Using IBM Rational Functional Tester to automate testing of globalized applications

Level: Advanced

Komal Mirchandani (komal.mirchandani@in.ibm.com), Software engineer, IBM
Shruti Ujjwal (shujjwal@in.ibm.com), Software engineer, IBM

25 Sep 2007

If you want to develop test automation scripts for globalized applications, but you are having difficultly because the automated scripts recorded on a particular location fail when run at another location, this article will help. It explains an approach that enables you to seamlessly run a test automation suite developed in IBM® Rational® Functional Tester across different locations. The test automation engineer simply has to be a little more cognizant of the object properties to be able to use that knowledge in the test suite development for localized applications.

Prerequisites

- Installation of IBM Rational Functional Tester Version 6 or 7, along with the necessary language packs.
- Installation of operating system language packs for all of the necessary locations, depending on the requirements (Japanese, Chinese, and French, for instance).

An overview of test automation

Manual testing is time-consuming, labor-intensive, and frequently monotonous. It introduces problems, especially when there are limited resources and stringent deadlines. If you need to improve your application testing make sure that it works without a hitch, it's important that you move toward automating all of the manual testing tasks.

In today's environment of reduced cycle times, automated testing empowers both professional and novice users to quickly achieve high-quality results from application testing. Automation tools record user interactions against an application, and the scripts derived from these are then used for subsequent tests. In a nutshell, test automation enables you to optimize the quality of the complex application in a cost-effective manner within the timeline. This helps you produce higher quality software faster.

By using IBM Rational Functional Tester as a test automation tool, test automation is a three-step process::

1. **Record:** Recording test scripts on the fly, as users navigate the application. You can also insert verification points to validate system response, and make test scripts data-driven to execute the same script on a variety of data inputs.
2. **Enhance:** Adding code is to perform a variety of functions. Typical modifications to enhance test scripts include conditional branching, refactoring, and exception handling.
3. **Play back:** Running scripts to emulate the same actions that a user has performed on the application under test while recording. Discrepancies are logged and the tester can conclude whether functionality works fine or regression bugs have been introduced.

Typical problems in globalizing test automation

There are several challenges that a test automation engineer faces due to the changing software deployment trends. The current trend is that software development organizations and customers using them are geographically distributed, which means that applications must be *globalized*.

Globalized applications are those in which all of the strings, such as messages, labels, and text, are *localized*, that is,

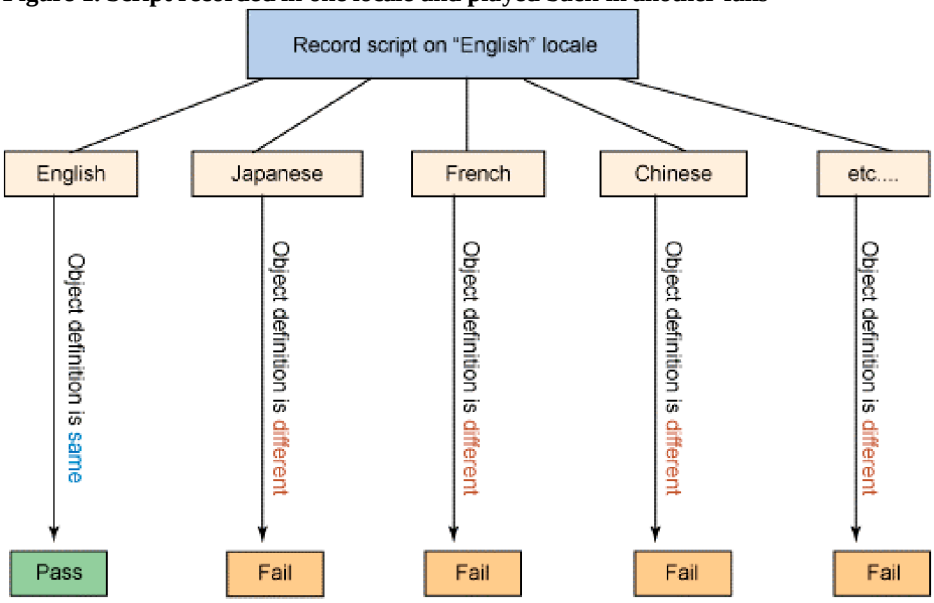
translated to the language of the location from which it was started or launched. For example, if an application is started from a Japanese locale or operating system, then all of the information will appear in Japanese. Similarly, if an application is started from a French locale or operating system, then he Information will appear in French, and so on. "Globalized" also refers to applications that allow input and output of a non-English character set.

