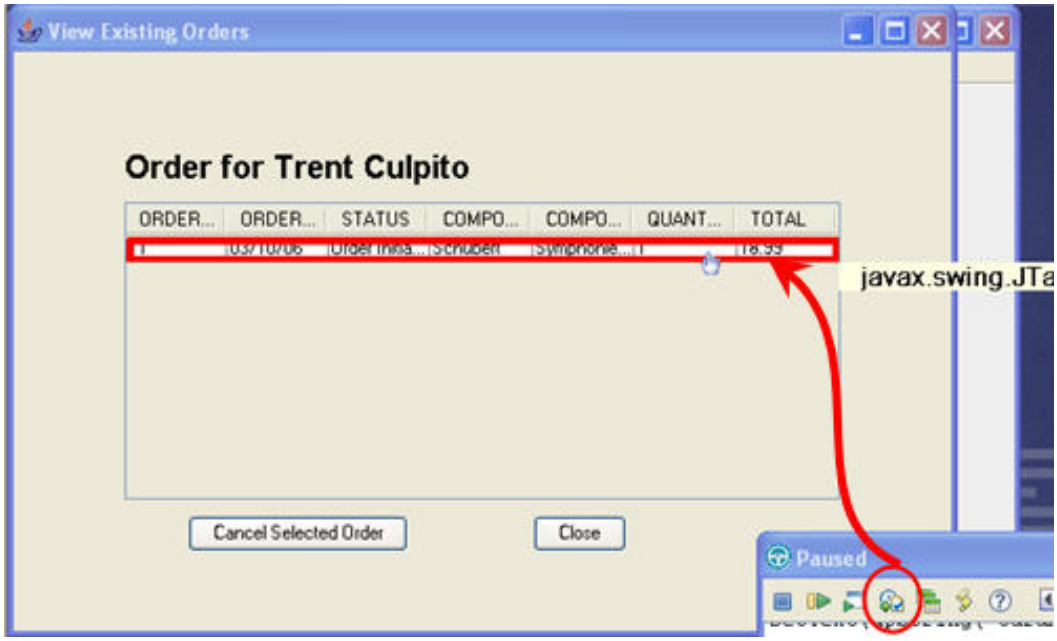
10. Click **OK** on the ClassicsJava confirmation message window.

Verify static data

Finally, you need to create a verification point to verify that the order was processed correctly.

1. From the ClassicsJava menu, select **Order > View Existing Order Status.** Click **OK** to log in again.

2. With the order displayed, click and drag the Verification Point and Action wizard onto the order information, so that the order is encased in a red square, as illustrated in Figure 13.
Figure 13. Select order for verification



3. Click **Next** on the first screen of the wizard.
4. On the second screen, select the data value **Table Contents** and click **Next**.
5. On the final screen, click **Finish** to store the table contents as the baseline data to which future test runs will be validated. This returns you to recording mode.

Shut down the application and stop recording

1. Click **Close** on the View Existing Orders window.
2. Click the red X in the upper-right corner of the ClassicsJava window.
3. Click Stop Recording on the Recording Toolbar.
4. The Object Map and Help windows appear. Close these windows at this time.

Section 5. Examining the test

Examine the Test Object Map

Once recording has finished, Functional Tester restores itself. The newly recorded test script is in the center view. Scroll through the code and observe the commands recorded.

