Conquest: Interface for Test Automation Design

Anand Gopalakrishnan
Florida Institute Of Technology
Melbourne, Florida
agopalkrishn2010@my.fit.edu

Dr. Keith Gallagher
Florida Institute Of Technology
Melbourne, Florida
kgallagher@fit.edu

***Abstract*— Test Automation engineers often need to use multiple automation test tools and are required to extend and maintain automation test scripts across these multiple tools. For the test automation engineer the problems are the usual: information overload and maintenance in multiple environments. In such an overwhelming circumstance, the process of maintaining the automation test scripts from multiple tools and mapping it to the business criteria can be lost. In order to improve test automation design, we propose an interface to be used by test automation engineers for (Web) application testing. The interface provides a single workspace for incorporating multiple open source testing tools and frameworks for system/integration testing of (web) based applications. The goal of the interface is to simplify the process of mapping tests to business criteria without the coding coming in the way of the process, and thus increase the efficiency and flexibility in maintaining the test scripts. This paper describes the background, features and implementation details of the interface.**

Keywords-Test Automation Design, Conquest, automation interface, test design.

I. INTRODUCTION

Terry the Test automation engineer has been newly recruited to work on a project. The company has never had automation testing done before. The manager got excited with the story sold by an off the shelf automation testing tool in a workshop and decided to implement automation testing in his project. The talk given by the tool vendor set an expectation of turn-key simplicity. The manager is under the assumption that in automation, the tester has to only record a sequence of actions and then play it without human intervention, to get a set of nicely formatted report about the test status. Terry is asked to go ahead and implement the record and playback tool. Terry starts implementing the tool in a record/playback manner and the manager asks the entire test team to replace their manual testing with execution of the automation test scripts.

Marty, a manual software tester finds that the recorded automation scripts are not testing for all the business scenarios and reports the problem to Terry. Terry, on further interaction with Marty, understands that the logic flow of the application is incorrect; the tests only concentrated on the user interface of the application. Marty also points out many other business scenarios related to performance, database tests, multiple browser compatibility which are difficult for the manual testers to carry out and automation could be a valuable aid. Terry finds out that the tool cannot support the different scenarios and starts investigating and installs other tools. Terry then starts maintaining separate workspaces for the multiple tools. Terry also starts understanding the business logic and the requirements from Marty and starts maintaining them, leading to information overload and maintenance problems. Use of multiple tools in different workspaces leads to script duplication issues. A separate process of documenting the business logic flow compounds the maintenance of the scripts. Every time Terry finds out a new tool that can help in test automation , the entire test suite must be modified. Terry now spends most of the time setting up and maintaining the test automation environment and less time concentrating on the application under test. As a result, Terry runs out of time to cover all the important business criteria and the product ships with issues, leading to irate customers.

The test automation engineer's problems:

- Lack of a clear mapping from the business logic to the test scripts.
- Technology and tool environment focus, rather than testing focus.
- Information and maintenance overload.

II. BACKGROUND

"The test framework is like application architecture" [1]. Good test automation architecture should be able to parameterize data, log test status, report error and setup the environment for test. "Test automation is Software development. Like all software under development, it makes a lot of sense to figure out requirements and create something that meets them" [2]. The growth of software complexity and the different test patterns involved to cover all business logic flows led to thousands of test scripts written across multiple testing tools.

For instance, record and playback tools which make use of User Interface elements for constructing test scripts have been available since 1990s. However, frequent UI changes render the test scripts non-usable. These tools have noted short-comings of maintenance, as they heavily rely on the User Interface. The need to maintain the test scripts led to the development of different test automation frameworks.

One test automation framework architectural style is called *data driven.* In data driven automation, the test data is held in

separate files which are read by the automated test code and then used as an input for the software under test [3]. Data driven testing made it possible to run the same set of test for different data by parameterization. The approach made it possible to modify data without touching the test scripts. However, data driven approach proved to be ineffective in handling procedural or functional changes in test scripts.

Keyword driven framework came in to existence to address procedural and functional changes. The functionality of the application under test is documented in a table as well as the step-by-step instruction for each test. The action that needs to be performed as a part of testing is passed along with the data. Modularized methods are developed in corresponding to each business action. The methods can either be generalized at a level for all web based elements; or it could have a layered approach, with one layer having actions customized for an application under test.

A more effective version of keyword-driven frameworks is to externalize the business logic flow with combinations of data required to carry out the testing. This style is termed as a *hybrid* framework.

