[Skip to content](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.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/ch16.html)
* [Expand Nav](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.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/ch16.html)

* [Search in book...](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html)

Top of Form

Bottom of Form

* [Toggle Font Controls](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html)
* [Share this](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.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/ch15.html)

[Chapter 15. Using Rational Functional Tester in a Linux Environment](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch15.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app01.html)

[Appendix A. Advanced Logging Techniques](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app01.html)

**Chapter 16. Internationalized Testing with Rational Functional Tester**

**Jeffrey R. Bocarsly**

*Both Java and VB.NET handle the full range of international character sets, and Rational Functional Tester and its objects do, too. With growing frequency, applications are built to handle a range of languages and the language-related customs that accompany internationalized applications. This chapter shows how to set Rational Functional Tester up to test applications that feature internationalized Graphical User Interfaces (GUIs).*

*Internationalized testing setup does involve working with script internals for those scripts that you want to execute with multilanguage support. You might want to review* [*Chapter 4*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04)*, “*[*XML and Rational Functional Tester*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04)*,”* [*Chapter 9*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09)*, “*[*Advanced Rational Functional Tester Object Map Topics*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09)*,” and* [*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)*,” before reading this chapter, so that the details are fresh in your mind. Furthermore, because this topic is considered an advanced topic, we treat it only from the perspective of the “regular” functional scripting apparatus (Java and VB.NET), and not from the perspective of the graphical Storyboard Testing option (released in Rational Functional Tester 8.1), which is meant to make Rational Function Tester more broadly accessible to nontechnical and novice users.*

**Unicode and Rational Functional Tester**

Application software, especially Internet-enabled software, increasingly is internationalized for global commerce. This means that systems are built so that their GUIs can display in a range of character sets and locales. The expanded character sets required for internationalized applications fall under the rubric of the Unicode standard. Unicode is the name of a major effort to standardize all human character sets, past and present, together under one system of platform-independent unique character mappings. At the time of this writing, Unicode 5.1.0 is the current release; it contains over 100,000 characters. The Unicode standard is under continual expansion, and new mappings for additional character sets are added regularly.

Note

The Unicode Consortium is the organization that develops the Unicode standard. You can get the flavor of the range of activities that the Consortium is involved in and look up character sets at the Unicode home page ([www.unicode.org](http://www.unicode.org/)).

Having a single character mapping for all characters, regardless of language, is only part of the answer. For the set of numbers representing each of these characters, there must be an encoding scheme. Numerous encoding schemes have been proposed, but the current *de facto* Internet standard is called UTF-8, which typically uses up to four bytes to encode a Unicode character (but may use up to six).

Note

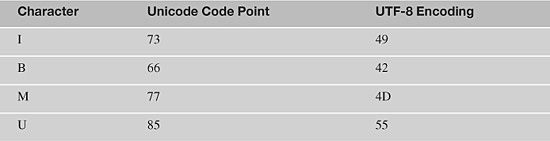
A complete description of UTF-8 and how it encodes Unicode code points can be found at [www.faqs.org/rfcs/rfc2279.html](http://www.faqs.org/rfcs/rfc2279.html). An excellent summary appears at [www.wikipedia.org/wiki/UTF-8](http://www.wikipedia.org/wiki/UTF-8).

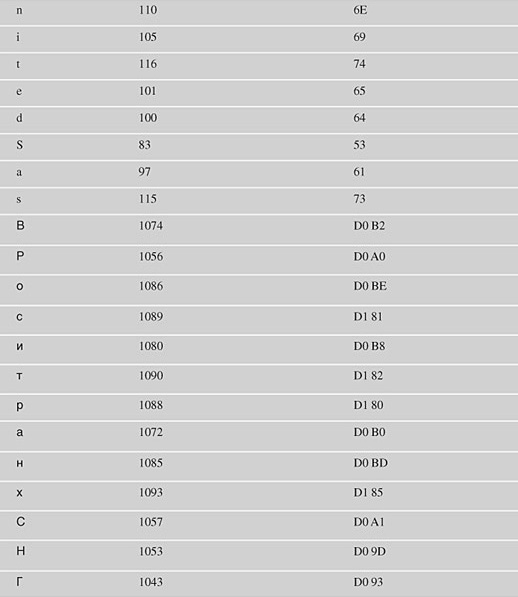
To make this a bit more concrete, let’s look at some examples of Unicode on the Internet. The IBM website is internationalized, and it can easily be viewed in numerous languages. If you examine the title tag for the home page in English and in Russian, you see the following:

image

Each of these title characters can be represented by corresponding Unicode values, and by the UTF-8 encoding of the Unicode code point. [Table 16.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16tab01) shows selections of these values.

**Table 16.1** Unicode and UTF-8 Sample Values





For the most part, with current text editors, you won’t have to deal with Unicode values directly or with the UTF-8 encodings. You just need the native fonts and texts for setting up an internationalized testing scheme. Of course, this likely requires access to resources fluent in each language, so the assembly of test data still won’t be as simple as with a single-language automation project (but you shouldn’t have to deal with collecting data as code points).

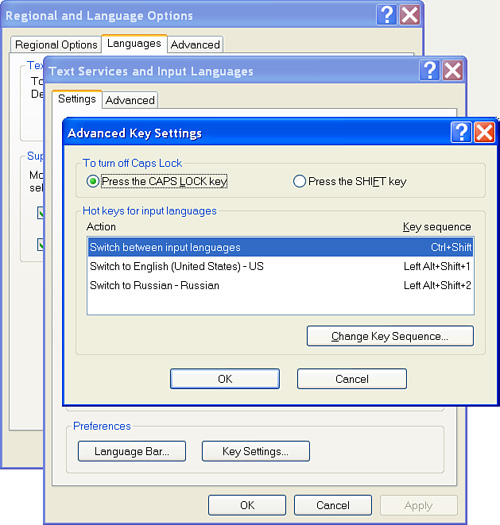
**Handling Internationalized Data in Rational Functional Tester**

Microsoft Windows offers broad language and locale support (a *locale* is the collection of writing practices and symbols used by a cultural group—for example, the United States locale uses the English alphabet, formats dates in the order month-day-year with either a dash or a slash delimiter, and for numbers, it uses a comma to separate thousands, a point to indicate the decimal place, and the dollar symbol ($) to signify money, and so on). You can have multiple languages and locales loaded on Windows at any given time. For information about how to set up internationalized support on Windows, read the article “Enabling International Support in Windows XP/Server 2003 Family” (<http://www.microsoft.com/globaldev/handson/user/xpintlsupp.mspx>). Numerous articles on the Microsoft website and on the Internet describe how to enable language support on Windows. For performing internationalized testing on Windows, setting up Windows with the different language-locale combinations that you need is the first step.

**Language Support in Windows and the Windows Language Bar**

After you have set up language support, you can access the different language-locale combinations you have chosen through the Windows Language bar (which displays as a floating toolbar or as a static toolbar next to the system tray, depending on your configuration choices). Using the Language bar, there is one additional Windows feature used in the implementation described in this chapter. Windows enables you to associate specific hotkeys with specific *keyboard* selections. This is done with the Windows Language bar Settings window (Text Services and Input Languages). You use the Key Settings button in the Preferences section of the window (see [Figure 16.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16fig01) for the dialog window). This chapter uses the keystroke sequence Left+Alt+Shift+num for each keyboard/language, where the value of num uniquely identifies each language.

