Automated User Interface Testing for Web Applications and TestComplete

Samer Al-Zain
Faculty of Information Technology, Birzeit University, Palestine, P.O Box 977
Telephone number, +970 (972)22973359.
szain@birzeit.edu

Derar Eleyan
Faculty of Information Technology, Birzeit University, Palestine, P.O Box 977
Telephone number, +970 (972)22973359.
deleyan@birzeit.edu

Joy Garfield
School of Technology, Wolverhampton University, Wolverhampton, WV1 1SB.
Telephone number, +441902 321464.
j.garfield@wlv.ac.uk

ABSTRACT

Automating software testing is an important and time-saving activity used by software testing teams working on rapid and large scale software projects. TestComplete is an example of a current widely used testing tool. However, its test recorder tool appears to have some weaknesses when using GUI (Graphical User Interface) test recording for dynamic web applications. After recording a GUI test using TestComplete recorder, it fails to run again later on because some of the onscreen objects cannot be recognized by TestComplete. Since TestComplete recorder tool generates tests in scripting languages, the test itself will be refined and modified to be robust and much more accurate. This paper presents an algorithm for writing robust and successful test scripts for TestComplete against dynamic web applications. It also presents a comparative study with the Web Performance Test tool provided by Microsoft Visual Studio.

Keywords

Automated Testing, TestComplete, Web Application, ASP.NET, Quality Assurance**,** Visual Studio

1. INTRODUCTION

Software is involved in every aspect of modern day life. Almost everything used today has software embedded to run it. In fact, software has the ability to connect, simplify, heal and entertain humankind. Global problems, such as killer diseases, climate change, overpopulation, worldwide financial meltdowns, alternative energy…and many more problems cannot be solved unless software is part of that solution [1].

Nowadays, in a typical workplace, everyone uses a computer and software applications. Entire organizations are powered by software systems, some of which are critical systems where software errors are not acceptable. Since software systems are developed by imperfect humans, failures and errors will always be present. Such software systems need to be developed in a way that reduces or eliminates defects and errors [2].

Software testers should take sufficient time to test software but time is a luxury that software testers do not have. Modern software development organizations dedicate special departments and teams to verify the quality of their software products [3]. These departments are known as QA (Quality Assurance) departments and have the responsibility of testing and improving the quality work of the organization. As useful software is complex to build, there is always a problem in building and developing quality complex software on time, as argued by [4]. Indeed many software projects fail to deliver software on time. Many of these projects, however, try to squeeze their development time by reducing the time for software testing. The result is a software product that is not well tested or verified because testing teams did not take enough time and recourses to test and verify the software product.

One of the difficulties with software testing is that customers want more functionality to be delivered faster and cheaper, while at the same time wanting software quality to meet and sometimes exceed their expectations. According to [5], more functionality means software will become larger and more complex. It also means that testers will run more test cases. Put simply, more software needs to be tested in less time and more often, by fewer people.

In modern software development processes, such as agile methods, software testing is not a separate phase that is carried out at the end of the project. It is integrated through the whole development process and starts at the early stages of a project. Every sprint adds new functionality to the overall system. Regression tests are key tests used by testers in such situations to make sure that new builds do not break previously tested software modules. However, doing manual system and regression tests is not practical and they are considered to be time consuming activities.

Having software to test software is called test automation. Graphical User Interface (GUI) test automation is an important part of software testing and provides software testers with early warning signs when parts of the system have changed or been broken. Time is saved because automated tests run faster than human tests, giving the ability to be run at night. This gives software testers time to write additional and creative test cases. User interface automation testing can also free software testers from routine or mundane tasks, which will increase as development moves on and new parts and software modules are built. Finally, automated tests provide safety nets through regression tests, which are executed whenever a new build is completed by the development team [6].

One of the premium automation testing tools, and most notable, is TestComplete, a product by SmartBear [7][8]. TestComplete can create, manage and run automated tests for any windows, web or rich client software. By using TestComplete, test engineers can perform several types of automated tests, such as functional Graphical User Interface (GUI) tests, regression tests, load and stress tests, unit tests and many more. Another reason for choosing TestComplete is that it provides testers with the ability to write test scripts from scratch using scripting language, such as Java Script. This ability enables testers to write complex and dynamic test scripts.

