ACTION-DRIVEN AUTOMATION TEST FRAMEWORK FOR GRAPHICAL USER INTERFACE (GUI) SOFTWARE TESTING

Li Feng
Sybase Software (China)
LongDong Ave. 3000#
Shanghai, 201203
(8621)68799918
Li.Feng@sybase.com

Sheng Zhuang
Sybase Software (China)
LongDong Ave. 3000#
Shanghai, 201203
(8621)68799918
Sheng.Zhuang@sybase.com

Abstract - In this paper we describe the design and implementation of an action-driven automation test framework especially for GUI software testing. The idea of action-driven automation test framework comes from the core concept of "Quality Assurance (QA)". Better quality can be ensured by increasing the coverage of test cases on the software but the process of creating large number of test cases has to be optimized. With this goal the framework was designed to primarily increase the efficiency and flexibility in composing test cases and simplify the process of learning the test cases. This paper describes the background, features, and implementation details of the framework.

INTRODUCTION

Before the automation test tooling was invented, the QA engineers had to do the test job manually especially for testing the GUI software. With the increasing versions, the QA engineers had to repeat what they did in the last testing cycle. The working pattern not only exhausted the engineers but also possibly left the regression of the software uncovered, which could lead to serious consequences. This problem was partially resolved with the generation of "test automation" concept. But it "has not met with the level of success that it could". [2]

Automation test originated from simply record and playback which makes QA engineers repeat the work in each testing cycle. Data-driven automation testing [1] is useful to cover those cases sharing the similar logic flow and different input data. That means you hard code the "logic flows" into the test scripts as the trunks and the trunks accept different test data input as the branches, so that different test cases are finally composed. Nowadays, the number of these "logic flows" is booming with the growth of software complexity. Therefore, you have to write thousands of scripts and maintain them in each cycle. So, extracting the logic completely out from the scripts is the main goal of our action-driven automation test framework. Here the "action" could simply be the click on a button, or it could be filling all blanks and submitting a wizard.

Based on this action-driven automation test framework idea, the overall testing work is separated into two parts: development of the framework and implementation of the test cases for certain applications. Accordingly, there are two roles while accomplishing the testing work: framework component designers and software testers. And our action-driven testing involves the creation of modular, reusable test components [3] that are built by component designers and then assembled into test scripts by software testers.

Main Features

From the above background introduction of our proposed action-driven automation test framework, the following features can be highlighted and specified as:

1. Highly reusable code. Each test component is a reusable unit which can be used to compose an executable test case. Once a component is done, it's available to all the test cases.
2. Easy maintenance. With the upgrade of the software under test, the framework can

minimize the change to the existing test components and test cases.

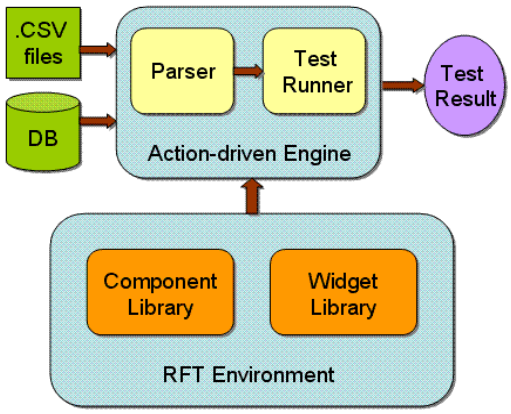
3. Separated automation test development roles. An action-driven test case is script-independent. That means the automation software testers again can be separated into two parts: component developers and test case composers. Component developers are responsible to create and maintain the components. Test case composers are responsible to create and maintain the test cases. As long as the testers are familiar with the software under test, they will be able to make test cases that can be interpreted and executed under the framework.

In the following sections, detailed implementation mechanism and the points which aim to solve the problems of old frameworks will be demonstrated.