The result is that, even after successful automation of such globalized applications, you may encounter these problems:

- **Automated scripts** recorded for a particular locale fails when played back in a different locale. This happens because the object definition used for playing back the automation script (for instance, the label of the button on which the automation script has to operate) may be different from an English to a Japanese locale. (See Figure 1.)
- **Verification points** checked in a particular locale fail when you attempt to affirm them in a different locale. This happens because the expected or initially recorded value of the verification point does not match the actual value displayed in the globalized application under test, because it was launched from a different location.
- **Data-driven test scripts** do not pick and populate the data set, depending on the location from which the application under test was launched. This happens because there is no implicit intelligence built into the automation script to help determine the language.

For example, Figure 1 depicts a globalized application that is launched initially on an English locale for recording the test scripts. The same script is later played back when the application is launched in the Japanese locale, but the script fails because the messages on the application under test were recorded in English. However, when the same application is launched in a Japanese locale, the messages get translated into equivalent Japanese text resulting in change of underlying object properties.

Figure 1. Script recorded in one locale and played back in another fails



Problem: Record/Playback model fails
Reason: The application under test is globalized

- Recorded on Locale 1 (English, for example)
- Played back on Locale 2 (Japanese, for example)
- **Result:** Script fails
- **Reason:** The object definition used for playing back the automation script is different for various locations.

Similar problems can occur with verification points and data-driven test scripts.

How automated testing of globalized applications works

A globalized application uses local resource files to show localized messages, labels, and text in the same application that will be launched from different locations. The approach described here is based on IBM Rational Functional Tester, and it uses the

locale resource files that are shipped with the globalized application. The locale resource file maps one-to-one between the object's property value and the variable corresponding to that value. This helps in picking the equivalent value of text from the resource file, depending on the locale from which the application was launched at the time of playback.

If you plan to globalize your test automation suite, you must deal with these object maps. The **object map** is nothing but a collection of all of the GUI objects in the application under test, with corresponding property values. You must select the property value (for example, the label of the button) and find the corresponding variable for it in the resource file (Figure 2). After the object property value is replaced with this variable, an underlying code fetches the value of the variable according to the current location (Figure 3).

Figure 2. A graphical representation of object mapping

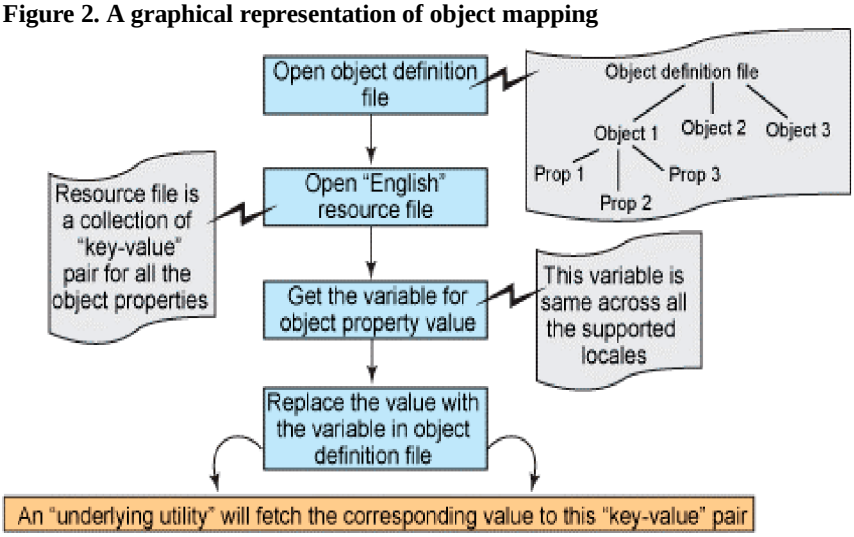
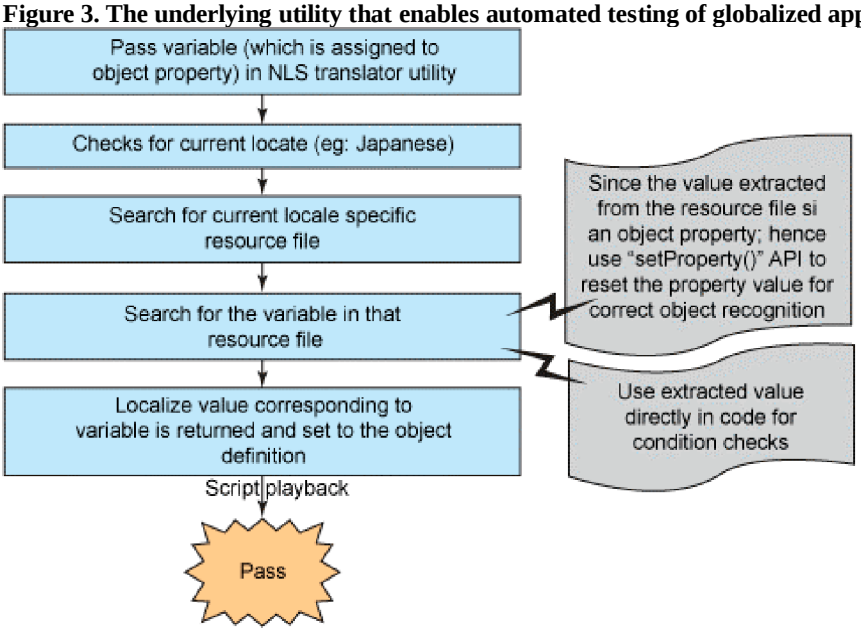