Among software systems, web applications are the dominant class. Web applications support a wide range of activities from e-commerce and medical to scientific activities. Recent reports and studies indicate that web applications are not as dependable as they should be [9]. For instance, one study shows that 29 out of 40 leading e-commerce web applications and 28 out of 41 government sites exhibited some type of functionality failure [10][11].

This paper will introduce the problem with TestComplete 8.5 recorder tool when recording and playing back an automated GUI test for dynamic web applications. After explaining the test environment, TestComplete will be used to record a test against a dynamic web application and show how it fails to play back again. After that, a methodology based on writing automated tests using TestComplete script language will be utilized to propose a robust solution. The solution itself will be tested against the same scenario and verified.

1.1 Problem Statement: Recorder Tool at TestComplete

One of the many features that TestComplete encompasses is its ability to easily create and run automated user interface tests, using the record-and-reply feature tool. According to Top Reasons to Try TestComplete [8], TestComplete gives the ability to review and enhance tests by providing test script views for those tests. So, every time a tester records a GUI test using a TestComplete recorder tool, TestComplete generates a test script in a test script language, such as JavaScript. This test generated test script can be enhanced and modified by the tester to improve test quality.

However, recording and running tests with the TestComplete recorder tool alone, appears to be weak when it comes to dynamic web pages. Most tests recorded for dynamic web pages, fail to run again later on, because the TestComplete engine cannot recognize some of the onscreen objects, such as links, buttons, text fields, etc. SmartBear acknowledges this problem, as argued in [12]. This problem has actually been present since TestComplete 7.5 and is still not solved in the current version of TestComplete 8.5, which is the version used in this paper.

TestComplete support presents more than one solution for this problem. These solutions can be found in the help section for TestComplete or at the online help portal. These solutions are based on enhancing or modifying the generated test scripts to make testing more robust. In fact, writing tests using test script in TestComplete provides wide access to APIs (Application Programming Interfaces) for TestComplete itself. This results in more robust and smarter tests.

These solutions provided by TestComplete support are presented as partial solutions, however, there is no clear and complete solution algorithm that developers can follow and implement. TestComplete help provides partial code in scripting languages to address finding onscreen objects but they do not provide complete code or algorithm/s that testers can use or follow easily.

One of the main causes of the problem (TestComplete recorder tool) is that numerous web applications are dynamic in nature, and some of its content controls (onscreen objects) have properties to change their values from one web page execution to another. TestComplete fails to recognize web page onscreen objects, such as buttons, links, text fields, etc., when they are recreated over and over again after recording a test.

This is due to the fact that, at the time of recording the test, TestComplete recognizes the web control through the values of a set of their attributes. Those attribute values are saved and used later to find page controls when replaying the test. If one attribute for onscreen object changes its value, TestComplete will not recognize it and the test will fail, as shown in later sections. When the test is re-played and the tested web page is recreated, TestComplete engine records different values for some of the attributes for onscreen objects [12].

Another cause of the problem is that, during software development, developers change the control hierarchy and page internal structure by modifying the tables' structure. When developers change the underlying tables (by adding and removing rows and columns), some of the attributes for onscreen objects change their value, because TestComplete engine depends on the page hierarchy to define some of the attributes' values. This causes TestComplete not to recognize the onscreen objects when the test is re-run.

TestComplete also has a problem when it comes to waiting for web pages to load. TestComplete should wait for the page to load completely and then start to access its onscreen objects. Apparently, this does not happen most times and, when some pages take more time to load, TestComplete may start accessing its onscreen controls even though they are not completely loaded. This will cause an error and the test run will fail. This paper will also address that problem.

1.2 Recording Tests for Dynamic Web Pages using TestComplete

In this part, the TestComplete recorder tool will be put under test against a dynamic web application to record test scenarios and highlight the problem occurrence. TestComplete recorder will record the test and later the same test will be played back by TestComplete to show the problem.

1.2.1 Experiment Design

The web application under test is sample dynamic web application software built using ASP.NET 3.5. The author will use TestComplete 8.5 to record and run GUI test automation. All pages of this application are created at run time. Some attributes of those page controls change, because the pages are created from XML files. When pages are recreated, TestComplete assigns different values to some of the attributes for onscreen objects.

