Create a test automation framework using Rational Functional Tester

Shinoj Zacharias

August 06, 2013

Shinoj Zacharias describes how to create an automation framework using Rational Functional Tester. He takes you through the entire process, starting with an introduction to Rational Functional Tester through the find Application Programming Interface(API) to create automation framework. This article includes examples and sample scripts.

Introduction to IBM Rational Functional Tester

IBM® Rational® Functional Tester is an automated functional testing and regression testing tool. This software provides automated testing capabilities for functional, regression, GUI, and data-driven testing. Rational Functional Tester supports a range of applications, such as web-based, .Net, Java, Siebel, SAP, terminal emulator-based applications, PowerBuilder, Ajax, Adobe Flex, Dojo Toolkit, GEF, Adobe PDF documents, zSeries, iSeries, and pSeries.

Benefits of using Rational Functional Tester

The benefits of Rational Functional Tester include:

• **Automated testing:** Enables testers to automate tests resilient to frequent application user interface changes with ScriptAssure technology.
• **Storyboard testing:** Simplifies test visualization and editing using natural language and rendered screenshots.
• **Data-driven testing:** Lets you perform the same series of test actions with a varying set of test data.
• **Test scripting:** Combines recorded user actions with multiple customization options and intelligent script maintenance capabilities.
• **Integrations:** Integrates with IBM Rational Team Concert and IBM Rational Quality Manager to provide access to work items and logical or compound SCM test asset support.
• **Framework creations:** Rich set of APIs to create test automation frameworks.

Prerequisites

To perform the steps used in this article, the user must Rational Functional Tester version 8.2 or higher and possess a basic knowledge of IBM Rational Functional Tester.
Need of an automation framework

Most automation tools enable testers to simply record a set of GUI interactions and play them back against the application being tested. The record-playback approach eases the creation of test scripts but leads to extensive test script maintenance. The test team ends up recording and re-recording scripts each time the application changes. Although record-playback features can be used to create suites of test cases quickly, this approach is limited because the application under test should be mostly complete and functioning correctly before a workable script can be recorded. Test teams often abandon the record-playback model and write test cases manually instead.

Testers are often limited to using GUI automation for regression testing. The object-recognition algorithms are complex and inaccessible, making updates to the scripts tedious and, in some cases, impossible. Though the tool exposes the recognition algorithms, thereby making updates much more manageable, it also has the unfortunate side effect of making object recognition less reliable.

Rational Functional Tester and automation framework

The automation framework offers many benefits:

- Powerful scripting capability. The automation framework leverages Java and Visual Basic programming languages.
- Object-oriented approach to test automation.
- Object recognition algorithm and object maps.
- Ability to dynamically find the test objects using the object's properties.
- Rich set of APIs for scripting without recording the object.
- Building automation, even if the application is not available.
- Extensibility mechanism using proxy SDK for custom controls.

Automation framework using find API in Rational Functional Tester

Using find API in building an automation framework has these benefits:

- The API for finding a control in Application Under Test (AUT) at execution time is easy-to-use.
- There is no requirement for an object map.
- Ensured resilience of the script, even if the application changes.
- The ability to use the find API on any test objects available in Rational Functional Tester (for example, GuiTestObject, BrowserTestObject, and so on).
- The ability to use the find API on Rational Functional Tester's RootTestObject.

Two types of find API in Rational Functional Tester:

- Find(Subitem Properties).
- Find(Subitem Properties, Boolean mappableOnly).

Subitems for Find API
Subitems can be either `atChild()` or `atDescendant()` or `atList()`.

**atChild**
One or more properties that must be matched against the direct child of the starting TestObject.

**atDescendant**
One or more properties that can be matched against any child of the starting TestObject.

**atList**
A sequential list of properties to match against. Valid subitems for `atList` are `atChild`, `atDescendant`, and `atProperty`. The first list item is matched first, to get a list of candidates. The descendants of the first list are matched against the next list item, and so on.

**mappableOnly**
Arguments that limit the search. If it is set to true, the search for children will be limited to those test objects that are mappable, otherwise non-mappable test objects are also searched.

### Find() with Subitem atChild

If `find()` is used on a TestObject with `atChild` subitems, then the `find()` attempts to match one or more properties against the direct children (immediate children) of the TestObject.

There are three types of `find()` API with atChild subitem:

- `find(atChild(Property[] properties))`
- `find(atChild(String propName, Object propValue))`
- `find(atChild(String propName1, Object propValue1, String , propName2, Object propValue2 ))`

An example using different variations of `find()` with atChild is shown in Listing 1.