Figure 3. The underlying utility that enables automated testing of globalized applications



When the script is played back hereafter, rather than looking into the property value (which differs for each locale), Rational Functional Tester uses the variable, which is the same across locales. Thus, the script plays back smoothly. This approach makes the test automation scripts reusable and resilient to locale changes. It also allows automation scripts to surface defects in globalized applications without the extra effort of manual globalization testing.

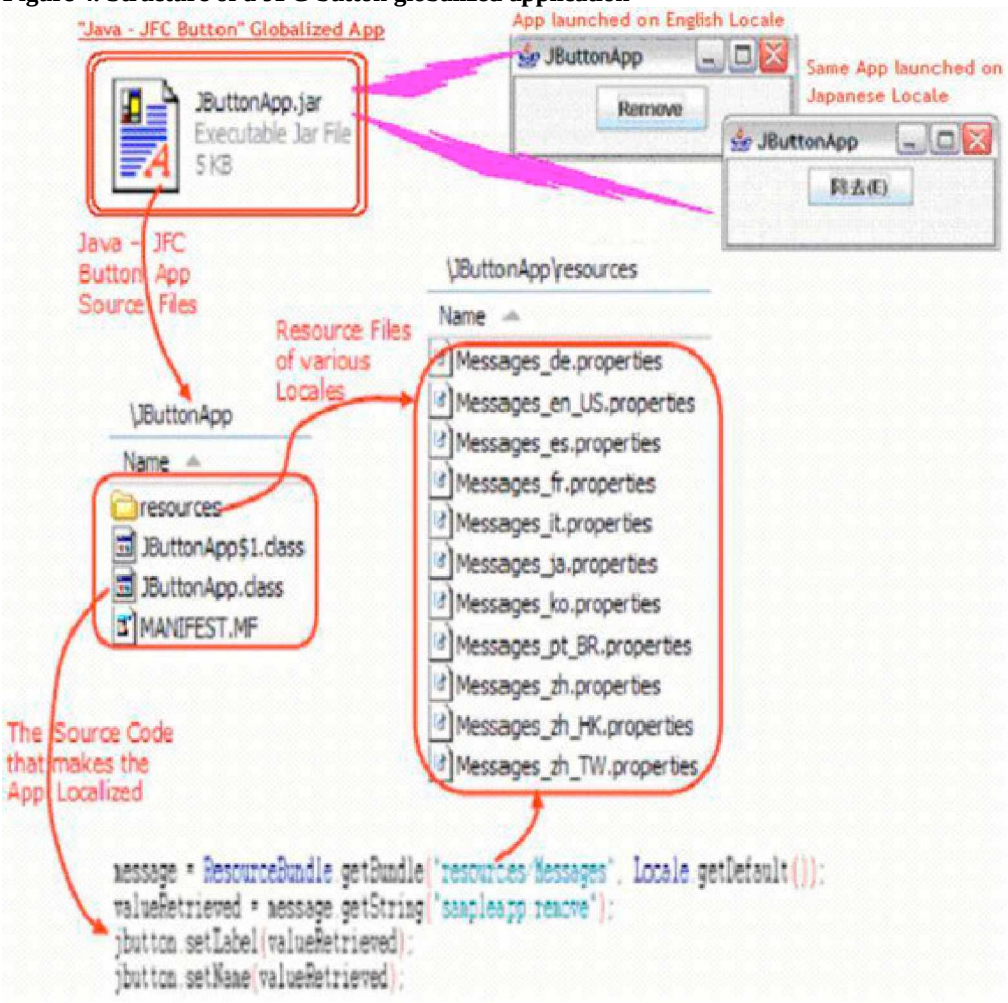
Steps to automate testing of a globalized application

To implement this approach, start with a sample Java™ foundation class (JFC) button globalized application. The

`accessibleName` and `name Properties` of this JFC button are localized, which indicates that the value corresponding to these properties will depend on the locale from which the application is launched.

