F10S – Software Engineering

Testing graphical user interfaces

`Design of a framework which can help a developer test graphical user interfaces, for Adobe Flex and ActionScript.`

Esbjerg Institute of Technology
Chris D. M. Kjærsig
February 1st, 2012 – June 6th, 2012

Preface

Title sheet

Theme:	Software Engineering
Title:	Testing graphical user interfaces
Group:	Chris D. M. Kjærsig
Supervisor:	David Hicks
Period:	February 1st, 2012 – June 6th, 2012
Field of study:	Software Engineering

Chris D. Morratz Kjærsig

Abstract

Testing graphical user interfaces becomes increasingly difficult as the complexity of the user interface increases, there is however only little help for a developer to find when it comes to testing and maintaining an application providing a GUI. This paper examines some of the few possible methods available to a developer for testing a GUI and also includes a proof of concept for a framework to test GUI applications with, that incorporates the concept of unit testing.

Contents

Chapter 1

Project description

1.1 Introduction

Adobe Flash has been used for many years to create games on the web, however it was not a very effective tool for developing applications, until recently. With the arrival of Adobe Flex 1.0 in 2004 Flash was beginning to look like a possibility for developing Rich Internet Applications (RIA) [5], besides games, and with the addition of Adobe AIR in 2008 it became possible to develop desktop applications entirely in Flash.

This project has been partly done in collaboration with a company known as Intertisement, they have for a while now been working on two projects for the furniture company BoConcept. The two projects are both RIA's developed in Adobe Flex with ActionScript, one which is BoConcepts online product configuration system [2], where it is possible to select a piece of furniture from their wide collection and modify it from a Flex based application, e.g. changing color, legs or adding additional modules to create a completely new kind of furniture. The other is a room configuration application which enables the user to draw rooms, insert furniture and also configure them like in the product configurator. Developing this kind of application is not straightforward, since there are a lot of different pieces that has to fit together, and it gets even worse when scaling this kind of application from a product configurator to a room configurator. Suddenly there are a lot more pieces that has to fit together and without any kind of test suite it becomes increasingly harder to make sure that everything works the way it is expected to.

Today there are a lot of tools and frameworks which can help developers test non-GUI applications, e.g. by writing tests for isolated parts of the system, also known as unit testing, some examples are:

- jUnit – A unit testing framework for Java
- unittest – A unit testing module for python

- FlexUnit – A unit testing framework for Adobe Flex, not the GUI part of the application however

However when dealing with a GUI application there are only few different tools available and when looking at GUI testing in Flex, there are only capture/replay tools available. Most of the tools requires a lot of time and effort by the developer in order to create a series of tests, that can be used to verify the correct behavior of the application. Some of the available methods will be examined later on.

1.2 Goal of the project

The goal of this project is to provide a framework which can help a developer create unit- and integration tests for a GUI application, and to provide a simple graphical user interface for inspecting test results. The framework will be developed for Adobe Flex and ActionScript testing.

Chapter 2

Related work

2.1 Capture/replay tools

Capture/replay tools are the most popular way to test GUI's. It requires someone to interact with the user interface, the interaction is then recorded and then the expected result is specified by the user and associated with the recorded interaction. This can be a time consuming process and is in no way automated. [7]

There are two types of capture replay tools, the first one captures raw mouse movements or keystrokes and then takes a snapshot of the screen. These kind of tools produce unmaintainable tests, since at every change of the layout all tests have to be recreated for the new layout.

The second one is more usable since it can work with the actual objects and widgets in the GUI application. This way it is possible to run the same tests with the same result even if the layout is changed. However this does still not guarantee 100% maintainability, since changing e.g. a label to a button will require all the tests for that specific label to be rewritten. [6]

2.2 GUI reverse engineering

Reverse engineering is the process of determining the model of an application based on the executable. One way to do this is to traverse all windows in the GUI application and reading the properties (e.g. font, background color), values (e.g. the label of a button) and the events listened to by all the available widgets. These values are then stored in a format that can be used by the testing tool to run the automated GUI tests and verify the results.

Memon et al. [7] does this by creating two trees, one with the widgets where a node is a widget and an edge is present from node A to node B if an event in window A triggers the opening of window B. The other one is an Event-flow graph where the nodes represents types of events and the

edges represents a sequence of events. The process of reverse engineering a GUI consists of two steps. The first step is to extract the structure of the application, the next step is to correct the extracted data, since if there are errors in the application it will not conform to specification, for that reason the incorrect parts has to be corrected manually.