**Figure 16.1** The Windows Language bar hotkey configuration dialog



**Language Support in Rational Functional Tester**

Language support in Rational Functional Tester is present by default. It’s already set up! All Rational Functional Tester objects (except the Console Views in both Eclipse and Visual Studio) can display internationalized text, whether you capture it in a script by recording, for example, or whether you type it in a datapool.

**File Formats for Handling Unicode Data**

Unicode data can be handled with any file format, either standard or custom, as both Java and VB.NET handle Unicode characters natively. As XML increasingly becomes a standard for storing data in files on disk (this is the .NET approach, and the Rational Functional Tester approach, too), the XML format is used for this chapter’s discussion.

**Unicode Editors and Files on Windows**

On the Windows platform, Unicode text is most simply handled using Notepad, WordPad, or any of a variety of other Unicode-enabled text editors. If you plan to use Notepad, simply remember to save your file using the UTF-8 encoding. WordPad saves in Rich Text Format or Unicode text format, both of which handle Unicode characters. Because XML format is used for this work, you can use any XML editor that handles Unicode or any text editor (such as Notepad) that is Unicode-enabled.

You can type internationalized characters in your editor window by changing the editor window’s keyboard using the Windows Language bar to any international keyboard that you have loaded. You may have to change the file encoding to a suitable choice (UTF-8, Unicode text, Rich Text Format, and so on) when you save to properly save your internationalized text.

Note

Although you can capture internationalized names in a script, this does not mean that you can save the script with international characters in it by default. In Eclipse, this is because the default text file encoding on Windows is Cp1252, and it is likely that many or all of your international characters will not map to this encoding, which triggers the error when you try to save your script. If you want to save Unicode characters in your script class, you must change the text encoding of the file. This is done at any one of three levels: You can change the encoding for just the script, for the entire Rational Functional Tester project, or for the Eclipse instance. To modify just the script, right-click on the script in the Functional Test Projects view, select Properties, and in the script Properties window, select Resource in the treeview and in the Text file encoding section, change the encoding. To change the text file encoding at the project level, right-click on the project in the Functional Test Projects view, select Properties, and in the project Properties window, select Resource in the treeview and change the file encoding in the Text file encoding section. If you want to change the file encoding at the level of the Eclipse instance, open the Eclipse Preferences window (Window Menu > Preferences), and then navigate in the treeview to General > Workspace. You make the change, as before, in the Text file encoding section. At any of these levels, you can set your text file encoding to UTF-8 (or to one of a number of encodings) to be able to save with Unicode characters. Visual studio will automatically prompt you to change a file’s default encoding from Cp1252 when it detects Unicode characters in the file.

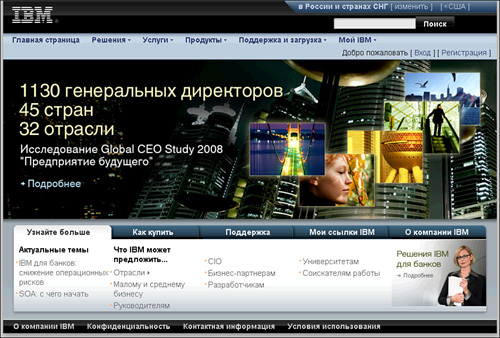
**Setting Up an Internationalized Test in Rational Functional Tester**

To illustrate how to set up an internationalized test in Rational Functions Tester, pages from the IBM website are used; they can be configured to display in a number of languages. For the examples shown in this chapter, the United States English homepage and the Russian homepage are used (see [Figures 16.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16fig02) and [16.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16fig03)). Simple data verifications on the page along with some standard object manipulations are performed to illustrate the setup for an internationalized testing project.

**Figure 16.2** The IBM United States English homepage

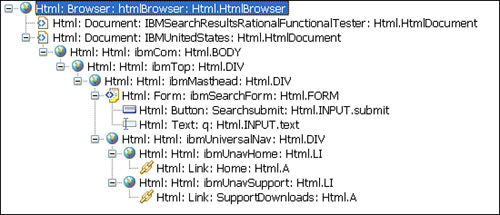


**Figure 16.3** The IBM Russian Federation Russian homepage

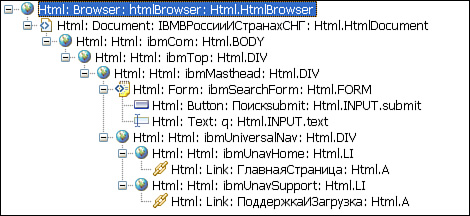


Note that one critical assumption for the setup discussed here is that each language version of the application must have the same object hierarchy (see [Figures 16.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16fig04) and [16.5](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16fig05)). This might be true of your application; if it is not, you need to take object hierarchy differences into your automation strategy. Be aware that this can add significant additional work to your project.

**Figure 16.4** The IBM United States homepage Object Map hierarchy (partial)



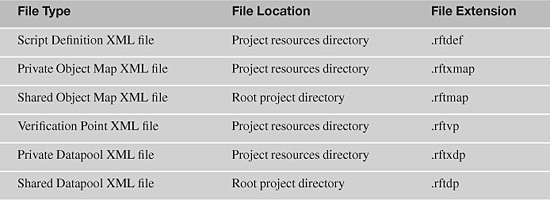
**Figure 16.5** The IBM Russian homepage Object Map hierarchy (partial)



**Rational Function Tester Script Files**

The overall strategy for implementing internationalized testing involves the following Rational Functional Tester script files. Rational Functional Tester can use up to four general types of supporting XML files for each script to persist script data. (There are actually six file extensions because both Object Maps and Datapools each have two sub-types.) XML file locations and extensions are shown in [Table 16.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16tab02).

**Table 16.2** Rational Functional Tester Script Files: File Locations and Extensions



The Script Definition and Object Map files are discussed in [Chapter 9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09). The other two types of files—the Verification Point XML and the Datapool XML—persist for each script its Verification Point (VP) data or Datapool data for the script. A script might have multiple Verification Points, and therefore, multiple Verification Point files associated with it. Only a single Datapool can be associated with a script, just as only a single Object Map can be associated with a script. Of the four types of files, only three (the Object Map, the Verification Point, and the Datapool) contain actual object data that might use internationalized characters.

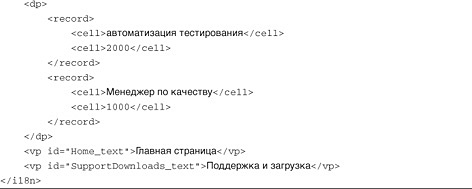
**Custom Internationalized XML for Unicode Testing Data**

To set up an internationalized testing implementation, object or transaction data in each of the Rational Functional Tester script files must be changed to suit each language version of the target application. This means simply that if you’re testing the English version of the application, all the data values must be in English, and if you’re testing the Russian version, all the corresponding values must be changed to their Russian equivalents, and so on for other languages. Because this chapter extends the Rational Functional Tester approach of using an XML format to store data, the approach described here needs a simple XML grammar to support the storage of the Unicode test data.