Figure 4 shows how a globalized application is built and packaged. In this case, the JFC button application is an executable Java™ Archive (JAR) file. It has source code built as Java classes and is bundled with various locale resource files that make the text on the application translatable. Thus, when this application is launched in different locales, the text appearing on the user interface is read from the corresponding resource file in that locale. This is what makes an application globalized.

Figure 4. Structure of a JFC button globalized application



Record

To automate any globalized application that is recorded in just one locale and played back in different locales (such as Japanese, Chinese, or French), follow these steps:

1. Set the **Enable Localization** variable to **True** (`rational.test.ft.services.enable_localization=true`) in the `ivory.properties` file, which is available from the Rational Functional Tester installation location. (See the code in Listing 1.)

Listing 1. Snippet from the `ivory.properties` file, with the Enable Localization variable set to True

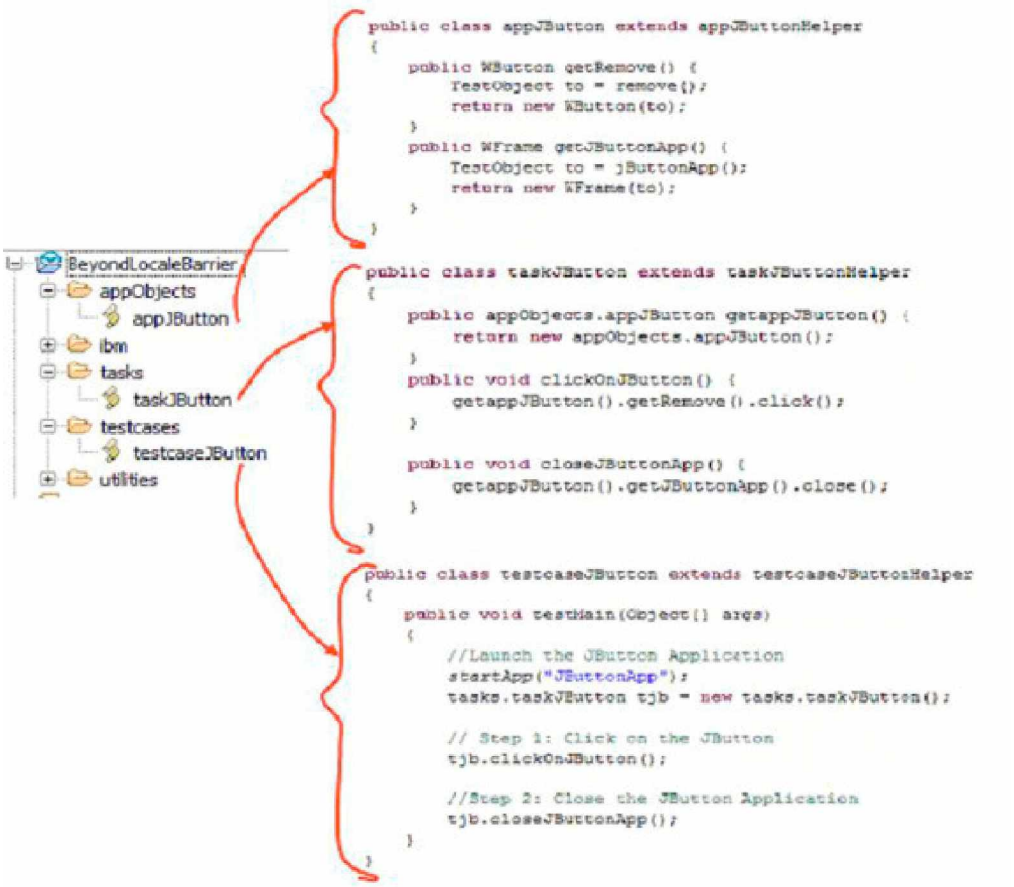
```
###
### Internal properties: Not intended for consumer modification
###
# Version number applied to the enabler.wsw plugin when an eclipse shell gets enabled
rational.test.ft.enabler.plugin.version=7.0.0
# rational client JVM startup options
#rational.test.ft.client.jvm_options=-xj9
# When enabled this option allows the install directory for the local
```

```
# TestContext to differ from the global setting
# (the install directory of the first TestContext created)
rational.test.ft.install_dir.ignore_mismatch=true
# When enabled this option allows recording / playback against product own UI
rational.test.ft.testability.allow_testing=true
# When enabled this property allows string lookup in the localized
# string table, if available
rational.test.ft.services.enable_localization=true
# Internal. Allow connecting to a .NET project for execution framework testing
#rational.test.ft.testability.allow_vbnet_remote=true
```

Step 2 leverages the use of IBM® framework (formerly known as ITCL) to develop test scripts in Rational Functional Tester. Using this framework ensures a structured approach to developing test script, and also offers other advantages: A layer of abstraction to the test automation scripts by organizing them into AppObjects, tasks, and test case layers. Minimized complexity and test scripts that are reusable and generic. A base set of library files that provides generic functionality to the automation harness for test script development and extension.