**Listing 1. Different variations of find() with atChild**

```java
public void testMain(Object[] args) {
    TestObject[] tobs1 = find(atChild(".class", "JButton"));
    TestObject[] tobs2 = find(atChild(".class", "JButton", "text", "OK"));
    Property prop1 = new Property(".class", "JButton");
    Property prop2 = new Property(".name", "ok");
    Property prop3 = new Property("text", "OK");
    Property[] props = {prop1, prop1, prop3};
    TestObject[] tobs3 = find(atChild(props));
}
```

### Find() with Subitem atDescendant

If `find()` is used on a TestObject with `atDescendant` subitems, then `find()` will try to match one or more properties against any child (not just direct/immediate children) of the TestObject.

There are three types of `find()` API with atDescendant subitem:
• find(atDescendant(Property[] properties))
• find(atDescendant(String propName, Object propValue))
• find(atDescendant(String propName1, Object propValue1, String propName2, Object propValue2 )

An example of using different variations of find() with atDescendant is shown in Listing 2.

**Listing 2. Variations of find () with atDescendant**

```java
public void testMain(Object[] args) {
    TestObject[] tobs1 = find(atDescendant(“.class”, “JButton”));
    TestObject[] tobs2 = find(atDescendant(“.class”, “JButton”, “text”, “OK”));

    Property prop1 = new Property(“.class”, “JButton”);
    Property prop2 = new Property(“.name”, “ok”);
    Property prop3 = new Property(“text”, “OK”);
    Property[] props = {prop1, prop2, prop3};
    TestObject[] tobs3 = find(atDescendant(props));
}
```

**Find() with Subitem atList**

Find with atList can be used to prove a sequential list of properties to match against. Valid subitems for **atList** are **atChild**, **atDescendant**, and **atProperty**. The first list item is matched to get a list of candidates. Their descendants are matched against the next list item. This process continues.

There are five types of find() with atList subitem:

• find(atList(subitems))
• find(atList(subitems[]))
• find(atList(subitems, subitems))
• find(atList(subitems, subitems, subitems))
• find(atList(subitems, subitems, subitems, subitems))

An example using find() with atList is given in Listing 3.

**Listing 3. find() with atList**

```java
public void testMain(Object[] args) {
    TestObject[] tobs = find(atList(atDescendant(“.class”, “JFrame”), atChild(“.class”, “JButton”, “.value”, “OK”)));
}
```

**Find() using RootTestObject**

**RootTestObject** represents a global view of the system being tested. It provides access to system-wide functionality, such as finding an arbitrary test object based on properties, location, as well as finding DomainTest Object.
There are special properties that apply to a RootTestObject find. These include:

**.processName**
Dynamically enables the process for testing. Constrains the find to processes with the specified name.

**.processId**
Dynamically enables the process for testing. Constrains the find to processes with the specified process id (pid).

**.domain**
Searches in top-level domains matching the .domain property.

**.hWnd**
Used in conjunction with .domain, if .domain is specified as Win, the matching window will be enabled for testing.

**Handle**
Used in conjunction with .domain, if the .domain is specified as Net, the matching window will be enabled for testing.

Listing 4 uses RootTestObject to enter text into the notepad application. The application notepad is configured using Rational Functional Tester’s Application Configuration Wizard.

### Listing 4. Sample script using RootTestObject

```java
public void testMain(Object[] args) {
    ProcessTestObject pto = startApp("notepad");
    Integer pid = new Integer((int)pto.getProcessId());
    RootTestObject rto = RootTestObject.getRootTestObject();
    TestObject[] textObjects = rto.find (atList(
        atProperty(".processId", pid),
        atDescendant(".class", "Edit")
    ));
    ((TextGuiTestObject)textObjects[0]).setText("Hello RFT");
}
```

### Creating automation framework using find API in Rational Functional Tester

Now that you understand the find API and its variations, the next step is to build an automation framework using this API. Automation engineers use the find API to build libraries that non-technical testers use to build test scripts. The libraries hide the complexities of object identification using Rational Functional Tester APIs. The novice user can use the high-level, easy-to-understand functions to create the tests quickly.

For example, a non-technical tester can write the test using simple functional calls, such as:

- ClickButton
• EnterText
• ClickLink
• ClickImage
• SelectComboBoxItem
• SelectRadio
• CheckCheckbox

The testers only provide the name of the control or the text to enter, for example. Finding the specific control using the find API is hidden from the testers. Automation engineers write the libraries containing the implementation of the above functions and provide them to the test engineers. All the complexities are hidden in the library.