Test patterns that are used to address the logic flows may include web functional testing, web database testing, web performance testing and browser compatibility testing.

Based on a layered approach to test automation, the overall testing work is divided in to two parts: (a) a framework for incorporating new automation tools to meet the business logic and (b) writing test scripts to meet the logic flow for testing the application under test. Accordingly, we have two stakeholders in our testing effort: (a) Test automation engineers who design and extend the framework and (b) software testers who analyze the business logic for the application under test. Our approach is to create an interface to aid the test automation engineers in maintaining a single workspace for the multiple test automation tools and also to make it easy for them to map the business logic flow from the software testers to their test scripts for easy traceability.

CONTRIBUTIONS

From the above background introduction, the following main features in our interface can be highlighted and specified :

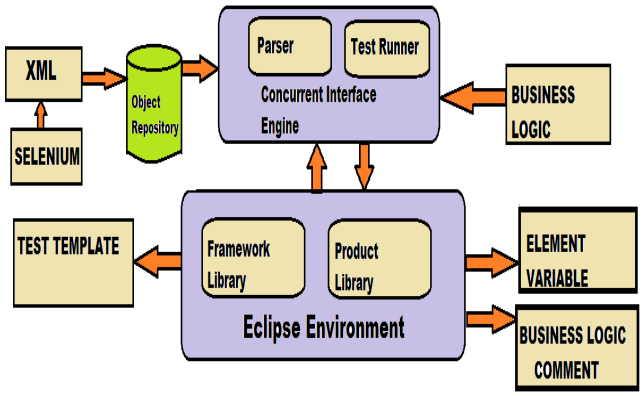
1. A direct mapping of business logic to code. The interface will make it possible to provide a Domain Specific Language which will link the business logic from the software testers in the form of comments in the code written by the test automation engineers.
2. Reusable code. A single interface for multiple tools will make it possible to share code across tools, if the tools are using the same underlying programming language for developing the test scripts.
3. Easy maintenance. Having all the different test automation tools under a single workspace would avoid duplication of test scripts. Single interface will make it possible to add and remove tools without affecting the overall architecture and implementation of the framework.

In the following sections, detailed implementation mechanism and design of the interface will be demonstrated.

III. INTERFACE ARCHITECTURE

The interface is built on top of the well known Integrated development environment (IDE) Eclipse for Java and an open source web application testing system, Selenium. Figure-1 shows the interface model.

Figure-1 Test Automation design model



A. Eclipse Environment

The interface is provided as an EclipsePortable Java project. The user only needs the latest Java Environment (JDK) setup on his machine to start using the interface. Modified version of EclipsePortable is provided as a part of the interface.

a. Framework Library:

Framework library contains all the classes which are for the framework and are product/project independent. The framework interface has functions for the following:

a.1. Initialize System:

This function deals with initializing the interface system. This could mean different functionalities for different testing tools. The function is responsible for initializing and starting the specific automation test tool and making it available to the interface. By default it has Selenium setup and initialized to be used by the interface. Any new tool can be added to it and made available for the interface. The interface engine described in the next section explains how to add a new tool. Extracting the tool support in to a separate class makes it possible to add and remove tools from the interface without affecting the test suite.

a.2. Test Logic:

This function deals with providing a wrapper for all the methods available in a test tool API. Providing a wrapper makes it easier to control the manipulation of API functions and enhance the code readability. Wrappers are provided by default for Selenium API methods.

Wrapping the complex Selenium API code in to methods has the following advantages:

- It provides common operation on controls and strong support for custom verification.
- Readability of the code is improved.
- Custom implementation and extended methods can be added to the API methods through wrapper without modifying the underlying API code.

a.3. Data Driver:

This function deals with providing support for data driven testing. Data driven testing is implemented using the @DataProvider annotation provided by TestNG. An object array is implemented to take Excel datasheets as input and use the data for parameterization.

a.4. Reporting:

This function deals with capturing the reports from different test runs across multiple test tools and integrating them to form a single seamless report. The function also deals with capturing screenshot on failure and linking it to corresponding step.

a.5. Database Check:

This function deals with checking the state of the database on providing user inputs and changing application outputs. The function also deals with updating and modifying the database to aid in testing disaster recovery scenarios.

a.6. Non Web:

This function deals with browser specific actions like testing the download of a file in Firefox or Internet Explorer etc. This function also provides support for cross browser testing. The cross browser testing is implemented using Selenium Grid, an open source java add on for Selenium, while the non-browser Windows API based testing is implemented using AutoIT, an open source windows testing tool.

b. Product Library:

Each Product will have its own library structure. The interface is designed to be a single workspace for multiple products.

b.1. Variable File Pattern:

Page Object pattern popularly advocated by the open source tool Selenium RC is a pattern that represents the screen of the web application as a series of objects [4]. Variable File Pattern introduced by the interface proposed in this paper is an addendum to the page object pattern. In Variable File pattern we divide the web application as a series of variables as opposed to objects. The variables are placed in their own class files which are then imported in the test file for the scenario. Every scenario will now have a test file containing the test scripts for the business logic and a test variable file containing all the web elements involved in the scenario as string variables. The pattern makes it possible to take advantage of Eclipse code completion feature to relate the web elements with the test cases. The interface engine automatically generates the link between the variable file and test file for a scenario, thus avoiding the tester to worry about mundane maintenance and environment setup tasks and concentrate more on the test scenarios.

The folder structure generated by the interface engine for each product can be explained as follows:

b.2. Product Variables:

Following the variable file pattern, this contains the class file with variables that are used throughout the product. Some examples of such variables can be given as URL of the product, Database driver details for the product etc.

b.3. Queries:

This folder contains all the database queries needed to carry out the database testing. The queries are provided as '.sql' files which are interpreted by the database check function of the framework library and used in the test scripts.

b.4. Test Cases:

This folder contains all the excel datasheets to be used by the data driver function for parameterization of test scripts using multiple data provided by the sheet (data driven testing).

b.5. Test Scripts:

This folder contains the test script files for different scenarios having the test logic flow implemented.

b.6. Test Variables:

Based on the Variable file pattern explained above, this folder contains the test variable files for different scenarios. The association between these files and their respective test script files are done by the interface engine.

B. Object Repository

The interface provides a custom format for Selenium IDE. Using this format while recording will generate an xml file with the page elements accessed while recording a scenario inside custom formatted xml tags. The record and playback tools are always used to record a scenario and then playback the scenario, but in our interface, the custom format provided modifies the record functionality of Selenium IDE. The Selenium IDE is used as an interpretation engine and not as a record/playback tool. Using Selenium IDE on a web page with the custom format will get the DOM element information for the objects in an xml format and not record the scenario. For example, in our scenario of testing the login to Gmail, the pages involved in our scenario are the Login page and the Landing Page. We need the elements in the Login page to perform actions and we need the elements in the Landing page

for verification purpose. We will start with using Selenium IDE with our custom format: "ATSFormat" and perform actions (click, type etc.) on all the elements (that we expect to be a part of the scenario) in the Login and Landing page. On completion of the record, we save the recording as an XML file under the Workspace folder provided by the interface. The recorded XML sheet for our scenario will have contents as shown below:

```
<page>
<field >
<Text>Email </Text>
</field>
<field >
<Text>Passwd  </Text>
</field>
<field >
<link>signIn </link></field>
</page>
```

As seen above, the XML sheet contains the DOM element identifiers for the web elements on the login page of Gmail. It also classifies the elements as <Text> element for text boxes and <link> element for links. The interface engine then parses the XML file to form a test variable file for the login page. The engine then adds an import of the test variable file to the test script file, thus associating the elements with the test script for the scenario. The test variable file output of the parser can be shown as follows:

```
=======================================
public class TestLoginVar extends Toolset {
public static String TestLoginOutput=
"TestLoginOutput.xls";
public static String TestLoginOutputDB=
"TestLoginOutputDB.csv";
public static String DBRoot =
"Result\\Gmail\\Data\\DBOutput\\";
public static String txtEmail= "Email";
public static String txtPasswd= "Passwd";
public static String lnsignIn= "signIn";
}
=======================================
```

As discussed in the variable file pattern, it is seen that the parser generates the test variable file with the web elements as static string variables. The reason it defined the elements as static is because the elements may be used across different scenarios. As seen in the output, the class in the variable extends the Toolset class. Toolset class contains the functions needed to initialize the system (as discussed in the Framework Library). The two static strings TestLoginOutput and TestLoginOutputDB are the report files created for functional test report and database test report.

C. Concurrent Interface Engine

The concurrency characteristics of the engine is in the idea that it can be modified and extended without affecting the implemented test suite execution. The tester and the test automation engineer can use the tool *concurrently*.