2. Use the IBM framework to organize the Rational Functional Tester script developed for the JFC button globalized application into three layers (shown in Figure 5):
    - **appObjects layer:** Creates a class called appJbutton, which is where the objects that the test script will interact with are stored.
    - **tasks layer:** Creates a class called taskJbutton, which is where the actual logic is written in form of various *tasks*, also known as *functions*.
    - **testCases layer:** Creates a class called testCaseJbutton, where the various tasks written in the tasks layer are used to make an end-to-end testing scenario.

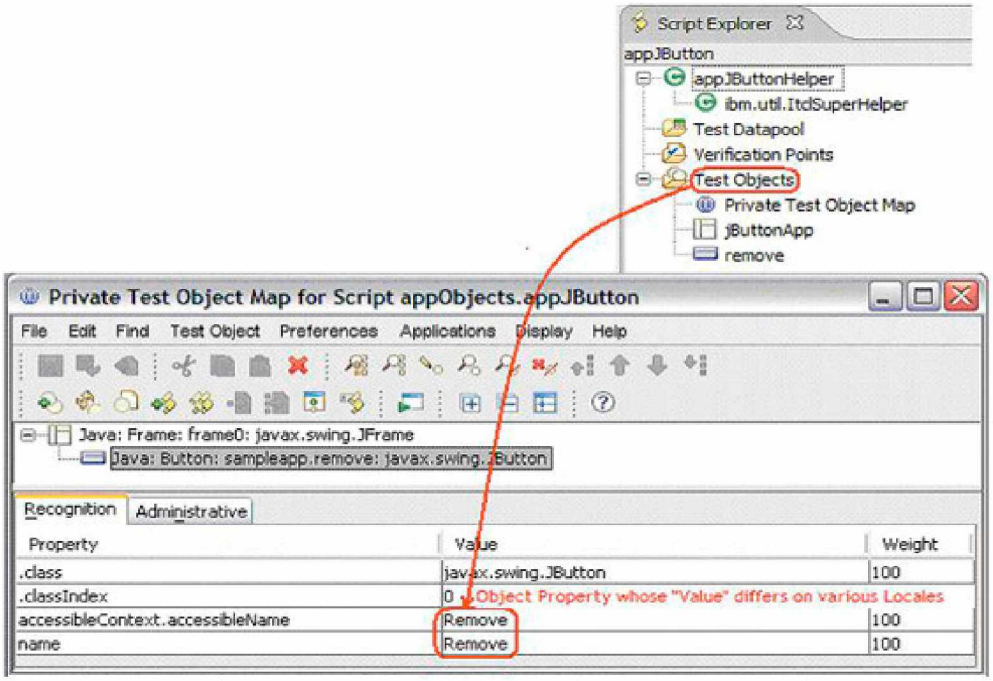
Figure 5. Using the IBM framework to organize the Rational Functional Tester scripts



3. Record a script for any locale (English, for example) by using Rational Functional Tester.
    A. Identify all object properties with values that vary in different locales.
    B. Now open the **object definition** file, which is known as the **object map** in Rational Functional Tester.

C. Choose the object property with the value that differs in various locations. In the case of this sample application, the `accessibleName` and `name` properties of the `javax.swing.Jbutton` object vary (see Figure 6).

Figure 6. Object map of a globalized application, object property with a varying value selected



4. Search the object property value in the JFC button application resource file. In this case, the varying object property value is **Remove text**. This is the value in the English locales, and there is equivalent text in each of the other locales (see Listing 2 and Listing 3).