The XML grammar used here is illustrated in [Listing 16.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex01). The <i18n> root tag has two attributes: One designates the language of the character data in the XML by its two-letter ISO code, and the other identifies the unique hotkey number to activate the language keyboard for the specified application window. This number is used as part of the hotkey keystroke sequence for each language configured using the Windows Language bar. You can see the full list of ISO language codes at (see <http://www.loc.gov/standards/iso639-2/php/code_list.php>).

**Listing 16.1** Custom internationalized Rational Functional Tester XMLs





Beyond the root tag, there are just four child tags: <url>, <obj>, <dp>, and <vp>. The <url> tag holds the language-specific URL for each version of the application; the <obj> tags contain the language-specific object information, and the <dp> and <vp> tags contain the Datapool and Verification Point data. A couple of additional comments are in order. The <obj> and <vp> tags each must have an id attribute that identifies the TestObject involved. There can be either one of two types of data in an id attribute: the object name as it appears in the Rational Functional Tester Script Explorer (and therefore, in the Script Definition XML) or the Rational Functional Tester GUID identifier for the object (which appears in the Object Map XML, the Script Definition XML, and as the Map ID property on the Administrative tab of the Object Map tool). The reason that both of these object identification types are supported is because you might have to manipulate the property value for an object that is not in your script’s Script Explorer, but *is* in the hierarchy of an object that is in your Script Explorer (see [Chapter 9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09)). The type attribute of the <obj> tag indicates which kind of id is used by each <obj> tag; if the type attribute has a value of ‘name’, the Script Explorer object name is used, and if it has a value of 'id', the Rational Functional Tester GUID is used.

You can avoid the use of the Rational Functional Tester GUID by putting the parents of all objects explicitly in the Script Explorer, but there is a downside to loading the Test Objects folder in your Script Explorer with a large number of objects; as the underlying Script Definition XML grows, the time required to look up objects in this XML grows. For relatively short Script Definition XMLs, adding several additional objects might not make much difference; however, eventually you might slow your script execution significantly.

The whole internationalized testing game at this point resolves to building code to take the data from the custom internationalized XMLs in [Listing 16.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex01) and save it to the appropriate Rational Functional Tester XML file. The example here uses a simple design for housing this code: a driver script that handles all of the data copying, changing, and saving, and that also calls the “testing” script(s), each of which runs multiple times using different internationalized underlying data for each execution run, with no changes to the actual script code.

**Internationalized Driver Script Design**

As noted previously, the driver script needs to accomplish two broad tasks for each language it handles. It needs to move language-specific Unicode character data from the custom internationalized XML files to the Rational Functional Tester XML files for each specific “testing” script and it needs to call that script; then it needs to move data for the next language into the “testing” script XML files and call the script again, until it cycles through all the languages that are queued up and all the “testing” scripts that are queued up.

**Driver Script Utility Methods**

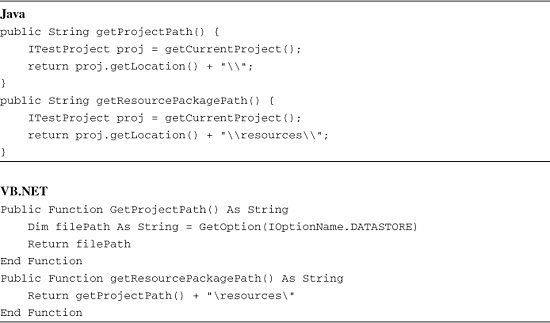
To start setting this up, this chapter’s example takes advantage of Rational Functional Tester’s XML file naming convention and extends it to use convention-based names for the custom internationalized XML files that are developed as part of this implementation, such as those in [Listing 16.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex01). To implement, a library of methods are needed that can generate the file paths for the XML files and load the XMLs into the XML DOM, so that scripts can modify the XMLs as necessary. For this latter work, the example developed here relies on methods discussed in [Chapter 4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04). (You may want to refer back to that chapter while you read this material.) All the utility methods that are developed here to support internationalized scripts are housed in a utility class in a project called I18NUtils.

Warning

The code you explore in this chapter operates directly on underlying Rational Functional Tester and Project files. These files are normally manipulated only by Rational Functional Tester. Back up all files that the code touches before you execute any of these methods!

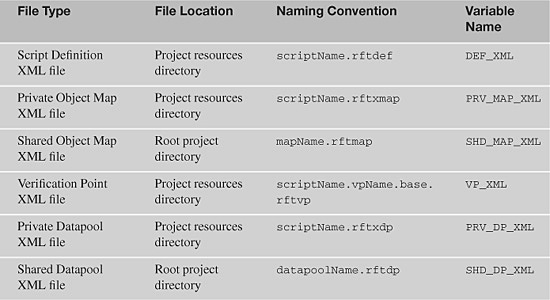
Start with a couple of simple methods that return the path to the current project directory and to the project’s resources directory (see [Listing 16.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex02)). These methods are straightforward and are just wrappers for methods from the Rational Functional Tester API.

**Listing 16.2** Methods to return project directory locations



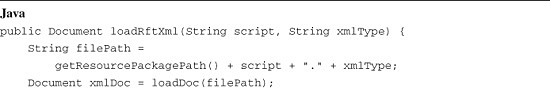
Constructing the path for a given XML file is relatively straightforward. We start by getting the file path from one of the methods in [Listing 16.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex02). The Rational Functional Tester file-path convention is simple: for the Private Object Map, Script Definition, and Private Datapool XML files, the file location is in the project resources directory, and the XML file naming convention is the script name followed by the XML extension. (See [Table 16.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16tab02) for details.) For a Shared Object Map or a Shared Datapool, the file location is at the root project directory, and the files appear with their own unique extensions ([Table 16.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16tab02)). For these Shared Map or Datapool files, the file-naming convention is the Map or Datapool name with a file extension identifying the type of XML. For Verification Point files, which are found in the project resources directory, the file-naming convention changes slightly; the script name is followed by the Verification Point name with a period delimiter, followed by the “.base” designation, and ends with the Verification Point extension ([Table 16.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16tab02)). These naming conventions are summarized in [Table 16.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16tab03). The file types, locations, and variable names used in the code Listings that hold the file extensions are also shown in [Table 16.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16tab03).

