[Skip to content](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html" \l "container)

* [*Safari*](https://www.safaribooksonline.com/home/)
* [Recommended](https://www.safaribooksonline.com/r/)
* [Queue](https://www.safaribooksonline.com/s/)
* [Search](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html)
* [Expand Nav](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html)
  + [Recent](https://www.safaribooksonline.com/recent/)
  + [Topics](https://www.safaribooksonline.com/t/)
  + [Highlights](https://www.safaribooksonline.com/u/p253751/)
  + [Settings](https://www.safaribooksonline.com/u/)
  + [Feedback](mailto:feedback@safaribooksonline.com)
  + [Sign Out](https://www.safaribooksonline.com/accounts/logout/)
  + [Settings](https://www.safaribooksonline.com/u/)**10** days left in your trial. [Subscribe](https://www.safaribooksonline.com/subscribe/).
  + [Feedback](mailto:feedback@safaribooksonline.com)
  + [Sign Out](https://www.safaribooksonline.com/accounts/logout/)

[Software Test Engineering with IBM Rational Functional Tester: The Definitive Resource](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html)

* [Search in book...](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html)

Top of Form

Bottom of Form

* [Toggle Font Controls](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html)
* [Share this](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html)
  + [Twitter](https://twitter.com/share?url=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html&text=Software%20Test%20Engineering%20with%20IBM%20Rational%20Functional%20Tester%3A%20The%20Definitive%20Resource&via=safari)
  + [Facebook](https://www.facebook.com/sharer/sharer.php?u=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html)
  + [Google Plus](https://plus.google.com/share?url=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html)
  + [Email](mailto:?subject=Safari:%20Chapter%204.%20XML%20and%20Rational%20Functional%20Tester&body=http://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html%0D%0Afrom%20Software%20Test%20Engineering%20with%20IBM%20Rational%20Functional%20Tester%3A%20The%20Definitive%20Resource%0D%0A)

[Prev](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch08.html)

[Chapter 8. Handling Unsupported Domain Objects](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch08.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html)

[Chapter 10. Advanced Scripting with Rational Functional Tester TestObjects](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html)

**Chapter 9. Advanced Rational Functional Tester Object Map Topics**

**Jeffrey R. Bocarsly, Daniel Chirillo**

*The Rational Functional Tester Object Map is a mechanism to give objects in the target application visibility to each Rational Functional Tester script. In* [*Chapter 1*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch01.html#ch01)*, “*[*Overview of Rational Functional Tester*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch01.html#ch01)*,” and* [*Chapter 2*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch02.html#ch02)*, “*[*Storyboard Testing*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch02.html#ch02)*,” you saw the basics of the Rational Functional Tester Object Map; in this chapter, you look under the hood of the Object Map. The map isn’t a single entity, but is constructed from methods that are autogenerated by Rational Functional Tester and inherited by each script. These methods return TestObjects, which serve as proxies between Rational Functional Tester and the target application objects with which each Rational Functional Tester script interacts. The recognition data for each object is persisted as an XML. All these bits must be present to have a runnable Rational Functional Tester script.*

**Rational Functional Tester Object Map**

Your typical interaction with the Object Map is through the Script Explorer View, which appears by default on the far right of the Rational Functional Test Perspective in Eclipse and on the far left in Visual Studio .NET. The Script Explorer displays a number of script assets, including the Test Objects folder, which provides access to the Object Map. If you open the Object Map (by clicking the **Test Object Map** icon), you can see the full contents of the Map. The fact that the Script Explorer might contain only a subset of the objects in the Map hints that the Rational Functional Tester Object Map apparatus is two-tiered, and indeed, it is.

**Object Map Components**

In the following discussion, you survey all the standard components that make up the Rational Functional Tester Object Map. These include the Script Definition, the Object Map, and the ScriptHelper class.

**The Script Definition**

Each Rational Functional Tester script has a Script Definition that persists all the assets associated with a script, such as the following:

• Object Map associated with a script

• Objects from that Object Map to which the script has access

• Verification Points (VP) that the script calls

• The datapool that is associated with the script

As this list indicates, the Script Definition is more than just a part of the Object Map, because it defines the script in terms of all its assets. However, it is part of the Object Map in the sense that it delineates the part of the Map that is seen by the script. Like nearly all noncode script assets, the Script Definition is stored as an XML document. This document is persisted in the /resources directory of your project in a file named by the convention ScriptName.rftdef. A typical Script Definition with a single object from the Map appears in [Listing 9.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09ex01).

**Listing 9.1** A Script Definition, XML



In [Listing 9.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09ex01), your main interest is the <ScriptNameMap> tag. This <ScriptNameMap> tag contains a <TestObject> child tag in this listing (it might contain any number of such tags). These <TestObject> tags define the specific objects in the target application to which this script has access. The <TestObject> tag has the child tags Name, ID, Role, and Deleted. The value of the <Name> tag is the display name of the object that appears in the Test Object tree in the Script Explorer. The <ID> is a GUID that Rational Functional Tester generates for each object in its Map. The <Role> tag indicates the type of object (Browser, Frame, Button, Text, Label, ComboBox, and so on).

**The Object Map**

The <TestObject> entry in the Script Definition in [Listing 9.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09ex01) tells Rational Functional Tester that this specific object is to be available to this script, so that the script can interact with the object at design-time and runtime. However, the actual information to recognize the specific object does not appear in the Script Definition; it appears in the Object Map, which, like the Script Definition, is stored in an XML format. The specific Object Map that the script uses is defined in the Script Definition XML (as you might expect), by the <Map> tag, which gives the path to the Object Map XML relative to the root directory of the project. This XML contains the actual object mappings and represents the central component of Rational Functional Tester’s Object Map. It contains the object hierarchy and recognition properties and their values. Map XML files are named using a similar convention to Script Definition XML files. If the Map is a shared Map, its XML file is stored at the project root and is named using the convention mapName.rftmap. If the Map is private, its file is stored in the \resources directory and is named ScriptName.rftxmap. The Object Map specified in the <Map> tag in [Listing 9.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09ex01), appears in [Listing 9.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09ex02).

**Listing 9.2** Object Map sample XML





The Object Map sample XML in [Listing 9.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09ex02) shows the basic features of the Map. First, everything in the map is a Map Attribute, which means <Attribute> tags contain the Map’s key information, and each <Attribute> tag might contain a different type of information. The type of information that an <Attribute> tag contains is indicated by a handful of tags common to all <Attribute> tags: a <Name> tag indicates the type of data in the <Attribute> tag and a <Value> tag specifies the <Attribute> values. The discussion here focuses on two types of Map attributes: one that defines the top level objects in the target application (<Name>.TopObjs</Name>) and one that defines the objects in the target application (<Name>.MtoSet</Name>).