Listing 2. Snippet from the English resource file for a key-value pair (variable-property value)

```
# NLS_MESSAGEFORMAT_VAR
sampleapp.remove = Remove
```

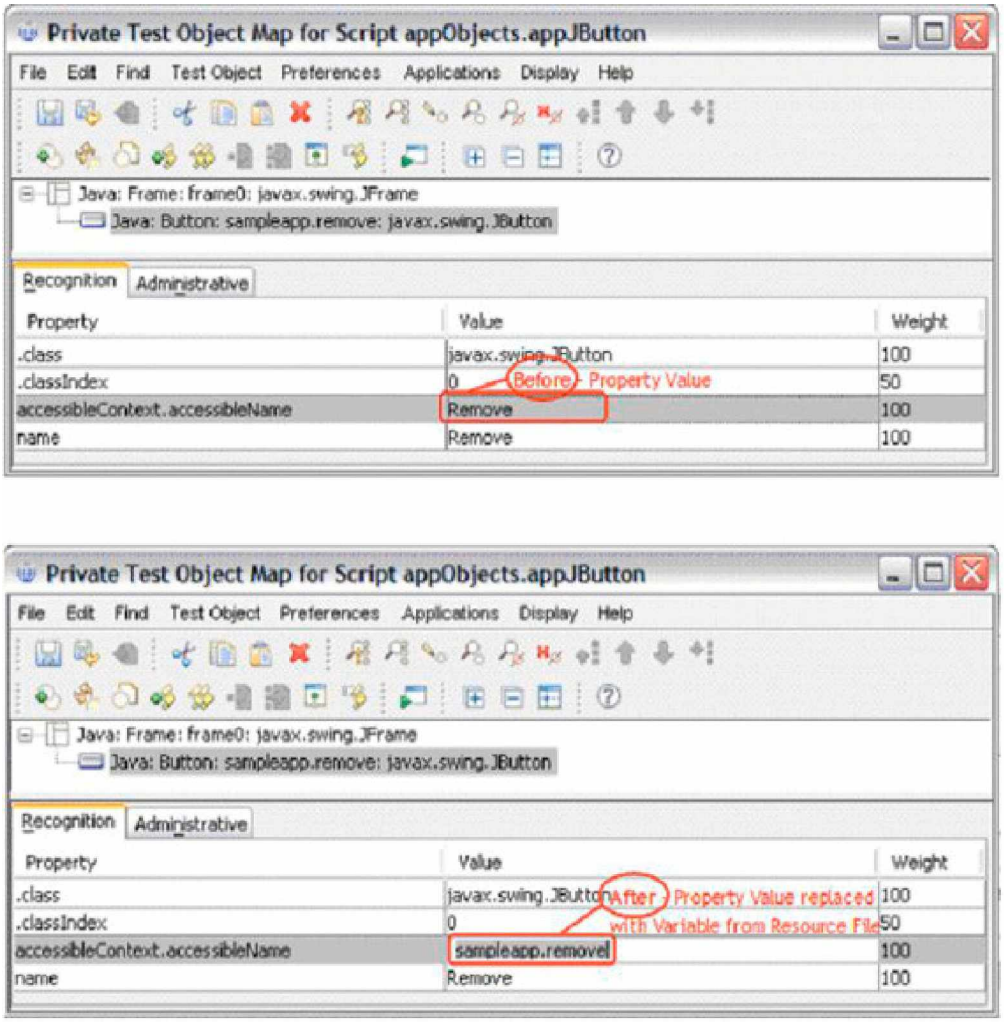
Listing 3. Snippet from the Japanese resource file for a key-value pair (variable-property value)

```
# NLS_MESSAGEFORMAT_VAR
sampleapp.remove = \u9664\u53bb(E)
```

Enhance

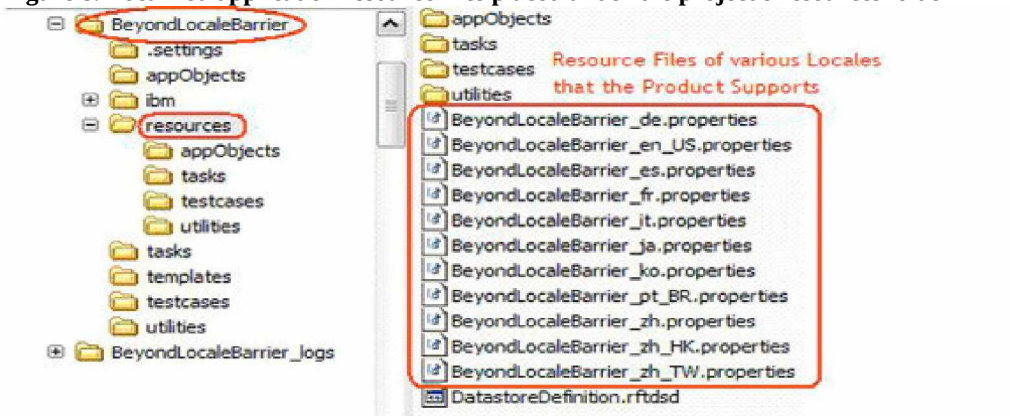
5. Replace the value in the Rational Functional Tester object map with the corresponding variable name found in the resource file. In this case, the property values for Properties: `accessibleName` and name of the `javax.swing.Jbutton` object was **Remove text**. This is replaced by the variable name `sampleapp.remove`, because this is the key corresponding to the Remove value (Figure 7).