Concurrent interface engine is a HTML Application (HTA) built on top of the Eclipse environment and provided as a part of the interface. The engine has a built in xml parser to parse the object files coming from the object repository, scenario editor and a tool plug-in module that allows for seamless and easy integration of new tools in to the interface. The concurrency characteristics of the engine is in the idea that it allows to extend and modify the framework without affecting the implemented test suite execution. The interface engine also makes it possible to keep track of the business logic while writing the test scripts without the need of a separate process for the same.

a. Parser: The xml parser is built in VbScript which accepts a pre-formatted xml sheet from the object repository and converts it in to a java file containing the web elements of application - under - test as static variables. The parser also generates the java test script template.

b. Scenario Editor: The scenario editor is built in HTA using VbScript, which accepts workflow and business criteria and adds them as a separate method and comment to the method respectively.

The functionality of the engine can be explained in detail with the following scenario to be automated:

"Test the Login functionality of Gmail"

The test automation engineer will pair with a software test engineer and document the logic flow in a Domain Specific language using the Scenario editor. The logic flow for testing a successful login can be expressed in DSL as follows:

Product: Gmail

Scenario: TestLogin

Workflow name: PositiveLogin (this can be named anything that seems to agree with the application domain).

Criteria: Login to Gmail with the correct "username" and "password" and the user should be able to see the landing page. (This is a sample criteria, of course in a real test implementation we would be adding more details about the landing page. The details about the landing page can be specified in the same criteria or in a collection of criteria based on the application domain).

The main entities of the DSL can be explained as follows:

Workflow: Workflow is a name given to a sequence of actions that will be carried out in the test scenario.

Criteria: Criteria is the sequence of actions that represent the test logic flow.

The engine converts the workflow provided as a java method and the criteria as a comment for the method. Figure -2

shows the Scenario editor with the DSL entered for the above stated scenario.

Figure-2 Interface Engine Scenario Editor

As seen in the image, On entering the Workflow name and clicking on Create button will automatically add the workflow inside the <Method></Method> tags. On clicking the criteria button it will create the <Criteria></Criteria> tags for us for entering our test criteria in between the tags. After the criteria is added, the engine generates a '.java' test file and '.java' test variable file (the concept of test variable file is explained in detail in the next section). The test file has the following code generated for the workflow:

```
=======================================

// <Method>PositiveLogin</Method>
// <Criteria>Login to Gmail with the correct
"username" and "password" and the user should
be able to see the landing page</Criteria>
//
// <Parameter>: String username
// <Parameter>: String password
@Test
public static void PositiveLogin(){
   }
=======================================
```

As seen above, it creates a Java method having a name as that of the workflow and it adds the criteria as a comment for the method. The comment system makes it possible to map the business criteria with the test script eliminating the test automation engineer to go through a separate process for the same. Also, the engine identifies the parameters which has been specified in double quotes and makes a suggestion to the test automation engineer to consider them for parameterization.

The @Test annotation added by the engine is for the integrated test harness: *TestNG*. The scenario entered in the scenario editor is also saved as a separate xml file. The existing scenarios can be opened and edited using the editor. The engine also generates the standard import statements and creates a new class for every scenario. The following code is generated by the engine in the test file before the workflow code shown above.

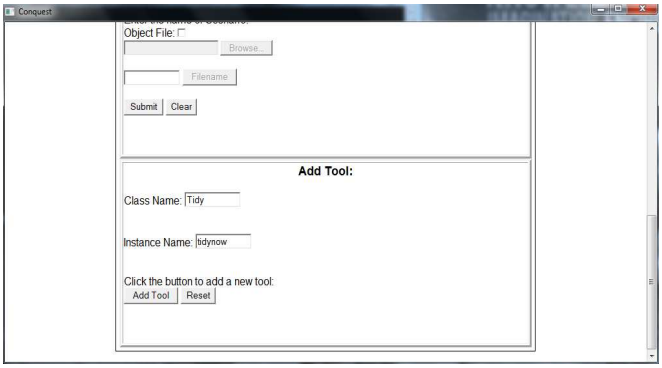
```
=========================================
package Gmail.TestScripts;
import org.testng.annotations.*;
import org.testng.annotations.Test;
import java.sql.*;
import java.util.Properties;
import jxl.*;
import java.io.*;
import Gmail.ProductVariables.*;
import Gmail.TestVariables.*;
import Gmail.*;
import Framework.*;

public class TestLogin extends Gmail {
@DataProvider(name = "DP1")
public Object[][] createDataImaging() throws
Exception{
Object[][]
retObjArr=driver.getExcel("Tests\\Gmail\\Test
Cases",
"DataProvider","Login");
 return(retObjArr);
}

@BeforeClass
public static void TestInit() throws
Exception{
driver.XLogger(TestLoginVar.TestLoginOutput,"
Gmail");
DBFunction.DBLogger(TestLoginVar.TestLoginOut
putDB,"Gmail");
}
=========================================
```