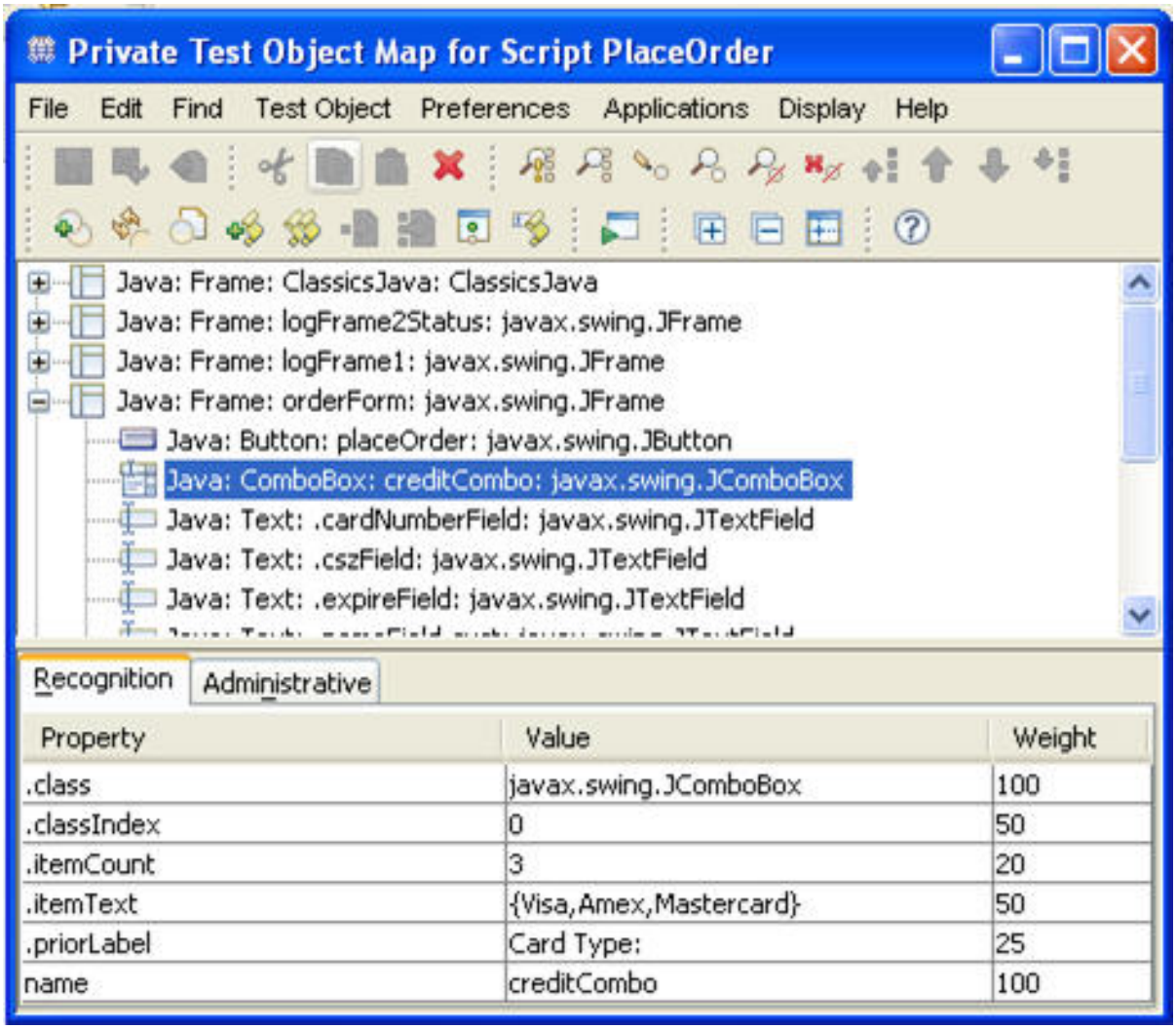
Animated demo
Would you like to see these steps demonstrated for you?

Show me

To the right is the Script Explorer view. This view shows you information about the script, such as the class hierarchy, datapools, verification points, and test objects associated with the test.

Double-click on the **creditCombo** object in the Test Objects tree. The Test Object Map window opens, as illustrated in Figure 14. The Test Object Map contains all the properties gathered during recording related to all the objects encountered. The properties in this map will be used during playback to identify the objects in the application. The weight on the right side of each indicates how important that property is for recognition.

Figure 14. Test Object Map for creditCombo



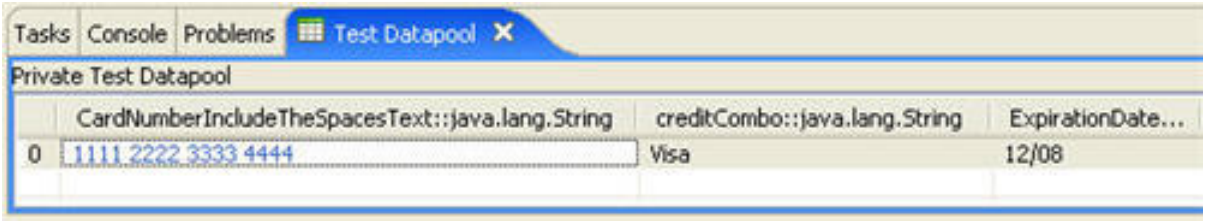
This particular control's entry in the map has six properties. Functional Tester's patented ScriptAssure™ fuzzy logic technology will use all of these properties to determine how good each match is during playback. A change in any one property will not cause the playback to fail. Even two or three properties can likely be changed in a new version of the application without causing playback to fail.

Close the Test Object Map.

Add data variations to the datapool

Double-click the **Private Test Datapool** object in the Script Explorer view. This brings up the datapool associated with this script in the lower portion of the window, as shown in Figure 15.