Figure 7. Object map before (with property value) and after (with property value replaced with variable from resource file)



6. Place a copy of the all the localized application resource files under the project's resources folder.
   A. Change the name of all the locale resource files so that they start with the Rational Functional Tester project name and include the locale names. In this example, these locale resource files are renamed to start with **BeyondlocaleBarrier** (because that was the project name) and then the respective locale names are added to the file name (see Figure 8).
   B. Thus, for example, the file name for Japanese becomes: `BeyondLocaleBarrier_ja.properties`. This ensures that the value corresponding to the variable assigned to the object property, as shown in Step 5, is picked by the Rational Functional Tester scripts for appropriate recognition of the objects during playback.

Figure 8. Localized application resource files placed under the project's resources folder



7. Write a National Language Support (NLS) utility (as shown in Figure 9) that:
    - Accepts a variable corresponding to an object property value.
    - Checks for current locale (Japanese, for instance).
    - Searches for the current locale-specific resource file.
    - Searches for the variable in the resource file.
    - Returns the localized value:
        - Set it as object property value for correct recognition using the `setProperty()` API.
        - Use it for required conditional checks or verification points (Figure 10)

Figure 9. The NLS utility written to test the JFC button globalized application

```
public class NLSUtility {

    public static String converttoLocale (String myString) {
        // Location of the Resource Files
        String localizationfiles = "C:\\JButtonApp\\resources";
        String myResourceFile = "Messages";
        String myConvert = "";

        try{
            File file = new File(localizationfiles);

            // Convert File to a URL
            URL url = file.toURL();
            URL[] urls = new URL[]{url};
            ClassLoader loader = new URLClassLoader(urls);
            ResourceBundle myResources =
                ResourceBundle.getBundle(myResourceFile, Locale.getDefault(), loader
            myConvert = myResources.getString(myString);
        } catch(Exception e) { }

        return(myConvert);
    }
}
```

Gets the "Current Locale" on which the product is launched

Gets the "Value" for the "Key/Object Property Variable" depending on the Current Locale that was determined

Figure 10. Leveraging the use NLS utility to perform a conditional check or verification point to ensure that the expected button name matches the actual one

```
public void clickOnJButton() {
    String expectedName = "";
    String actualName = "";

    expectedName = NLSUtility.converttoLocale("sampleapp.remove");
    actualName = getappJButton().getRemove().getProperty("name").toString();

    if (expectedName.equals(actualName)) {
        logInfo ("The JButton Name: '" + actualName + "' is Correct.");
    } else {
        logInfo ("The JButton Name: '" + actualName  + "' is Incorrect.");
    }

    getappJButton().getRemove().click();
}
```

NLS Utility which retrieves the "Value" corresponding to Object Property Variable from the Current Locale Resource File