The Map attribute that holds the top-level window objects has a simple structure: a <Value> tag with a child <Id> tag that has the GUID for the top-level window. In [Listings 9.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09ex01) and [9.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09ex02), the GUID is 0.5vYm2Dr5yw0:2FeZrp:LSlosoe:8WV, and Rational Functional Tester can easily determine whether an object in the Script Definition is a top-level window object by looking its GUID up in the .TopObjs attribute of the Map.

Similarly, Rational Functional Tester uses the object GUID to look up the .MtoSet (Mapped Test Object Set) attribute for an object, and this Map attribute contains the information to recognize the object at runtime. The actual recognition information appears in the <MTO> tag, a child of the <Value> tag. The <MTO> tag contains several child tags of which nearly all appear on the Administrative tab of the Rational Functional Tester Test Object Map viewer:

• Id—The Rational Functional Tester GUID for the object

• Name—The Rational Functional Tester name of the object, which is typically a window caption or htmlBrowser for a browser window, or an object name

• Parent—Rational Functional Tester GUID of a parent object if one exists

• TO—The TestObject subclass of the object

• Dom—Domain of the object (Html, Java, Net, and so on)

• Class—The class of the target application object

• Role—The role of the object, for example, window, frame, browser, and so on

• Proxy—Proxy class for target object

In addition, there are certain properties, called Recognition properties, that are specific to each object class. These object-specific properties appear in <Prop> tags, and Rational Functional Tester uses them to recognize each object specifically (they appear on the Recognition tab of the Test Object Map viewer). For illustration, a typical set of recognition properties for a Swing JLabel object is shown in the following XML snippet:



You can open any object in the Test Object Map viewer and compare what you find on the Recognition and Administrative tabs to the actual entries in the script’s Map XML.

**The ScriptHelper Class**

The final part of the Object Map mechanism lies in the “ScriptHelper” class, which every script inherits from. Each ScriptHelper class is named by appending Helper to the name of your script; this name is autogenerated and cannot be changed.

Note

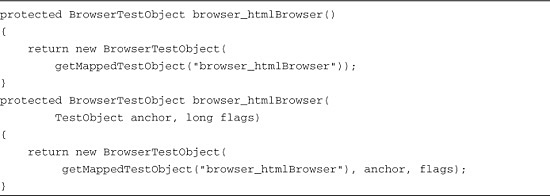
ScriptHelper is capitalized in this chapter to indicate that it *represents* the name of a class; however, it does not appear in the code font because it is not a true class name.

The contents of each ScriptHelper class are also autogenerated based on how you record your script. Each ScriptHelper contains methods that instantiate the TestObjects that represent target application objects in your script. These methods use the object names that appear in the Script Explorer view, and in the Script Definition XML. Typically, for each object in the Script Explorer, there are two methods in the ScriptHelper class: a default method that takes no arguments and an overloaded nondefault method that takes a Rational Functional Tester anchor argument. An anchor argument is an argument of type TestObject that a method call is “anchored” to. For example, in the following line of code, the document TestObject anchor argument “anchors” the inputKeys() method to the browser window with this specific document loaded:

browser\_htmlBrowser(document\_ibmUnitedStates(),DEFAULT\_FLAGS)  
.inputChars("rational")

[Listing 9.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09ex03) shows autogenerated code for a browser TestObject.

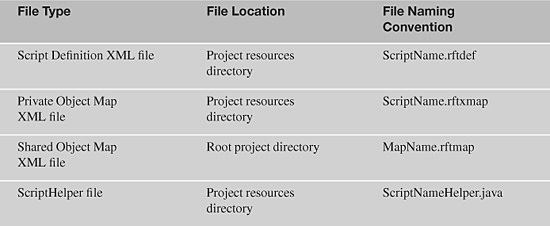
**Listing 9.3** Rational Functional Tester autogenerated ScriptHelper code for a browser



**Summary of Object Map Files**

[Table 9.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09tab01) summarizes the files that hold the Object Map components. Note that a script can have only one Script Definition, one ScriptHelper, and either a Private or a Shared Object Map file.

**Table 9.1** Rational Functional Tester Object Map Files: File Locations and Naming Conventions



You can see these underlying assets for a project simply by opening up the Eclipse Navigator View (**Window Menu** > **Show View** > **Navigator**). The Navigator view opens by default in the same pane as the Functional Test Projects View, and presents a tree view of projects. You can use this tree view to navigate all the underlying assets in your projects, including the Object Map assets.

Note

The Simple Scripting feature in Rational Functional Tester 8.1 provides a new layer that provides access to Rational Functional Tester functionality for nontechnical users and novice Rational Functional Tester users via a set of graphical tools. Under the hood, Simple Scripting uses the entire set of standard Object Map features as described in this chapter. For a script created under the Simple Scripting paradigm, all the same apparatus described here for a standard script comes into play.

**Rational Functional Tester Object Recognition Framework**

IBM Rational designed the Object Map feature set with two features that set it apart in the market space as an exceptionally well-designed and well-engineered offering. While considering the problem of script maintenance, the architects of Rational Functional Tester incorporated two powerful concepts into the Object Map:

• ScriptAssure, a weighting mechanism for recognition properties—The ScriptAssure weighting mechanism enables the user to change the recognition weights of properties according to their volatility across builds.

• Object Map Regular Expressions, an industry-standard pattern matching (Regular Expressions) functionality—The Regular Expression implementation enables the user to write patterns for Rational Functional Tester to match instead of fixed recognition values.

No other competing tool has the power of either of these features individually. Together, these features make the Rational Functional Tester Object Map technology the most advanced on the market.

**Object Recognition with ScriptAssure**

The ScriptAssure weighting values can be found in the third column of the Recognition tab of the Object Map viewer. The values that you are probably most used to seeing are the default ScriptAssure weights that are applied to an object when it is inserted into the Map. Typically, the default weighting works fine, but for situations where you know the weighting needs to be optimized, you can adjust it. When you click on the ScriptAssure weight field you’d like to modify, the value becomes editable, and you can enter any value between 0 and 100 (note that weights for some properties are not editable). Simply put, the significance of the values is that the role a property plays in object recognition scales with the weighting value. A lower value means that a property is a less important factor in recognition (a zero value means that a property plays no role in recognition), and a higher value means that a property plays a role of increasing significance in recognition.