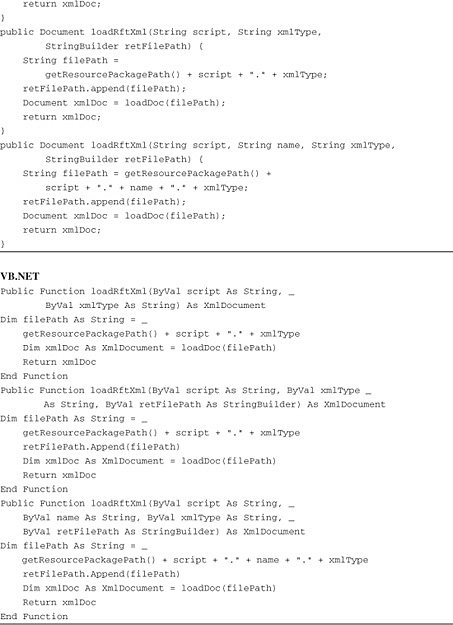
**Table 16.3** Rational Functional Tester Script XML File Naming Conventions



The methods in [Listing 16.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex02) are used as building blocks (along with methods from [Chapter 4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04)) for methods to load the required XML files (both Rational Functional Tester script XMLs and custom internationalized XMLs). The first of these methods has the following design: It constructs the file path for a chosen Rational Functional Tester script XML file and uses the method loadDoc() (from [Chapter 4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04)) to load the XML file into the XML DOM, and then returns a reference to the DOM. In addition, an overloaded method is shown that adds an argument to return the constructed path to the calling routine, so that the path to any file is available to the calling code for file management purposes. [Listing 16.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex03) shows these overloaded methods.

**Listing 16.3** Overloaded Rational Functional Tester script XML file loading methods

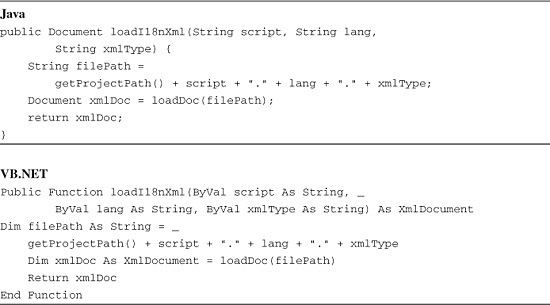




The sample project uses a Private Object Map and a Private Datapool, so all the files are in the Rational Functional Tester project resource package. The methods in [Listing 16.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex03) for obtaining the complete file path for each file type are simple. These methods require just the script name, an optional field name (for the extra field that Verification Point filenames have), and the file type to build any of the XML filenames.

The final method needed for this part of the work is a method to load the custom internationalized XMLs ([Listing 16.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex01)). For convenience, the custom files are stored in the project root directory with the script .java files (not in the resource directory with the script XML files). The advantage this affords is that Eclipse automatically picks up the presence of these files and shows them in the Project Explorer, and they can be viewed or modified using Eclipse’s convenient XML viewer. The naming convention for these custom files is ScriptName.la.xml, where the field “la” corresponds to the two-letter ISO language code. To keep this method parallel in structure to similar methods in [Listing 16.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex03), it takes three arguments: the script name, the language code, and the file extension. [Listing 16.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex04) shows the method.

**Listing 16.4** Custom internationalized XML load method



**The Driver Script Framework**

With the utilities illustrated in the previous sections, the driver script framework can be built. The driver script framework handles looping through scripts and languages, moving Unicode data into the supporting Rational Functional Tester script files in the resources directory, executing the internationalized script under each language, and performing file management so that the underlying files are protected from accidental deletion and the internationalized script is returned to its original language at the end of script execution.

For convenience, the driver framework is illustrated using just a single target internationalized “testing” script and just two languages (English and Russian). The driver framework must handle the following detailed tasks:

• Looping through scripts

• Looping through languages

• Loading Rational Functional Tester script XML documents

• Copying the original Rational Functional Tester script XMLs for safekeeping

• Modifying the configurations.rftcfg file for each i18n URL

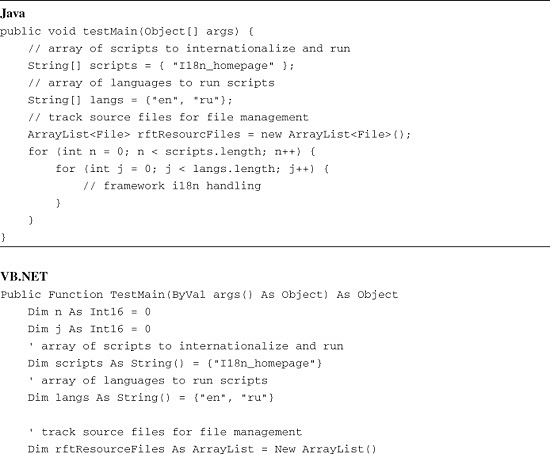
• Modifying the Object Map, Verification Point, and Datapool XMLs with Unicode data

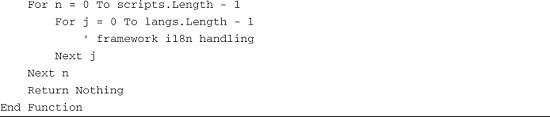
• Executing the script for each language configuration

• Cleaning up and returning the script to its original language configuration

The driver script starts with arrays and loops in the script testMain() method to handle the looping through scripts and languages. [Listing 16.5](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex05) shows this basic framework. Two arrays—scripts and langs—are needed to hold the names of the testing scripts called by the driver (script I18n\_homepage, in this example) and the languages that the scripts run under (English and Russian). An ArrayList (called rftResourcFiles) is also created that is used to track the names of the Rational Functional Tester XML files.

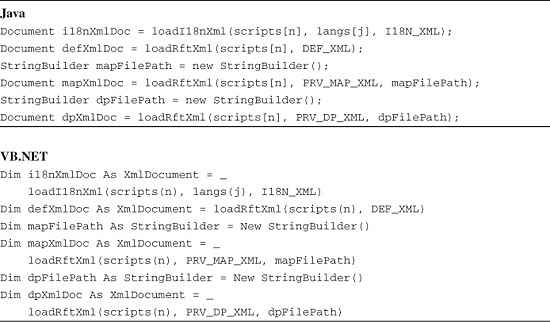
**Listing 16.5** Basic framework of Driver script for internationalized testing





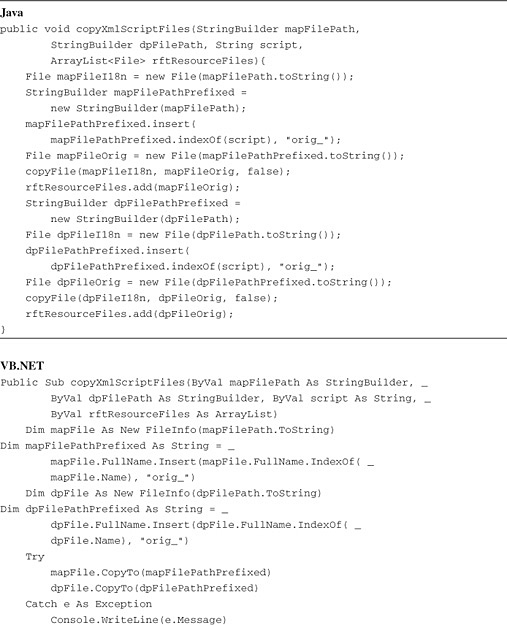
The next step in building the framework is to handle loading the XML documents that are needed. This is the first task in the inner loop. To do this, the script uses methods from [Listings 16.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex03) and [16.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex04). For the files that are modified with Unicode data, the file path is captured in the driver code so that the driver can manage the files during execution (by copying the original files to a safe location during runtime, and restoring the original files at the end of execution). [Listing 16.6](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex06) shows the code.