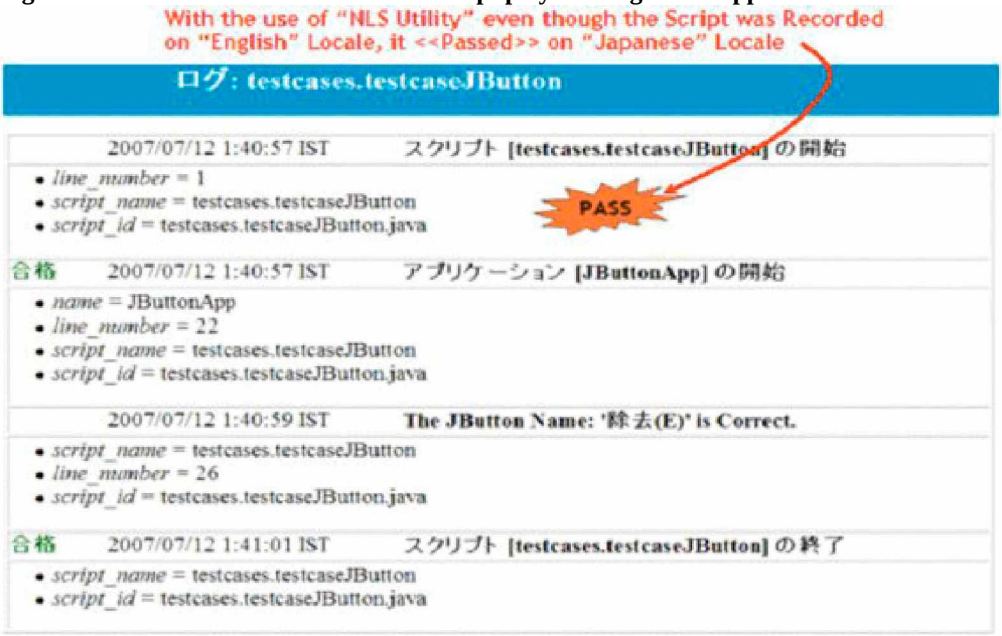
Play back

8. Play back the script on various locations, such as Japanese, Chinese, and French, and the test script will run successfully because it is now locale-independent (see Figure 11 and Figure 12)

Figure 11. Rational Functional Tester playing back the test script on an application launched in a Japanese locale, which is different from the one where it was recorded

Note:
With the use of the NLS utility, even though the script was recorded in an English locale, it passed in a Japanese locale

Figure 12. A Rational Functional Tester script playback log for an application launched in a Japanese locale



Advantages of this approach

There are several advantages of developing an automation harness for globalized applications, using the approach described in this article. A few of them are listed here:

- **Globalization regression testing**
  - Test automation teams can leverage this approach to build an automated regression test harness to test globalized applications build after build.
- **Record once, play anywhere**
  - Teams can develop automation scripts in an English locale and run the same script in other locations (Japanese, Chinese, French, and so on) without any modification to the script.
- **Using your time wisely**
  - If the test automation effort in one locale was, say, X days (for one tester), the test automation effort in nine locales will be 9*X days.
  - By automating globalized application testing using IBM Rational Functional Tester, the time spent will be a maximum of 2*X days.
- **Easy maintenance**
  - If there was a text or label change in application under test, only the resource file was replaced, rather than

having to change the automation scripts. This enables one-point updates of objects and their corresponding properties.

Resources

Learn

- Get an Introduction to IBM Rational Functional Tester 7.0. The IBM Rational Functional Tester tool automates testing Java, .NET, and Web-based applications. Starting with Version 7.0, it includes support extensions for both Siebel and SAP, plus integration with IBM Rational ClearQuest, support for the Eclipse Test and Performance Tools Platform (TPTP) logs, and support for testing HTML applications with Mozilla Firefox. This article explains these new features and capabilities.

- Read "Loading Siebel Test Data Using IBM Rational Functional Tester" See how you can use IBM Rational Functional Tester to create an automated data-loading utility to support Siebel testing. This process makes loading large sets of test data on Siebel CRM systems more efficient and accurate.

- Visit the Rational Functional Tester area on developerWorks for introductory to in-depth information.

- Visit the Rational software area on developerWorks for technical resources and best practices for Rational Software Delivery Platform. products.

- Subscribe to the IBM developerWorks newsletter, a weekly update on the best of developerWorks tutorials, articles, downloads, community activities, webcasts and events.

- Browse the technology bookstore for books on these and other technical topics.

- Visit the Rational software area on developerWorks for technical resources and best practices for Rational Software Delivery Platform products.

Get products and technologies

- Try Rational Functional Tester 6.1. The trial download is free but requires registration.

- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Get involved in the developerWorks Functional and GUI Testing discussion forum: A place for users of Rational Functional Tester and for the discussion of general testing topics.

- Check out developerWorks blogs and get involved in the developerWorks community.

About the authors

Komal Mirchandani is currently working as Software Engineer for IBM Rational where she is responsible for ensuring the quality of Rational TestManager and Rational Functional Tester. She has been with IBM Rational for around 3 years and has almost 5 years of industry experience. Her core skills comprise of estimation, design and execution of GUI test automation projects. She also has expertise in the QA, test automation and administration of enterprise applications and workflow products like ClearQuest. She has participated in various external customer engagements and has been continuously participating and presenting in various forums like: QSE, QAI, and TechConnect.

Shruti Ujjwal is a Prinicipal Software Engineer at the IBM Rational - India Software Lab, Bangalore. She works as part of the System Verification Testing Team and is responsible for testing Rational Functional Tester. She has strong automation background and is driving automation initiatives within Rational SVT organizations using the IBM-ITCL framework. Additionally, she has got expertise on Mercury's Quick Test Pro, Mercury's Winrunner and

White Box testing with Frameworks like: JUnit and XMLUnit. She also holds Brainbench Certification in Winrunner and Software Testing. She has been actively involved in various customer engagements and problem solving sessions throughout her stay in IBM. Additionally she has participated and presented in various forums like: QSE, RSDC, QAI, TechConnect, and for various teams in IBM-ISL and IGSI.