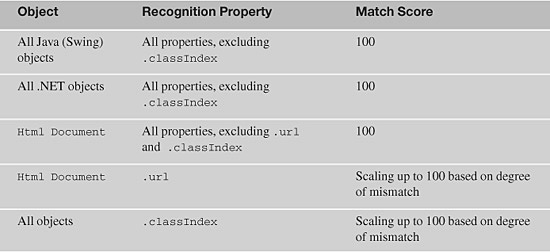
However, the actual implementation of this weighting concept is a bit more involved than that simple statement implies. The ScriptAssure algorithm works by scoring each candidate object by summing the contributions of each recognition property, scaled by the weighting scheme. As Rational Functional Tester tries to match recognition properties, if a recognition property value does *not* match to a candidate object found at runtime, the mismatch counts *against* recognition, and if it *does* match, the match contributes nothing to the total. If the total of all contributions from all properties for the candidate object exceeds a set amount, the object is not recognized by Rational Functional Tester because too many properties have failed to match. In this sense, the ScriptAssure scoring value is a measure of nonmatching because the higher the score, the poorer the match.

**ScriptAssure Recognition Property Scoring—Scoring Recognition Properties**

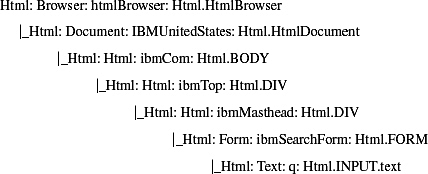
Rational Functional Tester matches objects at runtime by scoring the recognition properties of each candidate object based on matching recognition property values. The procedure works as follows: If a recognition property value for a candidate object does not match what the Map expects, the “intrinsic” match score of the property is multiplied by its ScriptAssure weight and the total of all contributions for the object’s recognition properties are summed. If property values do match, no contribution to the total is made. If the sum of the scoring contributions exceeds a specific amount, the object is not recognized (see the following for a description of scoring thresholds).

The scoring contributions of different recognition properties vary based on the property and, to a lesser extent, the domain. [Table 9.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09tab02) shows the intrinsic match scores of a variety of common recognition properties.

**Table 9.2** ScriptAssure Match Scores for Common Recognition Properties

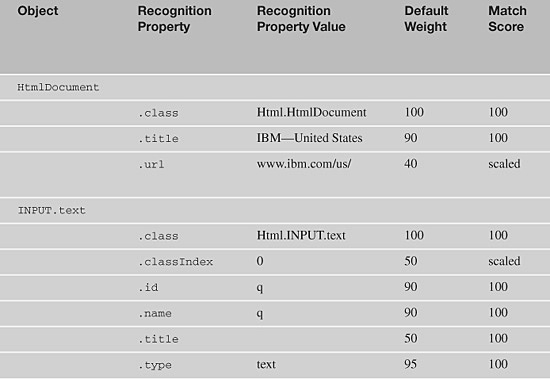


To see how this works, the following discussion takes some concrete examples and opens them up. Using the IBM home page, [www.ibm.com/us/](http://www.ibm.com/us/), you can capture the following slice of the object hierarchy (as of this writing):



These objects are all on the masthead of the page. This discussion focuses on the HtmlDocument object near the top of the tree and the search query Text input box at its bottom. The recognition properties of these objects are shown in [Table 9.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09tab03).

**Table 9.3** Recognition Properties of HTML Objects with Default ScriptAssure Weights and Scores



Let’s look at the consequences for the scoring if there are mismatches between the expected values for recognition properties and the actual values captured during the recognition process at runtime. For the INPUT.Text object, if Rational Functional Tester fails to match the .id property, the scoring is:

90 × 100 = 9,000

If the match failure is on the .title property, the score is:

50 × 100 = 5,000

If Rational Functional Tester fails to match on both .id and .title for the INPUT.Text object, the score is the sum of the individual scores:

(90 × 100) + (50 × 100) = 14,000

If the INPUT.Text object’s .classIndex value is off by 1, the match scoring is scaled as follows:

50 × 50 = 2,500

Look at some similar scoring numbers for the HtmlDocument object. If the .title property is mismatched, the scoring penalty is:

90 × 100 = 9,000

If the HtmlDocument's .url property is off due to some characters appended to the URL (for example, a new virtual directory level has been added to the application), the match score is scaled to 80. Using the default weighting of 40, the contribution to the total score is:

40 × 80 = 3,200

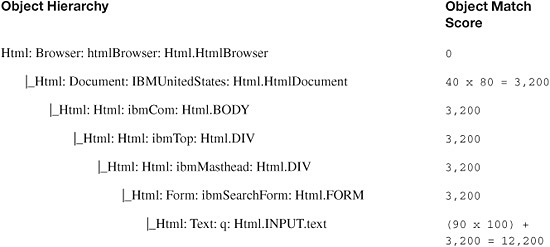
If both of these mismatches occurred, then the total score is:

(90 × 100) + (40 × 80) = 12,200

**ScriptAssure Recognition Property Scoring: Scoring the Object Tree**

Rational Functional Tester’s concept of object matching doesn’t rely on matching recognition property values of just the target object. Rather, to assure that the specific object in the target application has been located, Rational Functional Tester attempts to match the target object *and* all objects along the branch leading to the target object, up to the root of the object hierarchy (which usually is a top-level window). Moreover, the ScriptAssure scoring is *inherited*—meaning that if an object higher in the tree is scored with a mismatching property value, every object beneath the mismatched object inherits its score. This inheritance feature means that the recognition score of an object reflects its own score and the score of the inheritance tree above it.

Consider an example using the previous tree. Suppose the HtmlDocument object had a defective .url property value in the Map with a scaled match score of 90, as shown in the previous examples. Assume that, the .id property of the search query INPUT.text object is also a mismatch with a weight of 90 and a match score of 100 (see [Table 9.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09tab03)). The math works out on each tree node as follows:



All the object nodes under the HtmlDocument object inherit its score of 3,200 and because these objects have no mismatches, they contribute 0 additional to the score. The INPUT.text object contributes its own score of 9,000 due to its own mismatch, which gives it a total matching score of 12,200.

**ScriptAssure Scoring Thresholds**

The ScriptAssure implementation uses its scoring to achieve a more sophisticated behavior than a simple, binary match-no match conclusion does. This added sophistication is achieved through a multi-tiered set of threshold values, in which different threshold match scores are defined and are combined with recognition timeouts. The thresholds are as follows:

• Maximum acceptable recognition score—Sets the threshold score *under* which an object is recognized by Rational Functional Tester as a match candidate. The default value is 10,000.

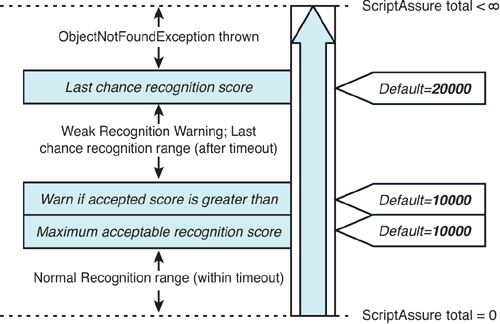
• Warn if accepted score is greater—Defines the threshold *above* which a “weak recognition” warning is written to the log. The default value is 10,000.