ARCHITECTURE

This action-driven automation test framework based on Rational Functional Tester (RFT) is initially designed for testing Sybase WorkSpace, however, it can be easily adjusted and used to test any other java applications. Figure-1 demonstrates the framework model.

Figure 1- Action-driven Framework Model



Let's go into the details of the three main concepts shown in the diagram.

Action-driven Engine

Action-driven engine is used to parse the test case from a variety of sources such as CSV file and database, generate the test case in memory, and then run the test case step by step. At the meantime, it logs the verification results and logs the test case results in the end.

Composition of Test Case

We could think the test cases as the gas for the engine. So how to apply qualified and sufficient gas to the engine efficiently is a big solution we have to figure out. For the projects which adopt the action-driven framework, the test case composers firstly have to create .CSV files as test cases. Then practically, after the component developer finish with the component scripting the test case composer can call the corresponding components and specify the action on the component with the key-value pairs. For example, in one case, one step is the user right-click a folder on the structure tree, then click the New on the menu, and then click "Mobile Application Suite" submenu. In the .CSV file, we define that step as: "component.common.menu.MobileDevelopmentNodeContextMenu,selectItem,path=New->{$MOB_APP_SUI}". (path=New->{$MOB_APP_SUI}is the so-called key-value pairs, $MOB_APP_SUI is defined in a global variables definition file.)

An entire test case is a series of such steps, the test case composers' job is to put the steps together to create a case. And then create enough cases to cover the features under testing.

Plus in order to simplify the test case composing work, the framework also provides a small tooling for the user to easily find the components and select the actions (like "selectItem" in the above example).

After the "gas" is provided, engine can be launched.

Here we just show the structure of the engine and briefly introduce each important class's function one by one. Let's start with the whole diagram, please refer to figure 2.