The application consists of login page, main page, person search and detail pages. Upon successful login, the main page is loaded. Using left menu links, it is possible to navigate to the person search page. On this page there is a link to add a new person by opening the person details page in another browser instance.

The rationale behind this web application is that it is dynamic in terms of web page creation. These kinds of web applications are very common and very few are of static nature and because of this, onscreen objects (buttons, text boxes, lists, etc) attributes change. Since TestComplete relies on these attributes when finding onscreen objects, some tests fail when playing back recorded tests.

The tested application is published on the testing environment that consists of a separate workstation. Another workstation will host TestComplete and will serve as a testing workstation for recording and executing tests, and will be on the same LAN. Microsoft Internet Explorer 8 is used as the default internet browser.

1.2.2 Using TestComplete to Record a Test Scenario

In this section, the tester will use TestComplete to record and play the automated user interface test against the target web application, showing how TestComplete fails to play the test again.

The test scenario steps are as follows:

1. Initiate IE8.
2. Navigate to target web application login page.
3. Provide user name, password and click on login button.
4. Wait for main page to load.
5. Click on link "Person", which is located on left menu, to view person search page.
6. Click on "new" at person search page to view person detail page.
7. At person detail page, tester will save basic person record by entering required fields only.
8. Test scenario ends.

First, the tester created a new project using TestComplete. Using the TestComplete recorder tool, the tester recorded the above scenario. The recorded test was then played back. TestComplete performed well in steps 1 – 4, but failed in step 5 and all subsequent steps. This is because TestComplete could not find the onscreen object "Person" link, so the whole test failed.

2. METHODOLOGY

The proposed solution for this paper is based on writing test scripts and TestComplete APIs, not on the TestComplete recorder tool. As shown above, the TestComplete recorder tool generated test failed when run again by the tester. Using TestComplete, all recorded UI automated tests can result in test scripts, and the tester can choose from various language scripts such as Jscript, VB, Delphi, etc. Consequently, the software tester can write robust and smart test scripts, without relying on TestComplete test recorder. Through these scripts, it is possible to access the APIs which TestComplete provides, to write test scripts that can search for web controls, using a minimum set of attributes that do not change when the page is re-created. Not all web page control attributes change. However TestComplete recorder does not know this when recording the test. So the solution is that when writing test scripts, the tester will only focus on attributes that will not change, leaving out the ones that are likely to change.

The tester can discover the attributes that do not change their values by investigating the properties of onscreen objects, using TestComplete *Object Browser* and *Object Finder* tools. These tools can show the attributes' values for onscreen objects at any time. If the software developer gave an "id" for the onscreen object during development, TestComplete will identify it as attribute named *idStr*, using the object browser tool. If attribute *idStr* is present for that onscreen object, it can be considered to be enough for identifying that onscreen object, and there would be no need for other attributes. If *idStr* is not present, then the tester can look for other attributes that are more likely to retain their values from one run to another. Examples of these attributes are *innerText* and *ObjectType*.

According to [13], TestComplete can use several models to present the hierarchy of web page elements. These models define how elements of tested web pages are shown in the Object Browser panel and, more importantly, how they are addressed in test scripts. The models are DOM (Document Object Model), Tree and Tag. Tag model does not depend on element hierarchy, as Tree model does. According to TestComplete documentation, DOM is not recommended to be used when accessing web page elements of the same type, as this will slow down performance. In such cases (as in this paper) Tree or Tag models are recommended instead.

Neither of the two models solves the problem of identifying elements when their attributes value and/or hierarchy change. This leaves the main problem, which is, after the test is recorded and an attempt is made to run it again, TestComplete will not find the on-screen objects because some of their attribute values have changed due to the dynamic nature of the tested application.

3. SOLUTION ALGORITHM

The solution is based on writing test scripts instead of using the TestComplete recorder tool. Solution code (shown in the appendix), is based on using TestComplete test script APIs. All functions used have complete specification documented at TestComplete help online or in the help section at TestComplete itself.

Prerequisites: adding IE to tested application for TestComplete, and providing URL for startup page.