• Last chance recognition score—If a match is not made in the timeout period, then this threshold defines the maximum score an object might have for Rational Functional Tester to recognize it as a candidate match. The default value is 20,000.

• Ambiguous recognition score difference—If the scores of the top candidate matches in the pool of candidates differ by less than this amount, an AmbiguousRecognitionException is thrown. The default value is 1,000.

The scheme works as follows (using the default threshold values). Rational Functional Tester first looks for a pool of candidate match objects with ScriptAssure scores lower than the “Maximum acceptable recognition score.” If it finds match candidates and can select one using the “Ambiguous recognition score difference” threshold, a match is made. If no candidates fit these criteria within the matching timeout, but candidates are found that are between the “Warn if accepted score is greater” and “Last chance recognition score” values, then if Rational Functional Tester can match to one using the “Ambiguous recognition score difference” threshold, a match is made. In this case, Rational Functional Tester writes a weak recognition warning to the log (and fires an onRecognitionWarning() event), and execution proceeds as if the object has been properly recognized. If no candidate scores below the “Last chance recognition score,” then an ObjectNotFoundException is thrown and appropriate messages are written to the log. These relationships are schematized in [Figure 9.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09fig01).

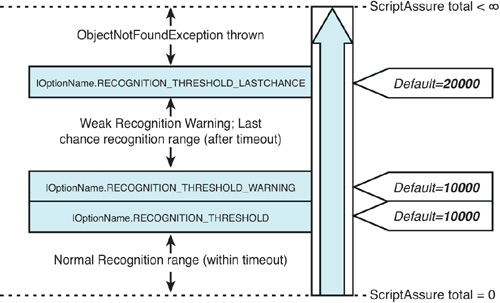
**Figure 9.1** Schematic of ScriptAssure threshold cutoff values



In the previous example, the score for the HtmlDocument, BODY, DIV, and FORM objects are all 3,200 each and the score for the INPUT.text object is 12,200. Only the score of the INPUT.text object exceeds any of the default thresholds—it exceeds the “Maximum acceptable recognition score” (10,000) and the “Warn if accepted score is greater” thresholds (10,000), but not the “Last chance recognition score” (20,000). This means that the INPUT.text object is recognized by Rational Functional Tester during script execution, but an Object Recognition is weak warning message is written to the log for this object. None of the other objects exceeds the Maximum acceptable recognition score threshold, so they are considered to be normally recognized by Rational Functional Tester.

Usually, the default threshold values, like the default ScriptAssure weightings, work just fine for most applications. In a rare event, you might have to modify the thresholds, which can be done *globally* using the Rational Functional Tester Preferences window (select **Functional Test** > **Playback** > **ScriptAssure** in the tree view). [Figure 9.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09fig02) shows the default constants by name within the threshold scheme.

**Figure 9.2** Schematic of ScriptAssure threshold constants



You also have the option of setting the threshold values in your code using the names shown in [Figure 9.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09fig02), using the method setOption() with the IOptionName constants. You can set the values of any of the thresholds in your scripts at runtime using any of the following:

image

In these examples, the variable thresh is the threshold value that you are setting for the execution. Note that setting ScriptAssure threshold values using setOption() only sets the value for the scope of the run; the default values set in your preferences are unaffected and apply normally to any other script where setOption() is not called.

**Pattern Matching: Regular Expressions in the Object Map**

The Rational Functional Tester Object Map has an additional power feature that is unequalled by any competing tool in the functional/regression automation space. This is the ability to use pattern matching, or Regular Expressions, in the Map in place of fixed values for recognition properties. If you convert a recognition property to a Regular Expression and provide an appropriate pattern, Rational Functional Tester searches for the object based on the pattern and accepts any value that matches the pattern. This feature can be extraordinarily useful in a number of situations. For example, when unique identifiers appear in recognition property values or when a series of related objects are named by a convention (and you want to handle them with a single Map entry), a Regular Expression provides a perfect solution.

Note

In the Java flavor of Rational Functional Tester, the power of Regular Expressions was offered before they were a standard part of the Java libraries. Therefore, Rational Functional Tester uses a customized version of the Apache Regular Expression implementation. You are still free to use the Java Regular Expressions engine (released in Java 1.4) in any Java code you write in your scripts.

To illustrate how powerful Rational Functional Tester’s Regular Expression implementation makes Object Map recognition, the following discussion explores some common Regular Expressions syntax to solve typical recognition issues that you might encounter. The first topic considered is how to use Regular Expression operators to define patterns. After that, Regular Expression syntax is used to solve the problem of matching a set of related URLs, a common issue that arises with web applications.

**Common Regular Expression Operators**

Regular Expression patterns are built out of common pieces of text along with *operators* (also called *metacharacters*) that specify a text pattern to match. Operators are typically indicated by punctuation characters. The following examples illustrate some of the Regular Expression operators that are most useful in the Object Map; a more complete discussion of Regular Expressions as implemented in Rational Functional Tester can be found in [Appendix B](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app02.html#app02), “[Regular Expressions in Rational Functional Tester](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app02.html#app02).” The operators considered in this discussion are:

• . (period)—Matches any individual character (except newline).

• \* (asterisk)—Matches 0 or more of the previous pattern.

• + (plus)—Matches 1 or more of the previous pattern.

• [] (square brackets)—Defines a character class. A character class describes a position in a string and a set of characters that are a match for that position.

To understand how the Regular Expressions matching engine behaves, consider the simplest kind of pattern, which is a pattern defined by a sequence of characters without any operators. For example, if you have the text “four score and seven years ago,” and you try to match it with a Regular Expression pattern of “seven,” the Regular Expressions engine happily matches the word “seven.” This illustrates that the engine fundamentally looks for a character match for its pattern.

The next step is to examine how operators can be used to generalize a text pattern so that only the characteristics of the matching text are described and not the specific text itself. As you look through these examples, it is important to pay attention to how the operators work, and more important, to how they work *together*. Much of the power of the Regular Expression operators is in how they can modify each other. The first and simplest example is to look at how the asterisk operator works. The asterisk modifies the pattern that precedes it and matches the preceding pattern any number of times, including 0 times. For example, in the pattern 'a\*', the asterisk modifies the pattern 'a'. The 'a\*' pattern matches but it also matches 'aa' and ‘aaa' and 'aaaa' and 'aaaaa', or any number of 'a's (because \* matches any number of the pattern it modifies). It matches 'b' as well, because \* matches as few as 0 of the pattern it modifies. Now, consider the combination of the asterisk with the period, '.\*'. In this '.\*' pattern, the period matches *any single character* (except newline), and the asterisk therefore matches *any number of any character* (including no characters). The '.\*' pattern is thus the most promiscuous pattern, matching nearly anything, *including* nothing.

