F10S – Software Engineering

Testing graphical user interfaces

`Developing and testing graphical user interfaces, with focus on Adobe Flex and ActionScript.`

Esbjerg Institute of Technology
Chris D. M. Kjærsig
February 1st, 2012 – August 10th, 2012

Preface

Title sheet

Theme:	Software Engineering
Title:	Testing graphical user interfaces
Group:	Chris D. M. Kjærsig
Supervisor:	David Hicks
Period:	February 1st, 2012 – August 10th, 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 grows. This paper examines both testing tools and methodologies that can help increase the quality and maintainability of a GUI application. Software development life cycle models are examined to see which is a better match when developing GUIs, also the test-driven development and the model-view-controller approaches are examined to determine if they can provide any value to the developer.

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] in 2004 Flash was beginning to look like a possibility for developing Rich Internet Applications (RIA) [8], 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.

The room configurator is very large and complex compared to the product configurator, in addition the room configurator has been developed in a way that makes it hard to maintain. This is mainly due to tight coupling of certain classes, because a new feature or bug fix in one class can cause a bug in another class. Without a test suite it gets increasingly difficult and time consuming to develop and maintain the application.

There are two aspects that will be examined in this paper, the first is (semi-)automated testing tools, these should be able to help the developer test a GUI application easier than by hand. The second aspect is to structure the code so it is both easier to read and so developing new features has a smaller risk of breaking existing functionality.

1.2 Goal of the project

The goal of this project is to examine methods and tools that can make GUI development and testing less painful and also makes it easier to maintain the application in the long run.

Chapter 2

Related work

This chapter will focus on the related work of multiple aspects of software development and testing:

- (semi-)automated testing – of the compiled application
- software development life cycle models – to structure development from inception to end
- development methods – techniques to produce testable and maintainable software

The first part *Testing tools* will describe some of the available methods that can be used to test graphical user interfaces based on the executable of the application.

The second part *Software development life cycle models* will describe the process for developing GUI applications and based on that propose some SDLC models that can be followed when developing a GUI application.

The third part *Development methods* will describe different methods that can be used to make testing and maintaining a GUI application easier.

The fourth part is a discussion that compares pros and cons of automated testing compared to applying a methodology that aims for better testing and maintainability.

2.1 Testing tools

2.1.1 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. [11]

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. [10]

2.1.2 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. [11] 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. [11]

2.1.3 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.

2.2 Software development life cycle (SDLC) models

Developing a GUI application, like any other application, is comprised of a number of steps. First requirements should be gathered, then the application is designed. Next step is to implement the application and then it should be tested. Finally the application is evaluated based on the initial criteria, to determine whether the application has been finshed or if something is still missing. When the development process is done and the application accepted, there might be maintenance in the future, e.g. new features or bug fixes.

However a GUI application can only be compared to a non-GUI application to some degree. A non-GUI application is considered done when the original requirements has been fulfilled. A GUI application also needs to fulfill its original requirements, but in addition it should also be intuitive and user friendly and that is most likely the hardest part of developing a GUI application. It is almost impossible to decide what is user friendly until the actual users have tried the application and given their feedback, although an application might benefit from doing the same as other applications of the same type. It also depends on the target users, e.g. it is harder to create a user friendly GUI for a non-expert user since they cannot be expected to learn how the application is used. Often they want an intuitive way, that does not require them to read a large instruction manual, to be able to use the application. Expert users on the other hand might simply want an easy way to do specific tasks, e.g. shortcuts for common operations, instead of using the mouse, to make them work faster, since they might need to do the same or similar tasks many times a day because of their job. Looking at the existing SDLC models, it seems some are better suited for GUI development than others[12, 7].

There are many different SDLC models, for many different types of applications. In general they can be put into three categories[12]:

- Linear model
- Iterative model
- A combination of linear and iterative models

A linear model is a sequence of steps where the completion of a step immediately leads to the beginning of the next, with no way to go back. An iterative model ensures that all steps may be revisited at some point in the process.

Developing a graphical user interface will in most cases be an iterative task, at the very least on the design part. However starting out with fairly basic criterion and adding new ones as the existing are fulfilled ensures that it is possible to adapt the application before it is too late in the process, e.g. if the customer decides the initial GUI design is not as easy to use as expected, or if some core functionality has to be changed or is missing. Also keeping in touch with the customer and adapting to new wishes ensures that they actually get what they are paying for, instead of a bad experience. An SDLC model that is suitable for this development style is the Rapid Application Development (RAD) methodology, which, among others, encompasses Scrum, Agile and Extreme Programming (XP)[12].

2.3 Development methodologies

2.3.1 Test-driven development

When developing an application it is not only important to test the application, but also to ensure its maintainability. The idea of test driven development (TDD), which is used often in agile software development [4], is to split the problem into the smallest possible parts and then try to solve each part of the problem separately, one at a time. There are two simple rules to TTD [4]:

- only write new code if a test has failed.
- eliminate duplication

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

The question is why use the TDD approach? Beck [4] has a good analogy:

> "Imagine programming as turning a crank to pull a bucket of water from a well. When the bucket is small, a free-spinning crank is fine. When the bucket is big and full of water, you're going to get tired before the bucket is all the way up. You need a ratchet mechanism to enable you to rest between bouts of cranking. The heavier the bucket, the closer the teeth need to be on the ratchet."