There are both pros and con to this method, first of all this method requires very little human interaction, making it an easier way to test the user interface compared to using capture/replay testing. One problem is that the data for testing is extracted directly from the executable and that there are no standard way to create GUI applications across different platforms. This means that the implementation of a tool for reverse engineering GUI's will have to use low level system calls and there needs to be an implementation of the tool for each supported platform, also some implementations might not provide enough information needed for automated testing. Another problem is that not all windows that can be opened from the application might be available, e.g. if a window needs a password to open, this will not be possible to extract by means of reverse engineering and therefore these kinds of holes needs to be filled in manually afterwards. [7]

2.3 Rich Internet Applications

Rich Internet Applications can be implemented using a variety of methods, but it can roughly be broken down into two parts. One part is the client side application which provides a graphical user interface. The other part is an optional server side implementation that can provide different services for e.g. storing data or doing intense asynchronous computations not suited for the client, etc..

A study in 2010 by Amalfitano et al. [3] explores the possibility of using execution traces to generate a Finite State Machine (FSM) model that can be used for GUI testing. The idea is to obtain a FSM (S, T, E), where S is a set of states in the user interface, T is a set of transitions between states and E is a set of events that triggers a transition between states. Amalfitano et al. also makes use of an Event-flow graph which was introduced by Memon et al., the Event-flow graph can be obtained using reverse engineering algorithms.

The execution traces can be obtained in two ways, one way is to collect execution traces from users/testers of the application and another way is to obtain them from a crawler of the application. A crawler is a tool that can be used to get the model of an application, this can done by executing all of the applications client side code. For example an Ajax crawler will trigger all events accessible from the webpage, e.g. clicking on buttons, anchors, etc., it then saves the user interface states, in this case the DOM states, reached after the javascript that is associated with an event has been executed.

Some of the problems involved with this approach are that the obtained execution traces might not be enough to accurately test all aspects of the application, especially not if they are only obtained from users or testers of the application. In that case the problem of generating accurate test cases will be the same as for capture replay tools. Here the crawler might be able to create more accurate test cases, but not all of the available technologies for creating RIA's are suited for a crawler.

2.4 GUI Unit testing

Most of the previously mentioned methods consists of the following steps:

1. Write source code
2. Compile application
3. Extract events, properties and values from application
4. Create test suites

Most of these steps are done automatically, with just a small amount of manual work, except for the capture/replay method. There is however another aspect that the previous methods are not so useful for.

When developing an application it is not only important to test the application, but also to ensure its maintainability. By incorporating the idea of test driven development (TDD), which is used often in agile software development [4], there is a natural need for structuring the source code since the idea is to split the problem into the smallest possible parts and attacking each part of the problem separately, one at a time. Basically TDD consists of three steps:

1. Write a test. Here it is important to imagine exactly how you want it to work, without thinking about the actual implementation.
2. Make it work. Write the simplest implementation that passes the test. This can be done in two ways:
   - by faking it: returning a constant is a valid way to pass the test
   - by using the obvious implementation: write the real implementation
3. Make it right. Go back to the working but (maybe) not so correct implementation and correct it so that the test passes with the correct implementation

This method provides both testing capabilities, although not as automated as some of the other methods, but it also provides a way to enforce a certain structure when developing.

Often developing a large GUI application will follow an agile methodology, since it is not always easy for the customer to get a feel for the user interface before they have used it. This means that when developing a user interface it is necessary to be in contact with the customer, creating the new feature, getting feedback, refactoring and so on in small iterations. In case a bug is found a new test case is created specifically for this bug.

A unit test is created for each widget class, and mockups are created to make sure that the widget class that is being tested is completely isolated from other classes, to test it without influence from outside the class.

There is however no solution that can currently do this for graphical user interfaces.

Chapter 3

Framework

Looking back at the methods examined in section 2, there are pros and cons to all of them. Reverse engineering seems unnecessary in this case, keep in mind that this is intended for the developer to test the elements of a graphical user interface and in this case the source code will always be available. In addition since this is intended mainly for the Flex part of the application, most of the source code is already in a format which makes it easy to parse. Next there is the capture/replay method, with this it is possible to assign what the expected outcome of a user interaction should be, record it and then replay it afterwards. While this might work quite well in most cases it is a slow processes which in all possibility will have to be repeated when the GUI changes. In addition it is my opinion that when testing software it is required to structure the software so it is both easier to test and to refactor. Since the other methods only looks at the resulting GUI after compilation, it does not enforce a structured way of developing, that is why the unit testing approach will be used for this framework.