Solution steps (algorithm):

1. Initiate Internet Explorer. This happens only once at beginning of test scenario.
2. Obtain IE process.
3. Navigate to target page URL.
4. Make sure that the IE process waits for page to load completely.
5. Make sure that the target onscreen object is loaded inside web page before accessing them.
6. Find onscreen object using attributes that do not change from one run to another.
7. Access onscreen object by getting, setting, or performing click events on it.
8. If actions result in opening page in another window, search and wait for that page to load.

3.1 Solution as JavaScript Code

Steps 1, 2 and 3 are done through code listing 1:

```
IEProcess = TestedApps.Items(0).Run();
```

Listing 1

Step 4 is done through code listing 2:

```
TargetPage    =    IEProcess.WaitChild(PageName,
WaitTime)
```

Listing 2

PageName: the page URL required.

WaitTime: time to make TestComplete wait for web page to load in milliseconds.

After obtaining the web page, it is necessary to wait and make sure that its target onscreen object is loaded and ready. When searching for the onscreen object, certain attributes will be chosen that do not change from one page run to another. The first choice will be the *idStr* property, which represents that programmatic name given by the developer. If for any reason the idStr is not presented, the tester should look at the Object Browser tool and look for other attributes that do not change. Several attributes can be used to find onscreen objects. Step 5 can be achieved by code listing 3:

```
control   =   TargetPage.Find(“idStr”,   “*txtName”,
1000);
while(control.Exists               ==               false){
        Delay(100);
        control      =      TargetPage.Find(“idStr”,
“*txtName”, 1000);}
```

Listing 3

If the onscreen object is still not loaded, and an attempt is made to find it, TestComplete will make its Exists attribute to be false. So, the tester should wait 0.1 second every time before trying to find it again.

With reference to the *Person* link that TestComplete failed to find when applying the test in the previous section (see figure 3), the tester will search for it using different attributes. This is because the *Person link* does not have an *idStr* attribute. Candidate attributes and values are *ObjectType=Link* and *innerText=Person*. These two attributes will be used to search for the link.

Here two arrays will be used, one for attributes and another for attributes values. *Find()* method has an overloaded version that accepts arrays as well, so steps 6 and 7 are in code listing 4:

```
arrProps   =   [“ObjectType”,   “innerText”];
arrValues = [“Link”, “Person”];
personLink=TargetPage.Find(ConvertJScript
Array(arrProps),
ConvertJScriptArray(arrValues),1000);
personLink.Click();
```

Listing 4

The author also used the *Object Finder* and *Object Browser* tools provided by TestComplete to find and select attributes and their corresponding values.

A complete solution test script is provided in figure 5 in appendix B. After TestComplete executed the solution test script, which is written in Java Script, the execution was successful and the test script ended successfully.

3.2 Comparative Study with Web Performance Test Tool

Web performance tests (Web Tests) are available at Microsoft Visual Studio and works at the protocol layer by issuing HTTP requests. When a tester records a test scenario, the web test records a series of HTTP requests and later, when performing a play-back, the web test executes those HTTP requests in the same order they were recorded.

Web tests can be used to test the functionality of web applications as well as testing the application stress, which is also known as load testing. Web tests automatically handle other aspects of HTTP, such as hidden field correlation, redirects, dependent requests and HTTPS/SSL.

Recording a web test in Visual Studio is relatively easy, and begins by starting Internet Explorer with an additional panel that represents the recorder tool itself. As the tester proceeds with the test scenario, the web test records all HTTP requests. Web test is most suitable when performing simple functional tests and when testing availability and navigability of a web application. For instance, a tester can easily create a web test that tests the availability and links of all web pages for a web application.

However, web tests in Visual Studio do not provide the dynamic and rich features provided by TestComplete. It is true that a tester can create data-driven web tests and convert the recorded HTTP requests in C# so as to add looping and branches: but the web test is only based on recording HTTP requests. On the other hand, TestComplete provides extensive flexibility that enables testers to write test scripts that can access, evaluate and manipulate all kinds of data and on-screen objects on a web page.

4. DISCUSSION

Automation testing for the GUI is important since many problems only manifest themselves at the GUI. Also some back-end changes in the code could have a considerable effect on GUI functionality. However, automating the GUI is difficult because the user interface changes frequently. For this reason automation test scripts need to be simple, well designed and maintainable.

Relying only on the recorder tool in TestComplete to generate test scripts can result in fragile scripts. These can break easily whenever minor changes are made to the GUI. When testing dynamic web pages, recorder-generated scripts fail to execute almost every time.