Let’s look at a closely related pattern to '.\*', the pattern '.+'. The only difference between the '\*' operator and the '+' operator is that the ’+' operator requires a minimum of one of the previous pattern for there to be a match. This pattern would match any set of characters of any size, but it would not match 0 characters as '.\*' would. The '.+' pattern therefore requires at least one character to match.

The final Regular Expression operator considered here is the square brackets, '[]', which define character classes. As noted previously, a character class specifies a set of characters that are matches for a specific position in a string. Consider a simple example—you want to write a pattern to match a word that has two spellings: gray and grey. Start by composing your pattern with the letters that both spellings agree with, the g, r, and the final y. The position after the r, which is the position at which the two spellings diverge is defined with a character class. The class is composed of the two possible letters that might occupy the position, a and e, producing the pattern gr[ae]y. This pattern translates as: match any character sequence where the first position is a ‘g’, the second position is an ‘r’, the third position is *either* an ‘a’ or an ‘e’, and the final position is a ‘y’.

Character classes have some additional syntax features. Although you can explicitly specify all characters of a class, ranges of characters can be specified by a simple shorthand notation. For example, '[0-9]' means a class including all the numeric characters between 0 and 9 inclusive. Similarly, the class '[3-6]' is equivalent to the class '[3456]'. Likewise, '[a-z]' includes the lowercase alphabet and '[A-Z]' is the uppercase alphabet. Shorthand notations can be combined in a single class: '[a-zA-Z]' includes both lowercase and uppercase alphabets.

Just as you saw with the '.', character classes can be modified by the '\*' and '+' operators. So, the pattern '[0-9]+' matches any number of digit characters, as long as there is at least one digit. The related pattern '[0-9]\*' would match any number of digit characters (including no digits, of course). The pattern '[0-9A-F]+' would match any number denoted in hexadecimal notation.

Now that you have examined some basic Regular Expression operators, look at some common situations where they might be used in the Object Map. To apply the Regular Expression syntax, consider the situation in which you have some related URLs from a web application that you want to represent with a single object in your Map:

image

These URLs all start with the same character string <http://www.etailer.com/products/>. They end with an instance identifier, a sequence of letters and number in three fields separated by underscores. The first field is all numbers, the second all uppercase letters, and the third is a mixture of the two. Think about some possible patterns that could be used to match the URLs. The simplest pattern, as suggested earlier, is simply to use the characters common to all three URLs:

http://www.etailer.com/products/

This matches all the URLs, and this type of match probably works for many of your Object Map patterns. However, imagine that you might in the future need to distinguish between URLs that end with /index and those that end differently. You can modify this pattern to include this by employing the most general metacharacter pattern, .\*:

http://www.etailer.com/products/.\*/index

This matches your URLs, and the approach is quick to implement. However, note that .\* can be *too* powerful—it can match more than you intend. A less general, more specific pattern can avoid unintended consequences due to an overly broad pattern. In this case, it is easy to write one. You can write a character class that contains all the permitted characters (all numbers, all uppercase letters, and the underscore character), and modify it with the + operator to require at least one member of the class:

http://www.etailer.com/products/[0-9A-Z\_]+/index

Again, you can ask whether this pattern is overly broad. You can certainly use your knowledge of the URLs and Regular Expressions to write an even more specific pattern because you can specify the three fields in the instance identifier separately. The pattern is:

http://www.etailer.com/products/[0-9]+\_[A-Z]+\_[0-9A-Z]+/index

In this pattern, the first field is constrained to be at least one number, the second field to be at least one uppercase letter, and the third field to be at least one character from the set of numbers and uppercase letters. Note that in this final pattern, if you have a letter in your first field or a number in your second field, the Regular Expressions engine does not find a match. This pattern opens the door for even more granular specificity in the match. If you wanted, for example, to match only URLs where the second field ends in X, you can write:

http://www.etailer.com/products/[0-9]+\_[A-Z]+[X]\_[0-9A-Z]+/index

There is one additional subtlety in the previous three patterns that is worth mentioning. As will all URLs, the server name uses a period ([www.server.com](http://www.server.com/)). However, the period is also an operator! Because the period operator matches any single character, it matches itself, and your pattern matches as expected. However, the pattern matches *any* character appearing where the periods appear in the server name (www!server!com would match, for example). If you want to, you can escape the period operator and tell the Regular Expression engine to treat the period as a normal character and not a metacharacter. Escaping operators is done by enclosing the operator in square braces (or preceding them with backslashes). The final pattern would therefore look like:

http://www[.]etailer[.]com/products/[0-9]+\_[A-Z]+\_[0-9A-Z]+/index

Note that in most cases, the first and simplest pattern is sufficient for Object Map purposes. There are cases, however, where a simple but general pattern might be too broad for the need, and in this case, the additional syntax becomes useful. The downside of writing a highly specific pattern is that you may exclude cases that you don’t want to—always be sure that your pattern does the work you need it to. For convenience, several of the regular expressions in the previous discussion are collected in [Table 9.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09tab04).

**Table 9.4** Summary of Regular Expression Examples



**Customizing Recognition Properties and Weights**

Under most circumstances, Rational Functional Tester’s default recognition properties and weights meet your needs (particularly when regular expressions are used). If you find yourself in a situation where you need to modify them—add or remove recognition properties or change their weights—Rational Functional Tester makes this possible.

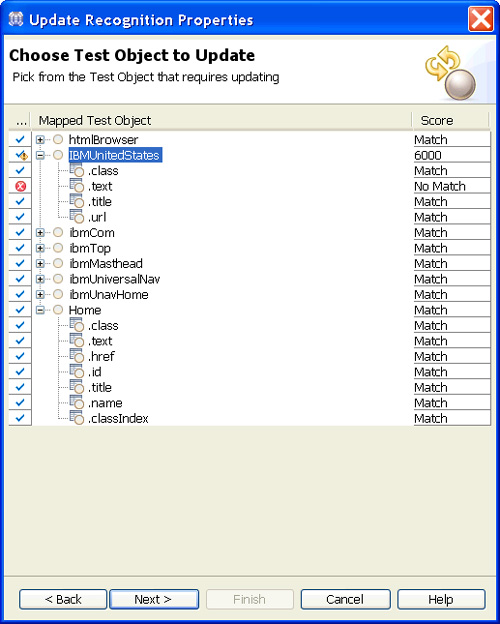
**Remove a Recognition Property for an Individual Test Object**

To remove a recognition property from the Object Map for a Test Object, open the Map and either set the property’s weight to zero (the recommended choice) or delete the property entirely by right-clicking on the property and selecting **Delete** in the context menu.