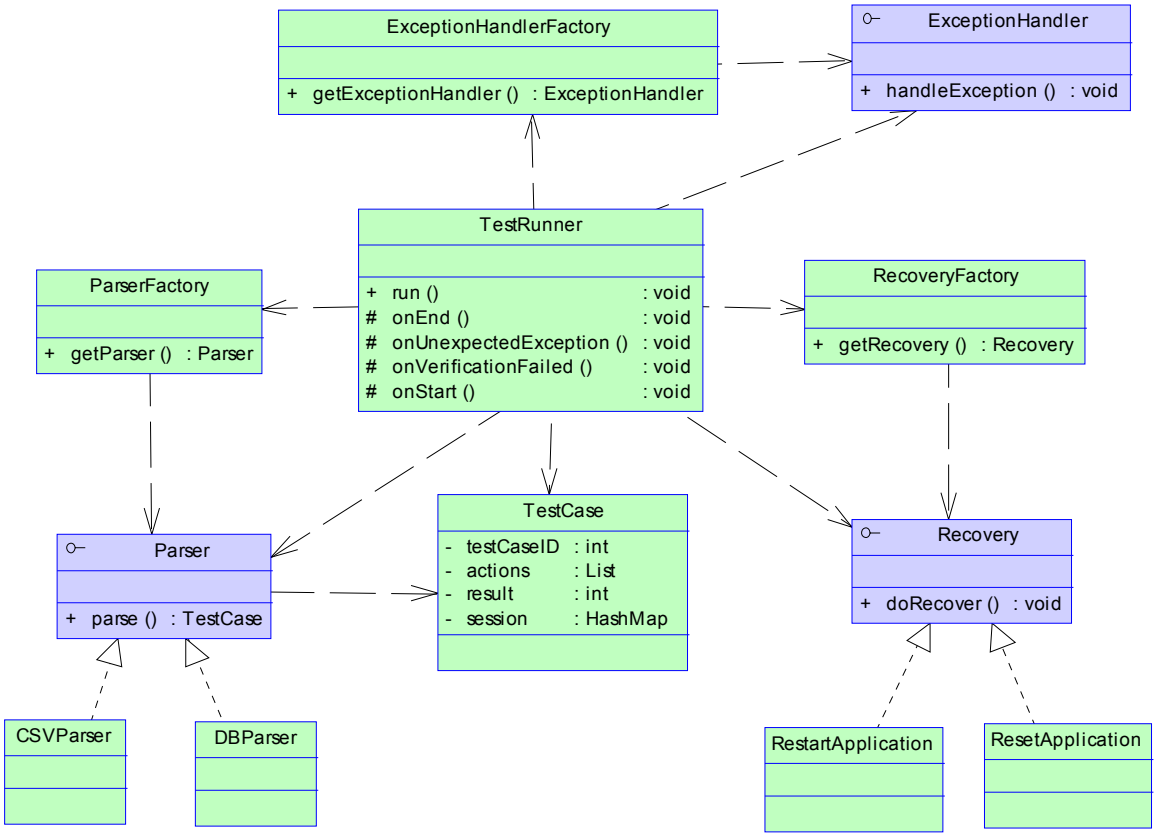


Figure 2 – Action-driven engine class relation diagram

Parser: The Parser class' function is parsing the TestCase files from different sources including CSV file and database table.

TestCase: The TestCase class represents the test case which is the result of the Parser and sent to TestRunner. Physically it could be a CSV file or some records in the database.

TestRunner: The TestRunner class' function is running a test case as well as manages the full cycle of the execution.

ReSetEnv Action: ReSetEnv action is invoked by TestRunner in the end of the execution of each test script. By doing this, the execution of test script is independent from each other and even though one test case fails, the rest can be executed without impact.

ComponentHelper: ComponentHelper is a "super helper class" which provides basic support for Component class development. With the ComponentHelper class, you can create various widget classes easily, create verification methods easily and access other frequently used functions easily.

Exceptions Handler: The exceptions can be handled either by the default handler provided by the framework or by the custom handlers which can be created specially by the user. By this way, it can be ensured that the rest cases won't be affected by the failure of the previous cases.

Component Library

The component library is a set of "component" classes residing in the component folder of the automation project. Each component class encapsulates a logical division of the application, which means it is logically a group of GUI elements.

This can be truly called Object Oriented (OO) approach to write GUI automation. RFT enables testers to simply record a set of GUI interactions and play them back against the application under test. Although "record-playback" features of RFT can be used to create suites of test cases quickly, the limitations are also obvious.

1. First of all, most of the tasks of the testers are recording the scripts time and time again, and when the application changes, the testers have to

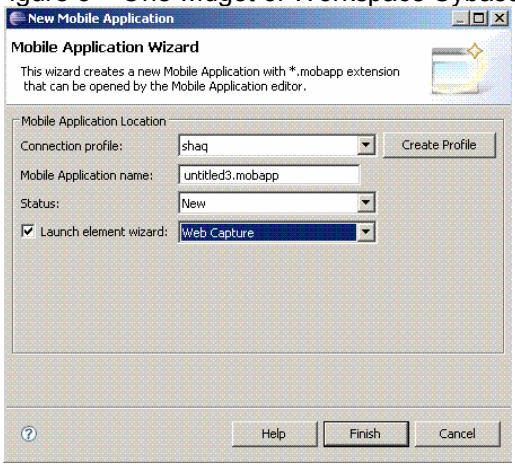
record them once again, that makes the job tedious and inefficient.

2. Furthermore, the "record-playback" approach leads to extensive test cases maintenance because of the low code reusability.

To solve the above problems, we design a common architecture to build the test cases which are called Component Library.