On the other hand, writing robust and simple test scripts that utilize the API of TestComplete, according to the proposed solution algorithm, has proven to solve those problems. Testers should build the test scripts based on modules and libraries that consolidate common and generic code, ending with easy to maintain scripts that can enable testers to keep pace with development when the GUI is changed.

The Web Test tool provided by Visual Studio does not provide the needed flexibility when writing complex test scripts as provided by TestComplete.

5. CONCLUSION

This paper has shown how the TestComplete recorder tool failed to generate robust test scripts that can be played back without failing when recording tests for web applications. The tester used the TestComplete recorder tool to record a test against specific test scenario for dynamic web applications. After recording the test, it failed to later run back successfully, as some of the onscreen objects could not be recognized by the TestComplete engine. This paper also introduced a robust solution test script, run by the tester against the same test scenario, which did not depend on the TestComplete recorder tool. This solution proved to be robust and TestComplete ran it without failing. The solution test script addresses dynamic web pages where their onscreen objects can change hierarchy and attributes from one page run to another. Solution script also addresses slow loading web pages, by waiting for onscreen page objects to load. Writing test scripts in this pattern provides software testers with full control of their test case and addresses complex automated test scenarios.

Automating manual GUI testing scenarios by test tools such as TestComplete will definitely save testers from repetitive and time-consuming tasks, giving them additional time to focus on writing more creative test cases. In large software development projects, where software is developed rapidly, testers have no choice but to use test automation tools. However, automation test scripts for GUI's need to be flexible, maintainable and based on modules and libraries for common code.

Rich Internet Applications are becoming more and more famous, such as Microsoft Silverlight. TestComplete 8.5 is seen weak when recording and executing automated tests against Silverlight applications using its recorder tool, by which recorded tests fail to sometimes run successfully. This is an important area that needs to be resolved in future studies.

6. REFERENCES

[1] Whittaker, J. (2009) *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*, Boston: Addison Wesley.

[2] Desikan, S. & Ramesh G. (2006) *Software Testing: Principles and Practices, Pearson Education*: India.

[3] Sommerville, I. (2004) *Software Engineering*, England: Addison Wesley.

[4] Bruegge, B. & Dutoit, A. (2004) *Object Oriented Software Engineering, Using UML, Patterns, and Java*, NJ: Prentice Hall.

[5] Dustin, E. & Garrett, T. & Gauf, B. (2009) *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*, Boston: Addison-Wesley Professional.

[6] McWherter, J. & Hall, B. (2009) *Testing ASP.NET Web Applications*, Indiana: Wrox.

[7] Riley, M. (2010) *DevProConnections*. Available at: http://www.devproconnections.com/article/software-testing/Review-SmartBear-Software-s-TestComplete-8-Enterprise (Accessed 15 April 2011).

[8] SmartBear Software (2011) *Top Reasons to Try TestComplete*. Available at: http://www.automatedqa.com/products/testcomplete/top-reasons-to-try/ (Accessed: 12 May 2011).

[9] Afroz, M. & Rani, L & Priyadarshini, N. (2011) 'Web Application - A Study on Comparing Software Testing Tools', *International Journal of Computer Science and Telecommunications*, 2(3).

[10] Business Internet Group of San Francisco (2003) *The Black Friday Report on Web Application Integrity*. Available at: http://www.tealeaf.com/downloads/news/analyst_report/BIGSF_BlackFridayReport.pdf. (Accessed: 9 May 2011).

[11] Business Internet Group of San Francisco (2003) *The BIG-SF Report on Government Web Application Integrity*. Available at: http://www.tealeaf.com/downloads/news/analyst_report/BIG-SF_Report_Gov2003-05. (Accessed: 9 May 2011).

[12] SmartBear Software (2011) *Testing Dynamic Web Pages*. Available at: http://smartbear.com/support/viewarticle/12725/ (Accessed 16 April 2011).

[13] SmartBear Software (2011) *Web Tree Models*. Available at http://smartbear.com/support/viewarticle/12439/ (Accessed: 5 June 2011).

[14] Crispin, L. & Gregory, J. (2009) *Agile Testing: A Practical Guide for Testers and Agile Teams*, Boston: Addison Wesley.