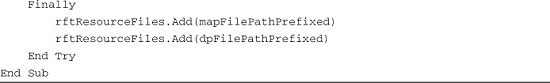
**Listing 16.6** Loading the Rational Functional Tester Script XMLs and the custom internationalized XML



For the next task, the framework needs to protect the original XML files by copying the files to a safe location. The framework does this by prepending the string “orig\_” to the filenames and copying the files to their home directory (the resources directory) under the modified names. In addition, the filenames are captured to the ArrayList created for this purpose (rftResourcFiles) so that later the driver script can perform file management on the files to return the Rational Functional Tester script to its original state. [Listing 16.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex07) shows a method for these steps.

**Listing 16.7** Copying the original Rational Functional Tester script XML files for safekeeping





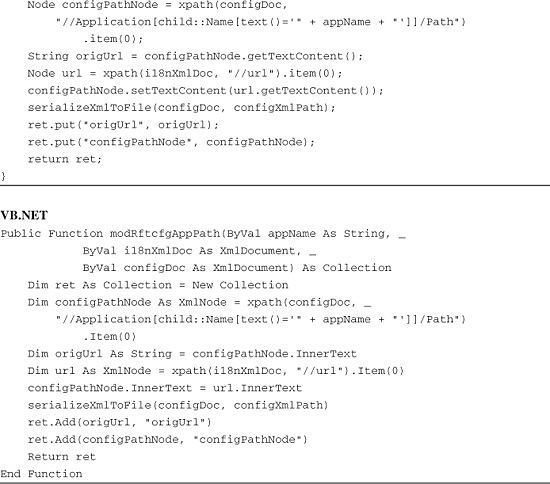
In [Listing 16.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex07), in Java, the file copy is performed using the method copyFile(), which employs the efficient Java “new I/O” classes (java.nio.\*) to copy the files (for discussion, see <http://java.sun.com/developer/JDCTechTips/2002/tt0507.html>; full code can be found in the supplementary materials, of course). In VB.NET, the CopyTo() method of the FileInfo class is used.

The next task for the driver is a bit off the beaten track. This is a modification of the configurations.rftcfg file. This file (in Windows, it is found at C:\Documents and Settings\All Users\Application Data\IBM\RFT\configuration\configurations.rftcfg; in the code listings, this path is denoted by the variable configXmlPath) is where Rational Functional Tester persists information about applications configured for testing. Because we’re going to launch each language version of the application as a separate application instance, we need to modify this file so that the correct language version of the application is launched for each language. It is worthwhile to pause a moment to note that you don’t actually have to use this approach; you can configure each language version as a separately configured application and pass each configured application name into each script. This is a perfectly reasonable solution; however the approach of modifying the configurations.rftcfg in code is used here to provide an example of how this can be done.

The key thing to be aware of for modifying application configuration data in the configurations.rftcfg XML is that the critical information for the application configurations appears in <Application> tags. In the example, the configured application is named ‘ibm homepage,’ so we’re looking for an <Application> tag with a child <Name> tag with a value of ‘ibm homepage.’ Rational Functional Tester won’t let you create two configured applications with the same name, so the value of this tag must be unique. After a reference to the correct <Application> tag has been found, all that needs to be done is to modify its child <Path> tag to hold the proper URL for each specific language version of the application, which is stored in the custom internationalized XML. In making this change to the <Path> tag value, the code should also capture the original value of the tag (in the origUrl variable) so that it can be reset when the driver performs housekeeping tasks. Finally, this is the first Rational Functional Tester XML that the driver modifies, so once the XML is loaded into the DOM, it should be saved to disk. This is performed with the method serializeXmlToFile() from [Chapter 4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch04.html#ch04). [Listing 16.8](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex08) shows the method.

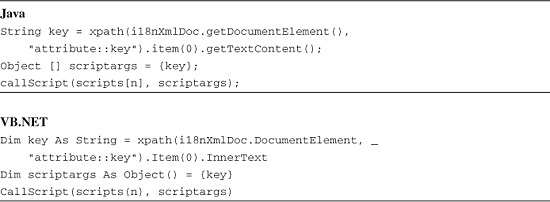
**Listing 16.8** Method to modify the Rational Functional Tester configurations.rftcfg file





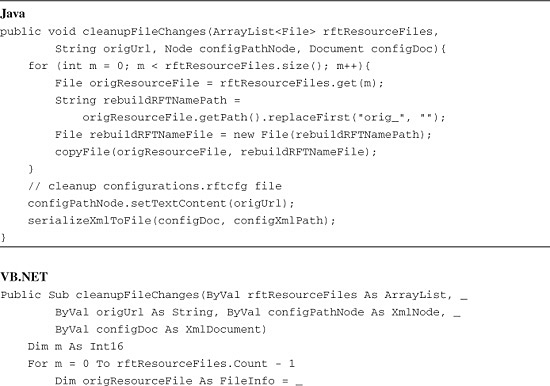
With regard to the driver framework, there are two more tasks that the framework needs to support: the actual execution of the target internationalized script and housekeeping tasks. The execution of the script is performed as usual with the callScript() method using the target script name (from the scripts[] array, see [Listing 16.5](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex05)); however, the called script must be passed an argument—the hotkey number that the script can use to change the keyboard of the application window to the desired language keyboard. You might recall that these key numbers are set up using the Windows Language bar functionality as part of configuring Windows to handle multiple input languages (see the section “[Handling Internationalized Data in Rational Functional Tester](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16sec1lev2)” for details). The hotkey numbers for each language are kept in the custom internationalized XML, in the key attribute of the root <I18n> tag. In the driver script, the key value is passed in as an argument to the script; the script has to handle the argument in its implementation. [Listing 16.9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex09) shows the script calling code.

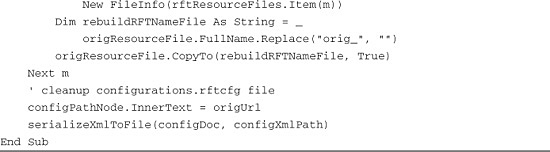
**Listing 16.9** Calling the target internationalized script



As noted, the housekeeping tasks make up the last part of the framework. The variable origUrl holds the original URL for the application ([Listing 16.8](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex08)) and the rftResourceFiles ArrayList holds the file paths for the original script files; these are used to return the script to its original state. [Listing 16.10](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex10) shows a method for these tasks.