3.1 Requirements

It should be possible to

- create test cases to test event listeners
- create test cases to test methods
- create test suites consisting of multiple test cases
- create mock objects to substitute classes that might influence the test (e.g. custom classes that the class being testet relies upon)
- displaying the test result so it is easy to inspect

For a test case it should be possible to do the following:

- dispatch an event on a widget, which may change properties and/or values by means of a listener function
- change properties and values directly from a test
- assert properties and values of a widget, to ensure that the widget behaves correctly.

There is however one major problem that needs to be addressed, namely actions that does not block the interface, in Flash these are mostly animations. For example, if a user clicks on a button that rotates an image by 90° with an animation, then to assert whether the image has actually been rotated correctly, the animation needs to finish playing before validating the images properties. At the time of writing there is no known standard way to do this in Flex, some minor tweaks are needed that does not inconvenience the developer.

3.2 Implementation

The implementation consists of a number of classes:

- GUITestSuite – This class is used to group test cases.
- GUITestCase – Tests can be added to this class and run either on its own or by adding it to a GUITestSuite.
- GUIMockObject – This class is used to mock objects that could otherwise influence the unit test
- MouseEventEmitter – This is a helper class that can be used to dispatch mouse events to a class of the type EventDispatcher.
- KeyboardEventEmitter – This is a helper class that can be used to dispatch keyboard events to a class of the type EventDispatcher.
- AssertionError – This error is raised if an assertion has failed in the test

Figure 3.1 provides an overview of how the above mentioned classes associate.

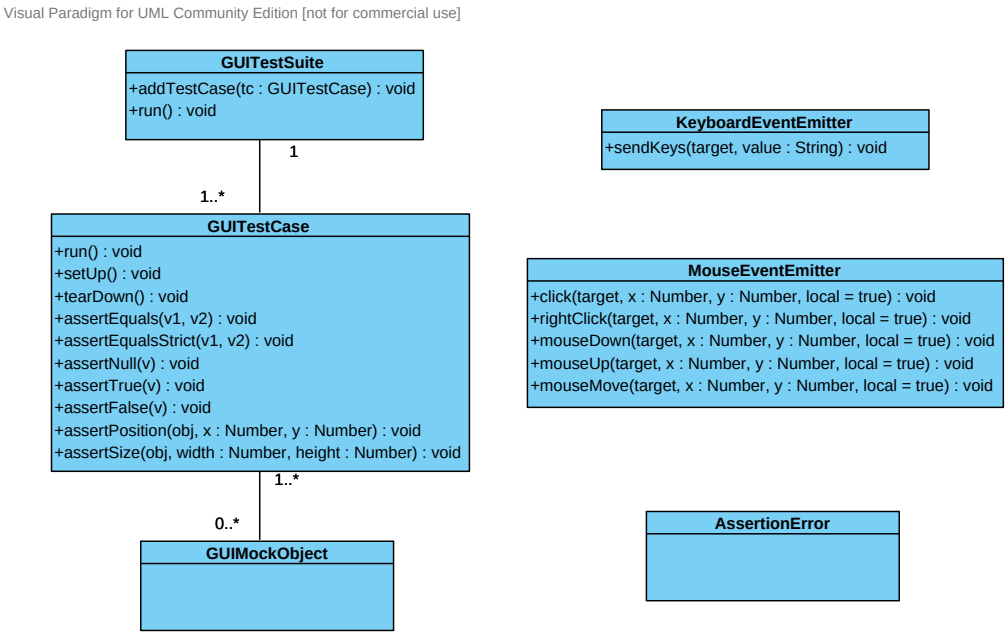

Figure 3.1: Overview of the frameworks classes

3.3 Testing

A simple proof of concept has been created which supports writing test cases, grouping them into test suites, however mock objects are currently not supported and the support for testing widgets with animations is unreliable at best. At the moment there is no GUI for viewing the test result either, it is only available in text format.