Figure 15. Test Datapool View



1. Right-click anywhere within the Test Datapool and select **Add Record...**
2. A second row of data will appear for your test. Double-click in the credit card number field and enter a new card number: `1234 1234 1234 1234`.
3. Change the credit card type to **Amex.**
4. In the expiration date field, enter the expiry date `06/08`.

Add custom behavior to the test

As mentioned earlier, the test script is full Java code -- not JavaScript and not a proprietary language. This gives you extremely powerful capabilities to address unique needs in your tests. Also, Functional Tester provides a rich Application Programming Interface (API) through which you can access test objects and control test execution.

Remember that as you recorded the test, you first manually entered the credit card number and expiration date, then used the data-driven command wizard to enable Functional Tester to populate these fields from your datapool during playback. This has left some unnecessary commands in your test. These don't really hurt anything, but you can remove them at this time to see how the manual editing of script code works.

1. In the test script in the PlaceOrder.java view, remove the four lines shown in Figure 16 (note that if you navigated between fields with the Tab key instead of clicking, your commands will look slightly different):
   Figure 16. Code to be removed

```
// Frame: Place an Order
cardNumberIncludeTheSpacesText().click(atPoint(9,11));
placeAnOrder().inputChars("1111 2222 3333 4444");
expirationDateText().click(atPoint(16,12));
placeAnOrder().inputChars("12/08");
```

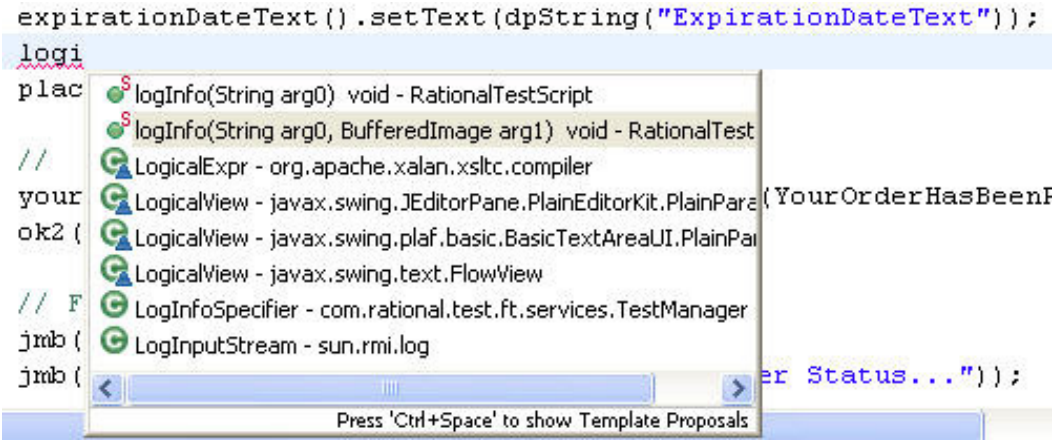
2. Next, you need to add a line of code that captures an image of the computer screen to the test log. In the test script, find the command that enters the expiration date from your datapool value. The line begins with `expirationDateText().setText`.

3. Position your cursor at the very end of this line and press the Enter key to begin a new line.

4. Begin typing `logi` and press Ctrl-Space. A pop-up window appears with all the potential code completions that are valid in this context, as shown in Figure 17. This feature is known as *Code Assist.*

5. Press the down arrow key on your keyboard to highlight the second item in the list. This is `logInfo()`, a static method on the `RationalTestScript` class that creates an informational entry in the test log, taking a `String` argument (the label) and a `BufferedImage` (the screenshot to insert).

Figure 17. Code Assist entry



6. Press Enter to insert the method call into the script.

7. The `String` argument placeholder is selected. Simply type `"Screen Snapshot"` (including the quotes) and press the Tab key to advance to the image placeholder.

8. To capture the screen image, use a method on the root test object. Type `getRootTestObject()`. Use Code Assist to minimize typing, if you want.

9. Add a semicolon to the end of the line. Save your changes. Your script should look like Figure 18 below.

Figure 18. logInfo() call inserted in script

```
expirationDateText().setText(dpString("ExpirationDateText"));
logInfo("Screen Snapshot", getRootTestObject().getScreenSnapshot());
placeOrder2().click();
```

There are many, many more things you can do with custom code in a Functional Tester script. Time does not permit us to explore custom code further, but capabilities like the extensive API, a powerful debugger, and Code Assist make custom scripting approachable, even for the novice tester.

Prepare for ClassicsJava Build B

Functional Tester comes with a second build of the ClassicsJava sample application. In the next section of this tutorial, you will play back your test against this new build to illustrate how Functional Tester can accommodate changes in the UI layout as applications evolve over time. But first, you need to prepare your script to use the new build.