**Add a Recognition Property for an Individual Test Object**

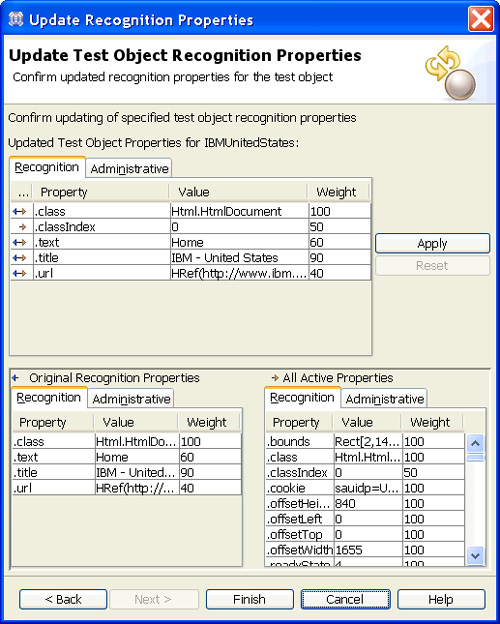
To add a recognition property to the Object Map, you must first make the Test Object available to the Object Map. After you do this, click the Test Object in the Object Map and select **Test Object** > **Update Recognition Properties** from the Object Map menu. Rational Functional Tester highlights the object on your desktop, captures its current (live) property values, and displays the dialog box shown in [Figure 9.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09fig03); recognition property discrepancies are automatically pointed out.

**Figure 9.3** Updating recognition properties—step 1



Now click the **Next** button; the dialog box shown in [Figure 9.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09fig04) appears.

**Figure 9.4** Updating recognition properties—step 2



This dialog has three lists of properties and values: all the recognition properties *currently* stored in the Object Map (Original Recognition Properties on the bottom left), all the Test Object’s current mappable (value class) property values (All Active Properties on the bottom right), and the properties and property values that are added, removed, or modified to the Map.

To add a recognition property to the Map, follow these steps:

**1.** Double-click the desired property in the **All Active Properties** list (or right-click on it and select **Add to Unified Test Object Properties**).

**2.** The property immediately appears in the Updated Properties list at the top of the dialog box.

**3.** If you accidentally add the wrong property, click the **Reset** button or right-click it and delete it. You can change the property values and weights.

**4.** After you add all the properties you want to add to the Map for the selected test object, click **Finish**.

**Changing Default Recognition Properties and Property Weights with the Object Properties Recognition Tool**

For each test domain-object class combination (for example, Java/java.awt.Frame and Html/Html.Table), Rational Functional Tester tries to capture a set of recognition properties. The properties it captures and the weights it assigns to each property for each type of Test Object are exposed to you via the Object Properties Recognition Tool. To start the tool, select **Configure** > **Configured Recognition Properties** from the Rational Functional Tester menu. The Object Properties Recognition Tool can be used not only to inspect but also to modify the default properties and weights Rational Functional Tester captures and stores in the Object Map.

If you decide that you want Rational Functional Tester to use the *.*text property by default as a recognition property for all HTML documents, follow these steps:

**1.** Select **Html**.

**2.** Under Select the Object Class, click **Html.HtmlDocument**.

**3.** Click the **Add** button.

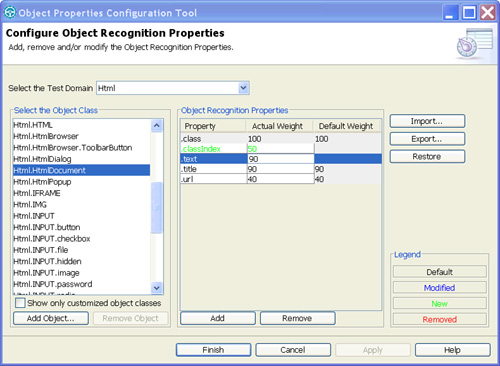
**4.** Double-click in the **Property** cell and enter **.text**.

**5.** Double-click in the **Actual Weight** column cell and add the weight you want Rational Functional Tester to assign to the .text property.

**6.** Click **Finish**.

If you add the .text property to the Html.HtmlDocument class and assign a weight of 90, you see what’s shown in [Figure 9.5](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09fig05).

**Figure 9.5** Adding a recognition property



To use the Object Properties Recognition Tool to change the weight of a recognition property used by a Test Object class, select the object class and change the value in the Actual Weight column.

Note that any customizations you make using the Object Properties Recognition Tool do not take effect in any Test Objects already in the Object Map unless you run the Update Recognition Properties utility described earlier in this section for each Test Object.

To remove your customizations at any point, that is, to go back to using Rational Functional Tester’s default properties and weights, select the Test Object class and click the **Restore** button.

**Renaming Test Objects (ScriptHelper Methods)**

Unclear or ambiguous method names make code difficult to understand and, therefore, to maintain. In some cases, the ScriptHelper method names generated by Rational Functional Tester’s recording engine do not make it obvious to the reader which objects they reference. In such cases, it is wise to rename them.

The simplest technique you can use to change the name of a Test Object (this amounts to changing the name of a ScriptHelper method) is to rename the Test Object from the Script Explorer view. Right-click on the Test Object, select **Rename**, and enter a new name. Eclipse automatically renames all calls in the current script to that ScriptHelper method. This is both simple and convenient, but the name change is made to the current script only. If the script in which you rename a test object is associated with a shared map and that test object has been added to multiple scripts, the new name is not reflected in any other scripts. Luckily, Rational Functional Tester has a mechanism you can use to rename a Test Object in multiple scripts at once.

To rename a Test Object in multiple scripts (that share the same test Object Map) in a single bound, do the following:

**1.** Highlight the test object’s node in the Object Map Viewer.

**2.** Change the test object’s Descriptive Name on the Administrative tab to the name you want to use.

**3.** Select **Test Object** > **Renew Name in Associated Script(s)**.

**4.** Modify the name if desired.

**5.** Click **Finish**. Rational Functional Tester renames the ScriptHelper methods in all scripts (associated with the given Object Map) that contain the Test Object.