**Listing 16.10** Housekeeping method for the internationalization driver script

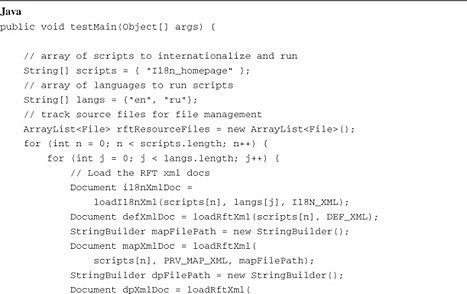


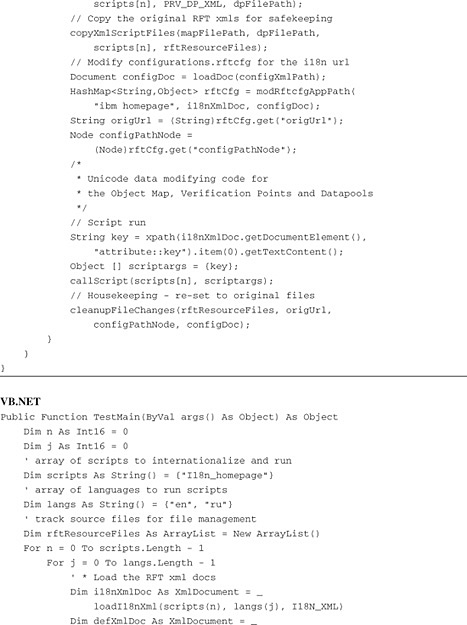


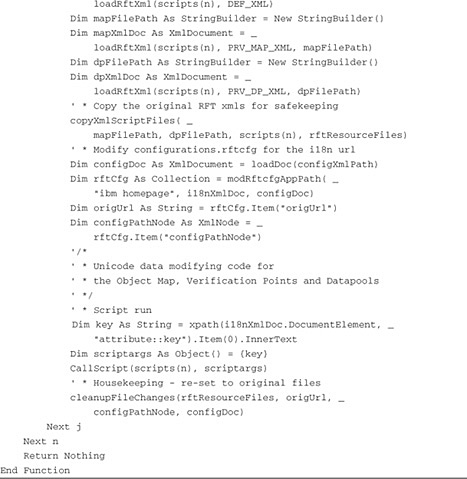
**Putting the Driver Script Framework Together**

We’ve reviewed all the pieces of the driver script framework, but it can be hard to “see the forest from the trees” when there are multiple coding tasks in a block of automation. [Listing 16.11](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex11) shows the full framework, with all the coded framework tasks in sequence; this example lacks only the detailed code to move Unicode data from the custom internationalized XML into each of the Rational Functional Tester script XMLs. The topics following [Listing 16.11](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex11) show the Unicode data-handling tasks for the Object Map (“[Modifying a Language-Specific Object Map](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16sec1lev4)”), for Verification Points (“[Modifying Language-Specific Verification Points](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16sec1lev5)”), and for Datapools (“[Modifying Language-Specific Datapools](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16sec1lev6)”).

**Listing 16.11** The complete driver script framework







**Modifying a Language-Specific Object Map**

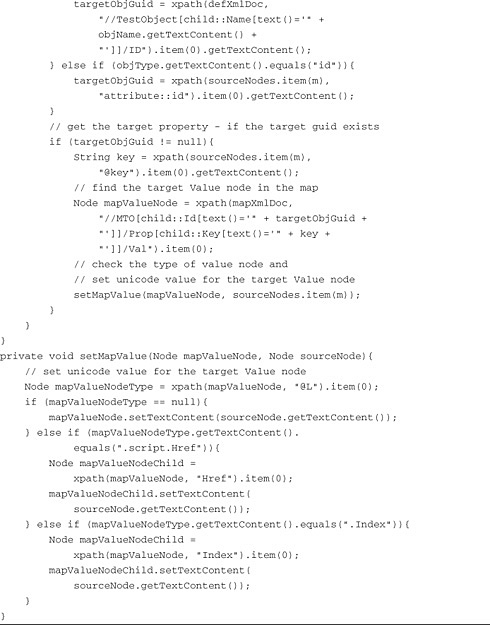
Modifying the Object Map for language-specific Unicode data is a multistep process. As noted previously, objects that appear in the Script Definition can be looked up by name in the Script Definition XML, and can be cross-referenced (using each object’s Rational Functional Tester GUID) to the Object Map XML. For those objects that do not appear in the Script Definition, but do have Unicode data (such as objects high in the object hierarchy that might not be in the Script Definition), their Unicode data has to be stored in the custom internationalized XML under their Rational Functional Tester GUIDs and not under their map object names. (You can review this by looking at [Listing 16.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex01), where some objects are identified by Script Definition name and others are identified by Object Map GUID.)

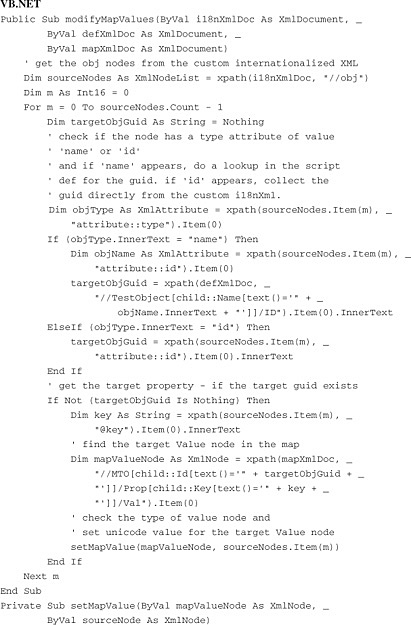
The Object Map modification starts with the structure of the custom internationalized XML ([Listing 16.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex01)). All object information—whether classified by Map object name or Rational Functional Tester GUID—is found in <obj> tags. So, the implementation starts by collecting the <obj> tags from the XML. The type attribute of the <obj> tag either has a value of ‘name’ if the tag describes a mapped object by name or ‘id’ if the tag described a mapped object by GUID. If the value of the type attribute is ‘name,’ the method must perform a lookup in the Script Definition XML to find the object’s GUID, and if the value is ‘id,’ the GUID is the text value of the <obj> tag. By either route, the implementation code identifies the Rational Functional Tester GUIDs of all of the objects that are in the custom internationalized XML. There is one additional piece of information needed from the custom internationalized XML <obj> tags, and that is the value of the key attribute for each tag, which describes the Rational Functional Tester recognition property whose value changes for each language in the internationalized test (.text, .href, .value, and so on). The strategy is to use the object GUID and then to use the key value, to manipulate the target Object Map recognition property data to appear in the desired language.

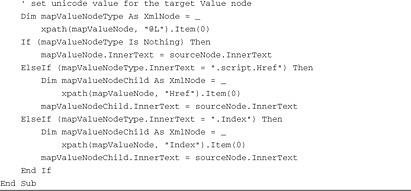
The method design that implements these tasks (shown in [Listing 16.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex12)) breaks this process into two methods: The first, modifyMapValues(), handles the lookups of the XML nodes, and calls the second method, setMapValue(), which modifies the node values of the Map XML DOM. Note that neither of these methods saves the data to file; you must call serializeXmlToFile() as usual to perform a save action.