1. Locate the `startApp` line of code near the top of the `testMain()` method in the script. Replace `ClassicsJavaA` with `ClassicsJavaB`. This will cause Functional Tester to use a different shortcut to launch the new version of ClassicsJava. Note that shortcut names are case sensitive, so be sure to use an uppercase *B.*
2. Save your changes.

Section 6. Running an automated regression test

A brief review might be in order before you move on to executing your regression test. Here are some of the things you have done so far in this tutorial:

- You set up your test environment by creating a test repository known as a *project.*
- You enabled your application environment to allow Functional Tester to see GUI components as objects.
- You configured your application to create a shortcut to simplify playback and make your tests more portable to other systems.
- You then recorded a test by manually walking through your test scenario with the application.
- While recording, you associated the data you entered with a datapool through a simple wizard with no hand-coding.
- You also inserted two verification points. One will verify static baseline data in a table. The other uses a regular expression to verify dynamic responses with pattern matching.
- You then enhanced your test by adding additional datasets to the datapool and adding custom code to make API calls to log a screenshot during playback.

Now that you have configured your test to launch the new build of your application under test, you are ready to execute a functional regression test and analyze the

results.

Execute the test

Animated demo
Would you like to see these steps demonstrated for you?

 Show me

1. From the toolbar, click **Run Functional Test Script**. The playback wizard gives you the opportunity to specify a log name and options before the test runs. Accept the default log name by clicking **Next**.
2. On the Playback Options screen of the wizard, you can specify the number of datasets through which the test will iterate. In the Datapool Iteration Count box, select **Iterate Until Done**. This specifies that you want Functional Tester to execute once for every row in the datapool.
3. Click **Finish.** Sit back and watch Functional Tester play back your tests. Note that if you have run this test previously, you might be prompted to confirm that you want to overwrite the log.

There are a couple things to note during playback:

- Notice how different the UI looks in this build of the ClassicsJava application. Several objects have been moved or resized, yet Functional Tester is able to locate them and interact with them properly.
- Functional Tester pauses for several seconds on the first login box before checking the password check box. You will look into this further when you examine the test results in the log file.
- The test runs through the entire scenario twice, once for each dataset in the datapool.
- Upon completion, the results are displayed in an HTML log in a browser. (This browser window might be hidden behind the Functional Tester window when it restores.)

In the next section, you will examine the results in the log file.

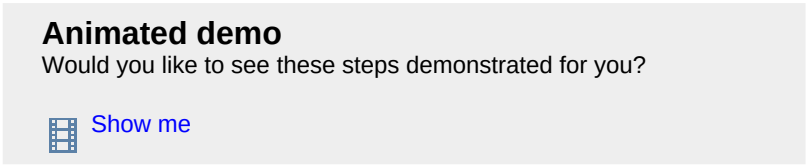
Section 7. Examine the results in the test log

By default, the test log is opened immediately upon test execution completion. If you

have closed the browser, you can find the test log in the Functional Test Projects view under the myFunctionalTestProject_logs node.

The left frame of the log provides quick navigation to interesting information. The right frame provides all the details.

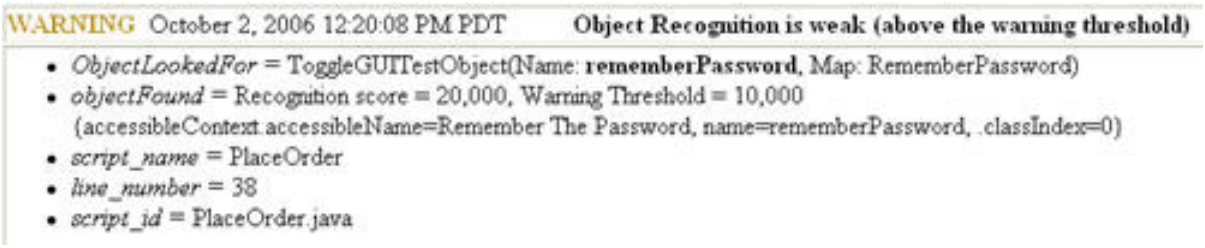
ScriptAssure



Scroll to the first warning, which should look like Figure 19. The message indicates "Object Recognition is weak (above the warning threshold)". Here is a case where a component on the UI has changed. In fact, multiple properties of this object have been changed by the developers between the two builds. Most test automation tools would have had no choice but to error out and abandon the rest of the regression suite upon encountering such an object. This is a real problem if you have a regression test suite that takes several hours to run and you have set it to do its business overnight, then come in to the office the next morning to find that it has failed and you have no results to show for it.