Depending on the application domain of the Test Object you rename (for example, HTML, Java, .NET), the method name you entered in Step 2 might be slightly different from the name Rational Functional Tester presents to you see in Step 4. For example, if you changed the Descriptive Name of an HTML table to customersTable, the default name that Rational Functional Tester generates and presents to you in Step 4 is table\_customersTable. The reason for this is that the ScriptHelper method name is created using an algorithm set in a project template file. The root of the Functional Tester project contains a templates directory that contains files used to set default names and attributes of various script assets. The pattern or algorithm for setting the ScriptHelper method name is found in the ft\_script\_<domain>\_object\_name.java.rfttpl file. For example, for the HTML domain, the file is ft\_script\_html\_object\_name.java.rfttpl. If you open the ft\_script\_html\_object\_name.java.rfttpl file, you can see that it contains a single line:

%map:#role%^\_%map:#name%^

You should interpret this as the value of the #role administrative property, followed by an underscore, followed by the value of the #name administrative property. If you look at the Administrative tab, you can see (#name) next to Descriptive Name and (#role) next to Role.

This is why, in the previous example (renaming an HTML table), the default name generated is table\_customersTable. If you don’t want the ScriptHelper method names of html objects prefixed with the role, you can modify the template file to contain the following:

%map:#name%^

**Searching the Object Map**

The Object Map editor offers the following four ways to search the map for Test Objects. Think of them as queries, as shown in [Figure 9.6](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09fig06).

• Find Test Objects that scripts use.

• Find Test Objects that no scripts use.

• Find Test Objects using a simple query.

• Find Test Objects using a custom query.

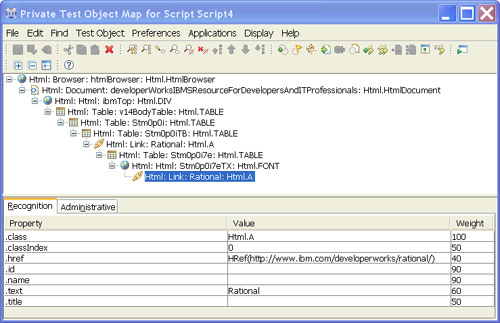
**Figure 9.6** Object Map find menu



**Searvching for Test Objects Used and Not Used by Scripts**

In the context of the Object Map, a *used Test Object* is a Test Object that either appears in the Test Objects folder of a script’s Script Explorer or is an ancestor within the Test Object hierarchy in a script’s Script Explorer. For example, if you perform a single action in a new script, such as a click on the link to IBM Rational at <http://www.ibm.com/developerworks>, the Script Explorer contains just one Test Object: the link. However, if you open the Object Map, you see what’s shown in [Figure 9.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09fig07).

**Figure 9.7** Object Map—a single object added to a script



As discussed in previous sections of this chapter, to find the link, Rational Functional Tester must find all the link’s ancestors that display in the Object Map. From Rational Functional Tester’s perspective, therefore, all the Test Objects shown in [Figure 9.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09fig07) are *used by* the script.

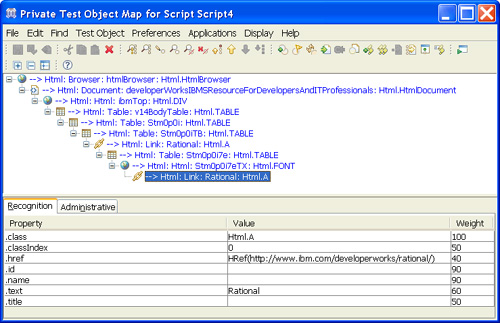
To search for Test Objects that scripts use, select **Find Used** in the Object Map’s Find menu. Rational Functional Tester finds Test Objects required by any scripts associated with the given Object Map and presents the results of the search to you by making three changes to the map editor (see [Figure 9.8](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09fig08)):

• An arrow appears to the left of each used Test Object.

• The font color of each found Test Object changes to blue.

• Navigation buttons are enabled on the toolbar.

**Figure 9.8** Object map search results



Use the navigate buttons on the toolbar (Find: First; Find: Previous; Find: Next; Find: Last) to navigate through the Test Objects that have been found. You can search for Test Objects that are not required by any of the Object Map’s associated scripts by selecting **Find Not Used** under the Find menu. The results of the search are presented in the same manner described previously.

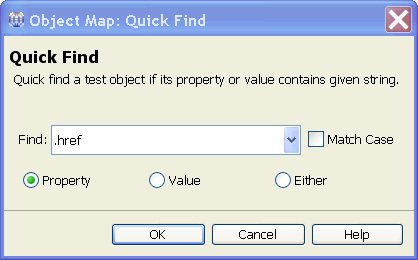
**Searching for Test Objects Using Filters**

The Object Map search facility enables you to create two types of queries to search for Test Objects: a simple query (**Find** > **Quick Find**) and a complex query (**Find** > **Find by Filters**).

**Searching for Test Objects Using Quick Find**

The Quick Find search utility enables you to search for objects using a simple Regular Expression (one without any operators) that can be a Test Object’s recognition property name, recognition property value, or either of these (see [Figure 9.9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09fig09)).

**Figure 9.9** Quick Find



If you’re familiar with SQL queries, think of the Quick Find utility as a way to create any of the following queries:

• SELECT \* from TestObjects WHERE PropertyName like % <>%

• SELECT \* from TestObjects WHERE PropertyValue like %<>%

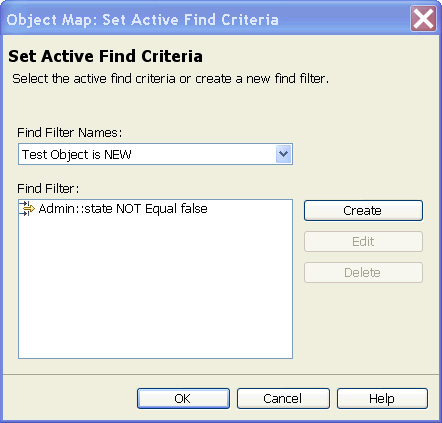
• SELECT \* from TestObjects WHERE PropertyName like %<>% OR PropertyValue like %<>%

For example, imagine that an HTML hyperlink has a .href property. If you want to find all the hyperlinks in the map, enter **.href** in the Find field, select **Property** to indicate that you want to find Test Objects with a .href Property, and click **OK**. If you want to find all Test Objects that contain *Rational* anywhere within any recognition (or administrative) property value, enter **Rational** in the Find field, select **Value**, and click **OK**.

**Searching for Test Objects Using Find by Filters**

To search the Map using multiple filters, select **Find** > **Find by Filters**. The dialog box shown in [Figure 9.10](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09fig10) displays.

**Figure 9.10** Find by Filters



The Find Filter Names contains queries already created. To create a query, follow these steps:

**1.** Click the **Create** button. In the dialog box that displays, select the properties you want to include as query filters. By default, the wizard groups the properties you select under an AND operator (thus creating a query filter of *roughly* WHERE x = <> **AND** y = <>, etc.). If you want to change the operator to OR, right-click **AND** and select **OR**. After you select all the properties you want to use in your query, click the **Next** button.