In listing 3.1 there is an example of how to use the framework. By including the GUITestCase and MouseEventEmitter it is possible to test the effect of mouse events on a class. The class being tested is in the global scope of the application and will not be reinitialized before cach test is run. The source code of the class that is being tested is fairly straight forward, when it receives a mouse down event it sets a read-only property to true and on mouse up it sets the same property to false.

Listing 3.1: Framework example

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
3                 xmlns:s="library://ns.adobe.com/flex/spark"
4                 xmlns:mx="library://ns.adobe.com/flex/mx"
5                 xmlns:c="components.*"
6                 creationComplete="created()">
7
8      <fx:Script>
9          <![CDATA[
10         import com.ck.gui.GUITestCase;
11         import com.ck.emitter.MouseEventEmitter;
12
13         private function created():void
14         {
15             /* Create test case */
16             var tc:GUITestCase = new GUITestCase();
17
18             tc.test_mouseDownEvent = function ():void
19             {
20                 // Emit a mouse down event
21                 MouseEventEmitter.mouseDown(testClass, 0, 0);
22                 // Verify that mousePressed is true
23                 this.assertTrue(testClass.mousePressed);
24             }
25
26             tc.test_mouseUpEvent = function ():void
27             {
28                 // Emit a mouse up event
29                 MouseEventEmitter.mouseUp(testClass, 0, 0);
30                 // Verify that mousePressed is false
31                 this.assertFalse(testClass.mousePressed);
32             }
33
34             // Run tests
35             tc.run();
36         }
37         ]]>
38     </fx:Script>
39
40     <c:TestClass id="testClass" />
41 </s:Application>
```

The source code for the test class can be found in listing 3.2.

Listing 3.2: Content of TestClass

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Canvas xmlns:fx="http://ns.adobe.com/mxml/2009"
           xmlns:s="library://ns.adobe.com/flex/spark"
           xmlns:mx="library://ns.adobe.com/flex/mx"
           mouseDown="onMouseDown(event)"
           mouseUp="onMouseUp(event)">

    <fx:Script>
        <![CDATA[
        import flash.events.MouseEvent;

        private var _mousePressed:Boolean = false;
        public function get mousePressed():Boolean
        {
            return _mousePressed;
        }

        private function onMouseDown(event:MouseEvent):void
        {
            _mousePressed = true;
        }

        private function onMouseUp(event:MouseEvent):void
        {
            _mousePressed = false;
        }
        ]]>
    </fx:Script>
</mx:Canvas>
```

The test is fairly simple but shows that it is possible to write a unit test for a widget and even using events to change the value of a property, validating it after the mouse event has been triggered and the property mousePressed has changed. In the above example it is further necessary to note that it is possible to group different tests together without resetting the state of the widget being tested. In the example, if test_mouseUpEvent was changed slightly as seen in listing 3.3, then the test would still pass, since the test_mouseDownEvent sets the value of mousePressed to true, by emitting a mouse down event on the target widget.

Listing 3.3: Updated test_mouseUpEvent

```
    tc.test_mouseUpEvent = function ():void
    {
        // Verify that mousePressed is still true, indicating
        // that the state of the widget has not been reset
        // between the two tests.
        this.assertTrue(testClass.mousePressed);

        // Emit a mouse up event
        MouseEventEmitter.mouseUp(testClass, 0, 0);
        // Verify that mousePressed is false
        this.assertFalse(testClass.mousePressed);
    }
```

It is possible to create a test for any kind of relation between two events, this can be useful to test two events that should do the opposite of each other. In the above example both the mouse up and the mouse down listener changes the same property, so it can be tested if activating both events resets the state to what it was before the mouse down listener was activated.

It is also possible to create two tests like above where the target widget is reset to its initial state before running each test. The reason the widget in the above test was not reset, is because it was in the global scope of the application and that it was not removed after a test and re added before the next test. This can be done by using the setUp and tearDown functions of the GUITestCase class. The setUp function, if specified, is called everytime before a test is run and the tearDown function, if specified, is called everytime after a test has run, but before the setUp function is called again for the next test. To simplify it a bit, the test in listing 3.4 will be run in the following order:

1. setUp
2. test_mouseDownEvent
3. tearDown
4. setUp
5. test_mouseUpEvent
6. tearDown