Functional Tester's ScriptAssure recognized that this was not exactly the same component it saw during recording. The pause during playback was Functional Tester waiting to see if the matching object might appear. After a specified timeout, Functional Tester decided that the object on the screen was close enough to the object it was expecting, logged a warning, and continued using that object. You can configure several property weights, timeout values, and thresholds to customize ScriptAssure's behavior to meet your needs.

Figure 19. ScriptAssure warning in test log



Custom log information

Scroll through the log to the next event, named "Screen Snapshot." This event is the result of the custom code you entered. Click the link that reads **Click to view full size**. This screenshot, shown in Figure 20, was captured during playback as a result of the `logInfo()` command you added to the test. Notice that the credit card number used was "1111 2222 3333 4444" -- the value from the first row in the datapool.

Figure 20. First screen snapshot

Click the Back button in your browser to return to the log.

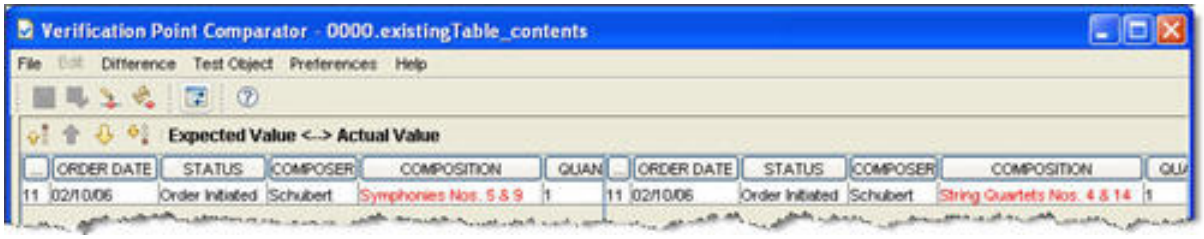
Investigate verification point status

Find the first verification point (VP) in the log. This entry in the test log for this verification point will be named "Verification Point [YourOrderHasBeenReceivedYou] passed," and should look like Figure 21. This is the VP where you used a regular expression to accommodate the dynamic order number.

Figure 21. Passing VP log event

Find the next verification point in the log. This event will be labeled "Verification Point [existingTable_contents] failed". Investigate this failure further by clicking the **View Results** link. The Verification Point Comparator shown in Figure 22 highlights in red the fact that the VP failed because, although the CD "Symphonies Nos. 5 & 9" was ordered, the application placed an order for "String Quartets Nos. 4 & 14." You have uncovered a regression in build B of the ClassicsJava application. Some seemingly unrelated change made to the code has caused the wrong item to be used in the order process. This is the real value in functional regression testing -- to find unintended changes in behavior. Test automation through Functional Tester makes the goal of regression testing every build achievable.

Figure 22. Verification Point Comparator



Close the Verification Point Comparator.

Verify datapool operation

Continue to scroll through the log. You will notice that the same set of events repeats a second time. This iteration occurs when the second row of data in the datapool is used. You can confirm the operation of the datapool by opening the second "Screen Snapshot" event. The credit card number will be "1234 1234 1234 1234," the value from the second datapool row, as shown in Figure 23.

Figure 23. Second Screen Snapshot

Section 8. Summary

This tutorial has provided an introduction to Rational Functional Tester. You enabled your application environment, recorded a functional regression test, enhanced the test through some simple custom coding, executed that test against a new build of the application, and analyzed the test results to uncover a regression defect in the sample application.

Stay tuned for the next part of this series, which introduces you to Rational Performance Tester. To keep up with all the Hello World articles, check the Hello World overview page.

Resources

Learn

- IBM Service-Oriented Architecture (SOA): Learn more about SOA from IBM.
- Functional Tester: Great resources from developerWorks.

Get products and technologies

- Eclipse: Learn more about this open source development platform.
- Rational Functional Tester V6.1: Download a free trial version
- Extension for Terminal-Based Applications for Rational Functional Tester: Download a free trial version. This extension enables you to use the powerful Functional Tester capabilities described here with terminal-based applications.

Discuss

- Participate in the discussion forum for this content.

About the author

Dennis Schultz
Dennis Schultz joined Rational in 1995 as a technical sales engineer. For eight years, he worked closely with numerous clients, implementing Rational solutions in their projects. Dennis helped deploy solutions for software configuration management, change management, requirements management, and test management and implementation. Since 2003, Dennis has been a Technical Marketing Engineer for IBM Rational software. Dennis holds a B.S. in computer engineering from Iowa State University. He is based in St. Louis, Missouri, and fills his non-work time with his four children.