Each component is a division of GUI and contains a set of widgets. Here, we address an action as the actions from one or more widgets. For example:

Figure 3 – One widget of Workspace Sybase



The figure represents an eclipse wizard named "MobileAppCreationWizard", it contains 3 combo boxes, 1 text field, 1 check box and 4 buttons. So the basic action methods are as following (according to the naming rules of our project):

1. selectConnProfile(HashMap)
2. clickCreateProfile()
3. setMobAppName(HashMap)
4. selectStatus(HashMap)
5. checkLaunchElementWizard(HashMap)
6. selectElementType(HashMap)
7. clickHelp()
8. clickFinish()
9. clickCancel()

Note that each action involves test input, as selecting a connection profile and then status. The corresponding action method accepts a parameter typed "HashMap", from which the action will get the test input by key.
Let's see the code fragment:

```
//...
public void selectStatus(HashMap
params) {
	String status = (String)
params.get("status");
	WidgetActionUtility.selectTextI
nCombo(statusCombo(), status, true);
}
//...
```

"WidgetActionUtility" is a static utility methods collection that encapsulates common functions to manipulate GUI widget, like text field, button, combo box, table, tree and so on. If we want to use this action method in the test case, it will look like as following (in the test case):

```
# .....
MobileAppCreationWizard, selectStatus,
status=New
# .....
```

The key-value pairs as test input will be parsed into HashMap, which is the parameter of the action method translated from the first two columns through Java reflection.

Once a component and its actions are done, it is available to all the test cases. When the GUI of the software under test changes, we just need to change the implementation of the actions of the components, without modifying the test cases.

Physically the component can be the area as: view, editor, wizard, dialog, menu and preference. A component class consists of the following two parts: (1) A private object map that contains a small number of related GUI elements; (2) Action methods that execute commonly-traveled paths through the GUI.

Let's just take the above case for example to see how it works. Get back to Figure 3.
Given the scenario that we are going to do a series of actions:

1. Select the connection profile as "melo"; b)set the mobile application name as "stockQuote_A";
2. Select "Approved" as status;
3. Check the "Launch element wizard";
4. Select "Web Capture" as type;
5. Click Finish to submit the wizard.

So this test case will be:

==========
MobileAppCreationWizard, selectConnProfile, connProfile=melo
MobileAppCreationWizard, setMobAppName, appName=stockQuote_A
MobileAppCreationWizard, selectStatus, status=Approved
MobileAppCreationWizard, checkLaunchElementWizard, checkOrNot=true
MobileAppCreationWizard, selectElementType, elementType=Web Capture
MobileAppCreationWizard, clickFinish
==========

Meanwhile, we could also create a "big" action that calls these actions together as:

```
Public void createMobApp (HashMap
params) {
  this.selectConnProfile (params);
  this.setMobAppName (params);
  this.selectStatus (params);
  this.checkLaunchElementWizard
(params);
  this.selectElementType (params);
  this.clickFinish();
}
```

Then the test case will be:
==========
MobileAppCreationWizard, createMobApp, connProfile=melo, appName=stockQuote_A, status=Approved, checkOrNot=true, elementType=Web Capture
==========

Big actions can help to simplify the test cases which contain a lot of execution steps. That's what we called flexibility in the introduction section. The test case composers can compose the test cases neatly by using the small action or big action as his/her desire based on efficiency improvement or habit.

Widget Library

Widget lib is used to simplify GUI widget manipulations. RFT APIs for TestObject focus on the behaviors of the widgets, and the internal status of the widgets are encapsulated in ITestData. It's beautiful meanwhile it will also make users write more codes to do some more complex operations on the widgets.

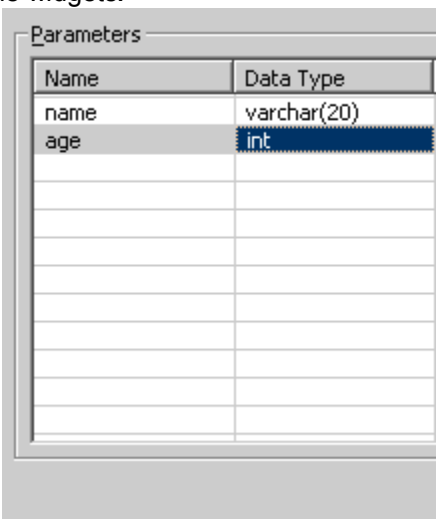


Figure 4 – Screen shot of one widget

If we use RFT API to set the "Data Type" of "name", the code will be:

```
Table().click(atCell(atRow(atIndex(0)
), atColumn("Data Type")));
TableDataTableCursor().click();
TableComposite().inputChars("varchar(
20)");
```