Listing 3.4: setUp and tearDown methods

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
3                 xmlns:s="library://ns.adobe.com/flex/spark"
4                 xmlns:mx="library://ns.adobe.com/flex/mx"
5                 creationComplete="created()">
6
7      <fx:Script>
8          <![CDATA[
9          import com.ck.gui.GUITestCase;
10         import com.ck.emitter.MouseEventEmitter;
11         import components.TestClass;
12
13         private function created():void
14         {
15             /* Create test case */
16             var tc:GUITestCase = new GUITestCase();
17
18             // Assign the setUp function
19             tc.setUp = function ():void
20             {
21                 // Create a new test class each time before running a test
22                 this.testClass = new TestClass();
```

```
23                  // Add the test class to the application
24                  addElement(this.testClass);
25              }
26
27              // Assign the tearDown function
28              tc.tearDown = function ():void
29              {
30                  // Remove the test class from the application after each
                        test has finished
31                  removeElement(this.testClass);
32                  this.testClass = null;
33              }
34
35              tc.test_mouseDownEvent = function ():void
36              {
37                  // Emit a mouse down event
38                  MouseEventEmitter.mouseDown(this.testClass, 0, 0);
39                  // Verify that mousePressed is true
40                  this.assertTrue(this.testClass.mousePressed);
41              }
42
43              tc.test_mouseUpEvent = function ():void
44              {
45                  // Emit a mouse up event
46                  MouseEventEmitter.mouseUp(this.testClass, 0, 0);
47                  // Verify that mousePressed is false
48                  this.assertFalse(this.testClass.mousePressed);
49              }
50
51              // Run tests
52              tc.run();
53          }
54          ]]>
55      </fx:Script>
56  </s:Application>
```

If the changed version of test_mouseUpEvent from listing 3.3 is used in listing 3.4 an AssertionError will be raised, since the value of mousePressed has been reset and is now false.

The GUITestCase class is a dynamic object, meaning that any property can be created at any time without getting compile or runtime errors. This is the method used for adding tests. When calling the run method of the GUITestCase class all the functions starting with *test_* will be run in the order they were added.

Chapter 4

Conclusion and future work

Developing large and complex GUI application can become a very complicated task, not only to test but also to maintain. To be able to develop, test and refactor the application time and time again it is not necessary, but rather preferred, that the testing framework enforces some kind of structure to the development process. By incorporating the test driven development (TDD) approach the developer will be forced to think about under which circumstances a unit is used and thereby also to think about how the interface of that unit should be.

A small proof of concept has been created that supports the most important part of the testing framework, test cases and test suites. It is possible to write tests for a Flex application that can test how events can change the state inside the application. There are still some issues though, a major one is the problem with animations, since they do not block code execution, meaning that when an animation is started the rest of the code, after the line that the animation is created at, will be executed even though the animation is still playing. This will have to be fixed in a future version of the framework before it is ready to be put into a full scale test.

On a last note the TDD approach is not always recommended. When working on a simple GUI application, writing tests might take up 80% or more of the actual development process. But if it turns out later that the simple GUI application needs to be improved and it suddenly grows very complex, it could become a major task to restructure the application to make unit testing possible. So it is pretty much an assessment that has to be made when starting development or at least as early in the development process as possible.

List of references

[1] Adobe flex. URL `http://www.adobe.com/products/flex/`.

[2] Boconcept product configurator. URL `http://www.boconcept.us/indivi-2.aspx?ID=83236&imageid=3639&hc=true`.

[3] D. Amalfitano, A.R. Fasolino, and P. Tramontana. Rich internet application testing using execution trace data. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 274 –283, april 2010. doi: 10.1109/ICSTW.2010.34.

[4] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[5] G. Lawton. New ways to build rich internet applications. *Computer*, 41 (8):10 –12, aug. 2008. ISSN 0018-9162. doi: 10.1109/MC.2008.302.

[6] B. Marick. Classic testing mistakes. *STAR East*, 1997.

[7] A. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *proceedings of the 10th working conference on reverse engineering (WCRE'03)*, volume 1095, pages 17–00. Citeseer, 2003.

[8] A.M. Memon. Gui testing: Pitfalls and process. *IEEE Computer*, 35(8): 87–88, 2002.

[9] A.M. Memon, M.E. Pollack, and M.L. Soffa. Hierarchical gui test case generation using automated planning. *Software Engineering, IEEE Transactions on*, 27(2):144 –155, feb 2001. ISSN 0098-5589. doi: 10.1109/32.908959.