**Listing 16.12** A method to modify Object Map recognition property values









Because of the need to save the XML to a file after the modifications are made, the actual calling sequence for these methods in Java is as follows:

Image

In VB.NET, this looks like the following:

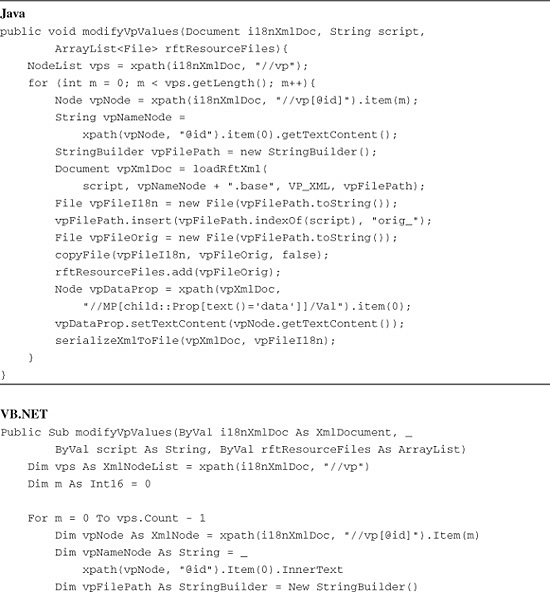
modifyMapValues(i18nXmlDoc, defXmlDoc, mapXmlDoc)  
serializeXmlToFile(mapXmlDoc, mapFilePath.ToString)

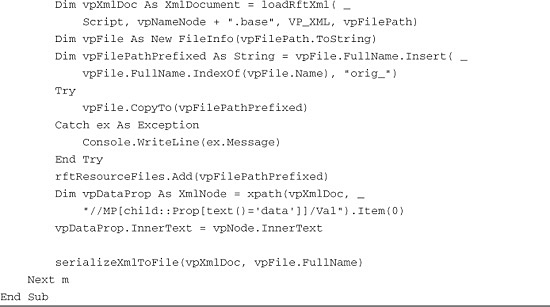
In these snippets, modifyMapValues() takes references to the three XML DOMs (the custom internationalized XML, the Script Definition XML, and the Object Map XML); the code then constructs the target file location for the modified Object Map XML (see [Listing 16.6](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex06) for how the file locations are captured), and then calls serializeXmlToFile().

**Modifying Language-Specific Verification Points**

While modifying Rational Functional Tester Verification Point XMLs for Unicode data uses many of the same techniques that modifying an Object Map XML uses (see the “[Modifying a Language-Specific Object Map](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16sec1lev4)” section), there is one major difference. A Rational Functional Tester script has a one-to-one relationship with its Object Map, but it can have a one-to-many relationship with Verification Point XMLs. This is because each Verification Point you create in a script persists its data in a separate XML file. Because of this design, the method that is built for dealing with Unicode Verification Point data performs all its own XML DOM and file handling internally, unlike the modifyMapValues() method (refer to [Listing 16.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex12)). Other than that, the same general approach is used: The original files are copied under a new filename (using a prefix of “orig\_” to create the new filename from the old), and the rftResourceFiles ArrayList is used to hold onto the new filenames. The Verification Point XML is searched for <MP> tags with a child <Prop> tag of text value ‘data’ to identify the nodes where Unicode data must be written. The <Val> sibling of the <Prop> tag actually holds the baseline data for the Verification Pont and therefore, it is the text value of the <Val> tag that is changed with each subsequent set of Unicode character data. [Listing 16.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex13) shows the modifyVpValues() method.

**Listing 16.13** A method to modify Verification Point baseline values





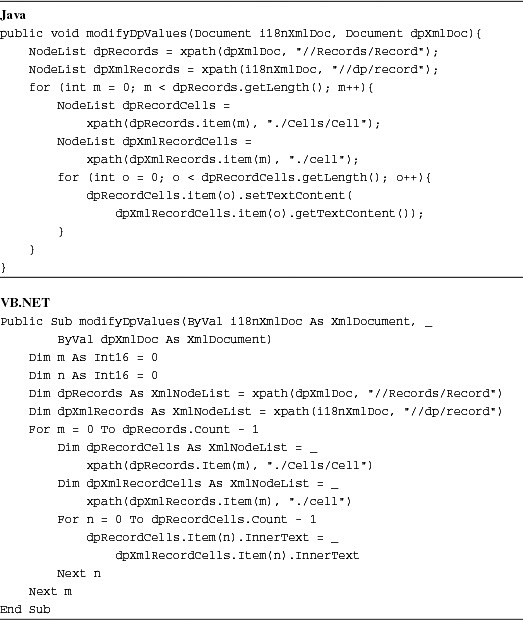
As noted previously, because modifyVpValues() handles loading the Verification Point XML into the DOM internally, and file handling for the Verification Point XML files internally (which includes copying the original files and saving the Unicode-modified XML to disk), the method call itself is just a single line of code (with the trailing semicolon in Java only, of course):

modifyVpValues(i18nXmlDoc, scripts[n], rftResourceFiles);

**Modifying Language-Specific Datapools**

Modification of Datapool XMLs follows much the same pattern as modification of Object Map XMLs. In the Datapool XML, data is stored in <cell> tags, and cell location metadata (such as the row and column values of each specific cell) are not explicitly written in the Datapool XML. Instead, row and column relationships are implicit; they emerge from the order of the tags in the XML. Each <Record> tag represents a row of Datapool data, and the <cell> descendants of the <Record> tags find their positions simply from their order. The tag layout used in the custom internationalized XML is much the same, so it is straightforward to correlate source data from the custom XML with the target in the Datapool XML. [Listing 16.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex14) shows the modifyDpValues() methods, which implement the copying of different Unicode data sets into the Datapool XML.

**Listing 16.14** A method to modify Datapool values



As with the modifyMapValues() method (refer to [Listing 16.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex12)), file handling is outside the method body, so the calling code for modifyDpValues() is almost identical, with the serialization of the modified Datapool XML external to the method. In Java:

modifyDpValues(i18nXmlDoc, dpXmlDoc);  
File dpFileI18n = new File(dpFilePath.toString());  
serializeXmlToFile(dpXmlDoc, dpFileI18n);

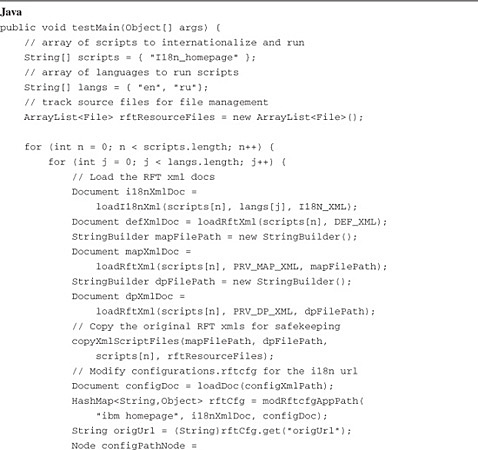
In VB.NET:

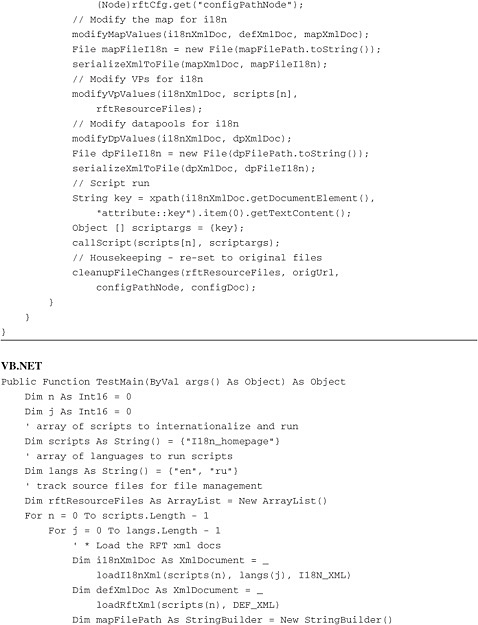
Image

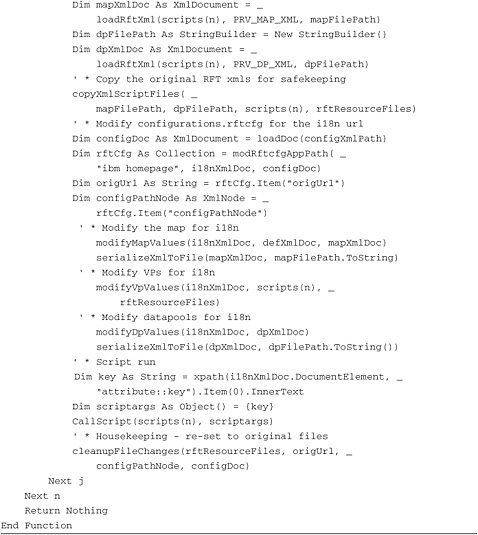
**A Fully Internationalized Driver Script**

If you take all of the development in the previous sections and combine it with the driver framework (refer to [Listing 16.11](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex11)), you get the full internationalized driver script. This is shown in [Listing 16.15](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex15).

**Listing 16.15** The complete internationalized driver script







**A Target Script for the Internationalized Driver Script**

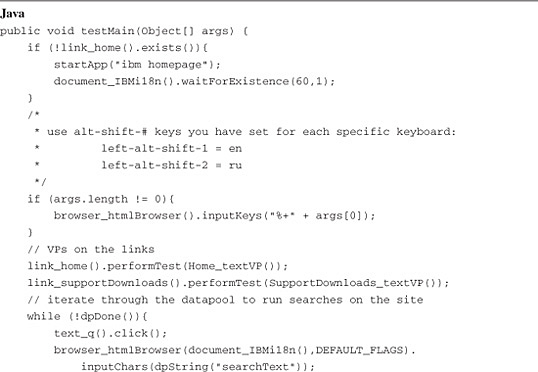
A target script for the internationalized driver script ([Listing 16.15](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex15)) looks like a regular Rational Functional Tester script. The driver script needs to know its name, of course, so that it can make its call to callScript(), but that doesn’t affect the actual target. Because all the Unicode handling is performed on the underlying script XML files before the target script is called, the target script never has to change in terms of its code, its TestObject names, or its Verification Points or Datapools. This example script is called I18n\_Homepage.

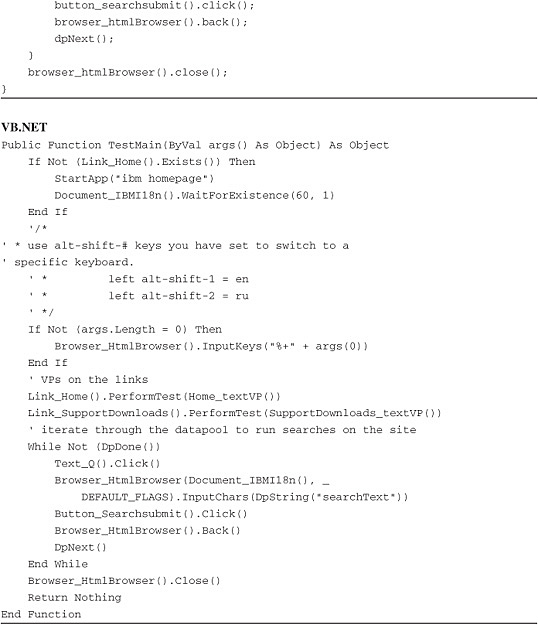
The one area where a target script departs from the typical script is in the area of handling the hotkeys for the different language keyboards. Normally, a script doesn’t worry about keyboard handling; it uses the default keyboard on the system. However, the internationalized test setup requires a small amount of additional handling to support testing with multiple Unicode character sets. As noted previously, when you set up language support on Windows, you should associate each keyboard with its own unique hotkey keystroke sequence, so that each sequence can be used to attach any keyboard desired to your application window. [Listing 16.9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex09) introduced the section of the driver that calls the target internationalized script, and a single argument is passed to the script—the unique keystroke (a number) to attach the required keyboard to the target application window. This argument must be handled in your target script, so that the correct keyboard is attached to the application window for the specific language iteration through the script. For most windows, it is simply a matter of a single line of code to type the keystroke sequence that assigns a keyboard to the window; in this example, the line is:

browser\_htmlBrowser().inputKeys("%+" + args[0]);

This call to inputKeys() types the Left-Alt-Shift-num sequence followed by the numeric script argument. [Listing 16.16](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch16.html#ch16ex16) shows the full target script.

**Listing 16.16** A target script for the internationalized driver script—I18n\_Homepage





**The Real World**

The i18n scripting example illustrated in this chapter makes several simplifying assumptions. First, it assumes that the object hierarchy is identical for all target web pages regardless of language. Frequently, this is not the case for internationalized web applications, and it often does not hold for i18n thick client applications. This means that, most likely, your implementation has to use branching logic based on the language in use; additionally, extra attention might have to be directed toward Object Map maintenance. Second, the example assumes that the transaction flow is not dependent on language, but often, this is not the case either; due to differences in culture and business practice that accompany language differences, application flows often change with language. The approach shown in this chapter (using a single script for each transaction regardless of language) can be adapted by building branching logic to accommodate different transaction paths based on the language in use. Finally, other aspects of applications can change with differing language implementations, such as the graphics that the application displays. Language-based image verifications can be handled in a manner similar to the techniques shown in this chapter, because image files are stored in the resources directory like other script assets (as portable network graphics or .png files), and are simply referred to by filename in the Verification Point XML.

**Summary**

The approach to designing an automated internationalized test shown in this chapter is, of course, not the only way to set this type of project up. The goal of this chapter is to make the different issues that you have to deal with in designing and building an implementation clearer and to offer one set of solutions to those issues. Your own solution might differ significantly, but hopefully, this chapter has shortened your learning curve for building an internationalized test suite.

**Further Information**

For another approach to this problem, see [www.ibm.com/developerworks/rational/library/07/0925\_mirchandani-ujjwal/](http://www.ibm.com/developerworks/rational/library/07/0925_mirchandani-ujjwal/).

[Prev](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch15.html)

[Chapter 15. Using Rational Functional Tester in a Linux Environment](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch15.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app01.html)

[Appendix A. Advanced Logging Techniques](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/app01.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/ch16.html)