**2.** In the Define Find Filter Relationships dialog, set the filter values you want to use for each of the properties used in your query. Select a property in the filter tree and then set the expression you want to express about that property. The wizard allows you to select one of three operators for each filter: IsNull, Exists, and Equals. Equals is what is used most often. In addition to these operators, you can negate the expression by selecting **NOT Relationship**. You can also use a Regular Expression in the value by clicking the Regular Expression button.

**3.** After you create expressions for each property, click the **Next** button.

**4.** The last step is to give your query a name. Enter a name and click **Finish**.

**5.** Your query now appears in the list of Filter Names. To execute your search, select it and click **OK**.

**6.** If you want to change your filter, select it and click the **Edit** button. You can then add or remove filter properties and change the expressions used for each.

**Sharing Test Objects Between Multiple Scripts**

A question that is often asked is, “How do you add a Test Object to multiple scripts?” The next question that’s asked is, “How do you share Test Objects between multiple scripts?” Recall that making a mapped Test Object available to a script requires not only that the Test Object be added to the script’s associated test Object Map, but also that the Test Object be added to the script. This second step is the potentially expensive operation. Even if you had a single shared Object Map for all your scripts and you wanted a Test Object to be capable of being referenced in all scripts, you have to add the Test Object to the map only once, but you then need to manually add the Test Object to each script from the Object Map. Without any scripting, this is your only option. With coding, you have more options.

Scripts access mapped Test Objects through the ScriptHelper methods. If ScriptA has a Test Object added to it, ScriptB can access the Test Object if it can invoke ScriptA’s ScriptHelper method. The challenge here is that the ScriptHelper methods are, by default, protected, which means they can only be called by classes in the same package as the ScriptHelper class and inheriting classes. The challenge is now how to give other script classes access to protected methods. The following sections explore two possible solutions.

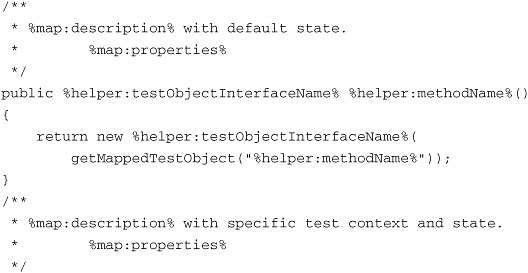
**Solution 1**

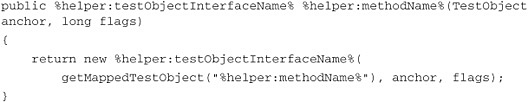
Make the protected ScriptHelper methods public. This is by far the simplest route. The ScriptHelper classes are maintained by Rational Functional Tester. In fact, they’re recreated every time you record in a script or insert a Test Object or verification point. If you change the access modifier from protected to public in the ScriptHelper class directly, your changes persist only until the ScriptHelper class is regenerated by Rational Functional Tester. Even if you could change the signatures directly, it could potentially be a huge task: there are two methods created for each Test Object that’s added to a script (a no-argument version and a version that takes an anchor argument and a state flag).

As it turns out, changing the signatures of potentially hundreds of ScriptHelper methods in your Rational Functional Tester project is amazingly simple. The helpers’ signatures are controlled by a project template file (located in the templates directory of the Rational Functional Tester project): ft\_scripthelper\_method.java.rfttpl (ft\_scripthelper\_method.vb.rfttpl in Visual Studio). Here are the contents of this file in Eclipse:



If you want the helpers to be public, all you need to do is replace protected with public and regenerate the ScriptHelper classes:



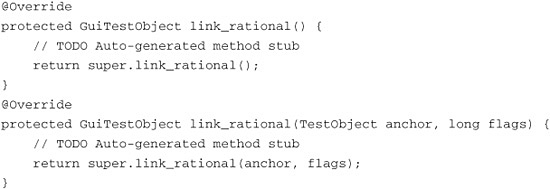


To regenerate a script’s ScriptHelper class, open the script and select **Script** > **Update Script Helper** from the Rational Functional Tester menu.

**Solution 2**

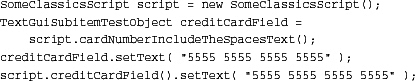
Create public wrapper methods in script classes around the protected ScriptHelper methods. This solution involves more work and maintenance because it requires that you create the public wrappers for every Test Object you want to share. Because the public methods are created by you, this does give you the opportunity to set the method names to whatever name you see fit.

If you choose this approach, you can have Eclipse do a lot (though not all) of the grunt work. Specifically, you can use one of Eclipse’s wizards to generate the wrappers for you. They just won’t be public. To use this wizard, right-click anywhere in a script and select **Source** > **Override/Implement Methods**. In the dialog box that displays, select the methods in the ScriptHelper class for which you want to create wrappers and click **OK**. Eclipse inserts methods that call the ScriptHelper class’ methods. For example:



Be sure you leave super.<ScriptHelperMethodName>. If you’re satisfied with the method names, the only modification you need to make to each method is to change the access modifier from protected to public. If you want to change method names, you can, but you need to remove the @Override annotation.

Whichever approach you choose (making the ScriptHelper methods public or creating public wrappers), any script that wants to access a publicized Test Object would do so by first creating an instance of the script class that exposes the ScriptHelper method and then invoking its public methods that return mapped test objects. For example:



If you want to reduce the number of test object variables in your scripts you have to declare, you can use a style in which you chain your calls:

SomeClassicsScript script = new SomeClassicsScript();  
script.creditCardField().setText( "5555 5555 5555 5555" );

All these approaches give you a range of options for making objects in your Maps more broadly available than they are in the standard Object Map use of Rational Functional Tester. Because you are working with the internals of a Rational Functional Tester project with the template-based approach, be careful! Back up your project or export it before you modify any templates.

**Summary**

In this chapter, you took a tour of the Rational Functional Tester Object Map internals to become familiar with the building blocks from which the Rational Functional Tester Object Map is assembled. You looked at how object recognition works in the Rational Functional Tester and investigated Rational Functional Tester’s powerful ScriptAssure and pattern-matching technologies. You also looked at practical power features, such as renaming objects, searching the Map, and sharing mapped objects across a project. With the underlying features of the Object Map in hand, you have the foundation to discuss TestObjects at an advanced level (see [Chapter 10](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10), “[Advanced Scripting with Rational Functional Tester TestObjects](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10)”).

[Prev](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch08.html)

[Chapter 8. Handling Unsupported Domain Objects](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch08.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html)

[Chapter 10. Advanced Scripting with Rational Functional Tester TestObjects](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html)

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

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

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

[**Make font larger Make font smaller**](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html)