If we use widget lib, it will look like:

```
WidgetFactory.buildTable(Table()).set
Cell(0,"DataType","varchar(20)");
```

Wrapper classes encapsulate calls of the Rational Functional Tester API to perform the common operations on controls. The wrapper classes can simplify the control manipulation and enhance the code readability.

The Rational GUI automation tool has a powerful recorder feature that records a user's activities and automatically generates code that simulates these activities. And these code can be played back immediately without further revision to perform simple simulation testing. However, large and complex applications, such as Sybase Workspace, often require more sophisticated testing than unmodified record and playback can provide. The better way is using Rational Functional Tester to manually write code for the simulation testing.

Wrapping complex RFT API code into methods in widget classes has obvious advantages:

1) It not only provides common operations on controls (Test Objects), but also provides strong support for custom verification;
2) Readability has been greatly increased by reducing the code size to half;
3) The wrapper library is independent from the software under test which means it can be applied on any programs built with Java;
4) (4) Last but not the least, the test script built with wrapper library can be executed under any screen resolution.

SUMMARY

In this section, the advantages of the action-driven automation test framework will be shown by comparing it separately with the data-driven framework and the keyword-driven framework [2].

Data-driven automation test framework was brought out to improve the automation test which merely based on record and playback. It separates data from function, so that allows an automated test script to be written for a business function, using data-files to provide both the input and the expected-results verification [1]. As mentioned before, the difficulty of test script maintenance makes us motivated to design and implement this proposed action-driven framework.

Keyword-driven framework is another model which aimed for solving the problems of data-driven framework. It is defined by the "Core data driven engine", the "Component function" and the "Support libraries". While the "Support libraries" provide generic routines useful even outside the context of a keyword-driven framework, the core engine and "Component functions" are highly dependent on the existence of all three elements [2].

Compared with keyword-driven automation test framework, action-driven automation framework offers a more flexible and easier mechanism to form the test cases. The main difference between the two frameworks is the granularity of "action". In keyword-driven framework, every action is a basic action which can be composed to a "step" which actually happens in the test flow. In action-driven framework, user can customize the granularity of such "action". So to some extent, the actions of keyword-driven are just the smallest actions of action-driven framework. The flexibility of the granularity of the action can result in such advantages for the whole framework as: simplification of the test case structure, the extensibility of the test data by using the key-value pairs, etc. Consequently, maintenance of the framework turns relatively simpler.

ACKNOWLEDGEMENT

Special thanks go to the members who involved in developing this framework. They are Jason Fu, Jun Li, Charles Huang and Subrata Nandi who have provided meaningful suggestions and ideas which did improve the framework gradually. And thanks to Himagiri Mukkamala who gave us constructive advice on paper composing. We are one team to take the ideas into practice and benefit the automation test works.

REFERENCES

[1] Zambelich, K. *Totally Data-Driven Automated Testing* 1998 http://www.sqa-test.com/w_paper1.html

[2] Carl J. Nagle *Test Automation Frameworks* 2002 http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm

[3] Jerry Zeyu Gao, H.-S. Jacob Tsao and Ye Wu *Testing and Quality Assurance for Component-Based Software* Artech House Publishers 2006

[4] Nagle, C. *Data Driven Test Automation: For Rational Robot V2000* 1999-2000 DDE Doc Index

[5] Mosley, D. & Posey, B. *Just Enough Software Test Automation* New Jersey: Prentice Hall PTR, 2002.