The @DataProvider and the @BeforeClass are TestNG annotations added for the TestNG harness in the interface. The job of the interface engine is to setup the Test Suite and generate template to start writing tests.

IV. ADD A NEW TOOL

Interface Engine provides an easier mechanism to add new java web testing tools (jar) to the interface. As discussed in the earlier section, all the tools used in the interface are extracted and maintained in a separate class file. Maintaining the tools in a separate file makes it possible to add new tools to the interface without affecting the test suite structure. The tester needs to add the jar file to the class path and then define a class and instance in the tool class file. Interface engine provides a simple user interface to carry out this task. Figure-3 shows the implementation of this feature:

Figure-3 Adding a Tool using Interface Engine



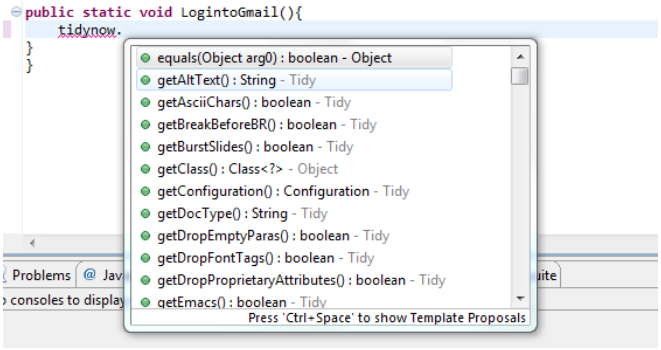
As seen in the image, the tester only needs to provide the main class name used by the tool, which he can obtain from the JavaDocs of the tools. He then needs to provide an instance name with which he wants the tool to be recognized in the interface. The engine will then open up the folder where he needs to place the jar file. Once the tester places the jar file, the engine will add a class instance for the tool in the tools class file and also automatically add it to the class path, making the tool available for the interface. The tester can then invoke all the API methods of the tool from the test scripts file by simply using the instance. An example of the tool adding process can be given as follows,

Consider the case of adding the tool Tidy (a HTML syntax checker tool) to the interface. The javadocs for the tool indicates that the main class for the tool is Tidy. Let us use the instance name as 'tidynow' (the instance name can be anything that will make it easier for the tester to identify the tool). Now the engine will open the library folder and ask the user to put the tidy jar file in the folder. On moving the tidy jar file inside the library folder, the engine will add the tool to the class path and generate the following piece of code in the tool class file:

```
==========================================
public static Tidy tidynow = new Tidy();
==========================================
```

So the tool is now available to be used from the test script files by using the instance name 'tidynow'. Figure-4 shows the usage of the tool in our test script file

Figure -4 Using the newly added tool in the test script file (the benefit of eclipse code completion is seen here)



As seen in the image, typing the instance name followed by a '.' inside our test script workflow method, shows all the methods provided by the Tidy API.

V. RELATIONSHIP WITH DATA-DRIVEN FRAMEWORK

Data-driven framework concentrated on externalizing the data from the test scripts to provide parameterization and easy maintenance of data.

Positives:

- Test data changes does not affect the test script
- Test data can be easily reviewed
- Test data can be used across different scripts

Negatives:

- Does not deal with frequent functional changes in the underlying script.

Interface:

- The interface treats data-driven framework as a module in its implementation and not as the only available framework. The interface supports data driven testing through TestNG data provider. The interface engine automatically places the below data provider object implementation for every new scenario:

```
=====================================
@DataProvider(name = "DP1")
public Object[][] createDataImaging()
throws Exception{
Object[][]
retObjArr=driver.getExcel("Tests\\Gmail
\\TestCases\\login.xls","DataProvider",
"Login");
 return(retObjArr);
      }
=====================================
```

The Data provider object is pointed to the Test Cases folder which is created for every project, as explained in the Product Library section. The combination of data for driving the test can then be passed through a spreadsheet placed in the Test Cases folder. In our example, the spreadsheet is "Login.xls". The data to be used for parameterization can be passed through the spreadsheet.