One example of how one of these functions can be implemented using the find API is shown in Listing 5. The automation engineers write the libraries and functions, and testers use the libraries to create the test. For "click a button", the test engineer provides the name of the button (or value of the button). When entering text in a text field, use the name of the text control and value of the text. The example in Listing 5 is for web applications.

Listing 5. A sample function written using find API

```java
public static void ClickButton(String name) {
    GuiTestObject buttonObj = findButton(name);
    if (buttonObj != null) {
        buttonObj.click();
    } else throw new ObjectNotFoundException();
}

private static GuiTestObject findButton(String value) {
    TestObject[] tobs = find(atDescendant(".class", "Html.INPUT.submit", ".value", value), true);
    if (tobs.length == 0)
        return null;
    return (GuiTestObject)tobs[0];
}
```

The ClickButton() method uses an argument, in this case the value property of the button. In the findButton() method, use the find() API to get the control that matche the properties passed to the find() API. You can modify the method to search for the control based on properties such as .text or .name. This logic can also be included in the findButton() routine. You can also write different types of ClickButton methods, such as ClickButtonByValue(), ClickButtonByName(), or ClickButtonByLabel(), to search for the button by value, by name, or by label, respectively.

In Listing 5, the ClickButton method takes an argument, then the test engineer passes the .value property of the control. If the value of the .value property or any other properties used in the find() is not available, use the Test Object Inspector tool available in Rational Functional Tester, shown in Figure 1, to get all the properties and values that can be used in the find(). In the preceding example, return the first element that is returned by the find(). The result could be more than one candidate. If that happens, modify the routine to create more properties to find a unique element.
Similarly, the automation engineers can write the modules for other functions such as `clickText`, `EnterText`, and so on as in Listing 6.

**Listing 6. Examples for find() API based library functions**

```java
public static void ClickText(String name)
{
    TextGuiTestObject textObj = findTextObject(name);
    if (textObj != null) {
        textObj.click();
    } else {
        throw new ObjectNotFoundException();
    }
}

public static void EnterText(String controlName, String value)
{
    TextGuiTestObject textObj = findTextObject(controlName);
    if (textObj != null) {
        textObj.setText(value);
    } else {
        throw new ObjectNotFoundException();
    }
}

public static void ClickPassword(String name)
{
    TextGuiTestObject textObj = findPasswordObject(name);
    if (textObj != null) {
        textObj.click();
    } else {
        throw new ObjectNotFoundException();
    }
}
```
private static TextGuiTestObject findTextObject(String name) {
    TestObject[] tobs = find(atDescendant(".class", "Html.INPUT.text", ".name", name), true);
    if(tobs.length == 0)
        return null;
    return (TextGuiTestObject)tobs[0];
}

private static TextGuiTestObject findPasswordObject(String name) {
    TestObject[] tobs = find(atDescendant(".class", "Html.INPUT.password", ".name", name), true);
    if(tobs.length == 0)
        return null;
    return (TextGuiTestObject)tobs[0];
}

If some of the properties used for the find() method change for different instances or different builds of the application, you can convert those property values to regular expressions and minimize the script maintenance.

RegularExpression buttonRE = new RegularExpression("*button", false);
TestObject[] tobs = find(atDescendant(".class", buttonRE, "name", "OK");

Automation engineers can now create a library out of these framework classes, and test engineers can use them to create scripts easily. The following example shows how the test engineers use the functions in the framework to start an amazon.com, search for an iPhone, then sign into the account to buy the phone.

public void testMain(Object[] args) {
    startApp("www.amazon.com");
    sleep(5); // sleep for 5 second for amazon page to come up
    EnterText("field-keywords", "iphone");
    sleep(3); // sleep for 3 second for the next page to come up
    ClickButton("Go");
    sleep(3); // sleep for 3 second for the next page to come up
    ClickLink("Hello. Sign in Your Account");
    sleep(3); // sleep for 3 second for the next page to come up
    EnterText("email", "name@gmail");
    ClickPassword("password");
    EnterPassword("password", "password");
    ClickImage("Continue");
}

Conclusion

In this article, you have seen how Rational Functional Tester's find() API can be used to create an automation framework for an HTML-based application. The same approach can be used to write the framework for other domains that Rational Functional Tester supports.
Related topics

- Read all about Rational Functional Tester 8.5
- Watch Getting started with Rational Functional Tester
- Explore Rational Functional Tester supported domains (GUI technology)
- Improve your skills. Check the Rational training and certification catalog, which includes many types of courses on a wide range of topics. You can take some of them anywhere, anytime, and many of the Getting Started ones are free.
- Download a free trial version of Rational software.

© Copyright IBM Corporation 2013
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)