VI. RELATIONSHIP WITH KEYWORD-DRIVEN FRAMEWORK

Keyword-driven framework externalizes the actions from the test script in to separate library files and provides a layered approach to automation.

Positives:

- Readable tests closely related to business domain
- Enables easy test reviews

Negatives:

- Linking the library files with different test scripts.
- Test automation environment setup and maintenance taking up more time than the test design.

Interface:

- Automated template generation for linking library files with test scripts.
- Automated and easy test environment setup.

The interface engine automatically adds the import statements for the framework level class files every time a new scenario is created.

VII. REUSABLE SCRIPT AND EASY SCRIPT MAINTENANCE

A single workspace for multiple automation tools makes it possible to reuse common interface level scripts. Higher level scripts such as a random data generator for database tables can be placed in the framework library thus making it available for database testing, load testing and usability testing tools.

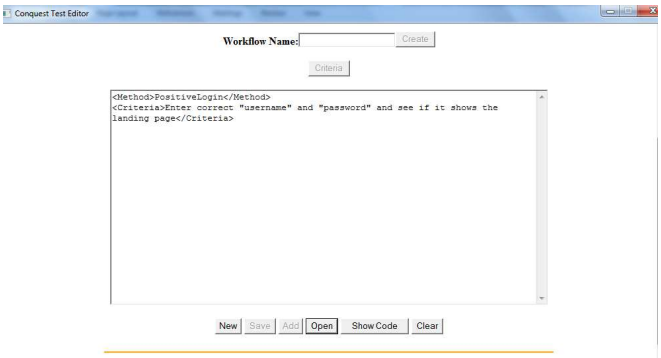
Single workspace also avoids duplication of test scripts. For instance, Consider a test scenario for adding a user to an application. Our testing might involve verifying the functionality of adding a user as well as load testing the application for performance check. In both the case, the actions that needs to be performed to add a user will remain the same.

The test scenario will branch out after the steps for adding the user are done. For functionality testing we would be verifying if the added user appears on the application as well as is added as a database record. For the performance testing we would be using a separate performance test tool to monitor the performance changes for multiple user adds.

Single workspace will avoid the repetition of test script for the common actions to be performed on the application under test for both the tools.

Interface engine maintains the relationship between the business rule and test script implemented. Interface engine makes it easy to locate the test script implemented for the business rule, thus making it easier for the test automation engineer to keep track of the business case covered by the test script.

Figure -5 Interface engine: Relationship between business rule and code



As seen in the figure, for every Business rule under a test scenario, we can look at the implemented code by clicking the "show code" button. "Show code" will launch the modified eclipseportable version with the respective test scenario class (java file) and the corresponding business workflow method.

VIII. SUMMARY

The paper proposed an interface design to extend and maintain automation scripts across multiple test automation tools. The paper also demonstrated how an automated template generation can ease the process of test design.

The interface proposed gives a design pattern to build a single platform approach for the different framework implementations across multiple tools with an efficient traceability process built in it.

IX. REFERENCES

[1] Hayes, Linda G., "Automated Testing Handbook", Software Testing Institute, 2nd Edition, March 2004.

[2] Kaner, Cem , "Architectures of Test Automation", http://www.kaner.com/pdfs/testarch.pdf, August 2000 (November 1, 2011)

[3]Kent, John, "From Record/Playback to Frameworks", http://www.simplytesting.com/Downloads/Kent%20-%20From%20Rec-Playback%20To%20FrameworksV1.0.pdf, 2007 (November 1, 2011)

[4] Stewart, M. Simon, "Page Object Pattern", http://code.google.com/p/selenium/wiki/PageObjects, October 2011 (November 1, 2011)

[5] Selenium: http://seleniumhq.org/ (November 1, 2011)

[6] Selenium IDE: Adding Custom Format http://wiki.openqa.org/display/SIDE/Adding+Custom+Format (November 1, 2011)

[7] EclipsePortable: http://sourceforge.net/projects/eclipseportable/ (November 1, 2011)

[8] Jennitta Andrea, "Envisioning the Next Generation of Functional Testing Tools," *IEEE Software*, vol. 24, no. 3, pp. 58-66, May/June 2007, doi:10.1109/MS.2007.73