What he means is that the tests in the TDD approach are the teeth on the ratchet, the more complex an application the more tests that cover very specific parts of the application are required. This makes it easier to isolate problems to a very specific part of the application and also requires the different classes/modules of the application to be loosely coupled.

In case it is unavoidable that two classes depend on each other there is a possibility that changes to one class introduces an error in another class and it may also be hard to determine which of the classes actually produces the wrong behavior. To avoid this problem a common method is to use mock objects [9]. A mock object is a substitute implementation that emulates a class, the idea is to keep the mock object simple, e.g. by returning the same value each time a function is called. This way classes that depend on each other can be tested independently, without interference from each other.

Unit testing

Unit testing is a type of testing where the developer writes tests for a *unit*, which is usually a class or module and it is usually used in conjunction with TDD. It is often provided as a framework that supports writing the tests, running them and then display some output containing which tests failed and why. There are many frameworks that allows this kind of testing in multiple programming languages like jUnit for Java, the unittest module for Python or FlexUnit for Flex applications. However applying this to a GUI application can be difficult [13].

GUI unit testing

When testing a GUI application it is necessary to have programmatically access to the on-screen components and a way to manipulate them, this makes it necessary to have a class that can simulate user input. There are a few GUI unit testing frameworks available for Java, these include UISpec4J, jfcUnit, Abbot and LIFT, however most of them a very complex and requires a lot of experience to use [13]. There is no equivalent for Flex.

2.3.2 Model-View-Controller

Another approach that can be used is the Model-View-Controller or MVC approach. Here the idea is to split the application into three parts[14, 6]:

- a model, which contains application data and business logic
- a view, which is a graphical representation of data
- a controller, which communicates user interaction to the model and the view.

See figure 2.1 for a brief overview.

The motivation for MVC is code re-usability and separation of concerns, considering that more time goes into developing the graphical user interface, than the rest of the application[6]. The MVC approach allows the developers

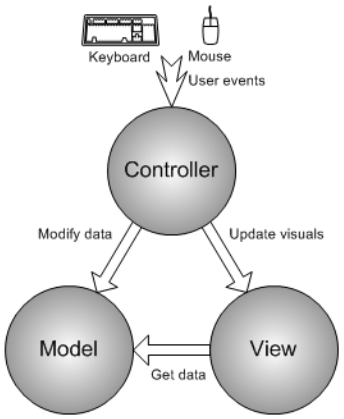

Figure 2.1: MVC overview

to change the design of the user interface without making changes to the functionality.

When the user interface and the functionality has been separated it is possible to unit test the functionality without concerns for the graphical part of the application.

Although MVC sounds simple enough in theory there are a lot of interpretations of how this should be done in practice. There are MVC frameworks that does things very differently and many of the frameworks have a very steep learning curve. Because of this it can, in some cases, be faster to choose a simpler method.

2.4 Discussion

Testing in general is a comprehensive process and it is necessary that all functionality is tested to make sure the product lives up to expectation. However the closer each class/module of the application is coupled the greater the possibility that something breaks due to a new feature or a bug fix. The benefits of a (semi-)automated testing tool is that the developer can easily determine if something broke due to a bug fix or feature, however it is not always easy to determine exactly where the problem occurred, because in case two or more classes affect each other the error could be in any of the interacting classes. Using the TDD approach with the (semi-)automated testing tools is not possible since the tests cannot be created until a GUI is available, and the TDD approach requires tests to be present before implementing anything.

Writing unit tests can be time consuming [5] but when using TDD it forces the developer to think more about the functionality before actually starting to code. In simple applications it might be too much work to write tests and in those cases a simple test like capture/replay might be much more efficient,

but in larger and more complex applications it is necessary to have a more detailed view of the errors and that requires a certain extent of structure to the code, in these cases TDD and unit testing is the way to go.

In some cases it might even be a good idea to follow the MVC approach, as this approach still supports unit testing and TDD, but it also provides the developer with a way of separating the GUI from the application logic. With this it is easy to refactor the GUI later without concern for breaking the underlying logic.

Whether one follows the TDD approach, with or without including MVC, the most important part is to be flexible regarding development, instead of specifying all requirements from the beginning and then implementing the application just so the customer can tell you that this was not what they expected. It is by far better to incrementally implement parts of the application and showing it to the customer before continuing, this way a design flaw can be caught early in the process and thus save time in the long run.

Chapter 3

GUI unit testing framework, for Adobe Flex

There are unit testing frameworks available for Adobe Flex however none of them has out-of-the-box support for programmatically interacting with the GUI. That is why a proof of concept has been created with the possibility for merging it with an existing unit testing framework at some point.

3.1 Requirements

First of all it should be simple for a developer to write tests, and 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 tested relies upon)
- displaying the test result so it is easy to inspect

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

- emulate user input programmatically
- 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 emulate user input (mouse events).
- KeyboardEventEmitter – This is a helper class that can be used to emulate user input (keyboard events).
- 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.

3.3 Testing

A simple proof of concept has been created which supports writing test cases, grouping them into test suites, however 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.

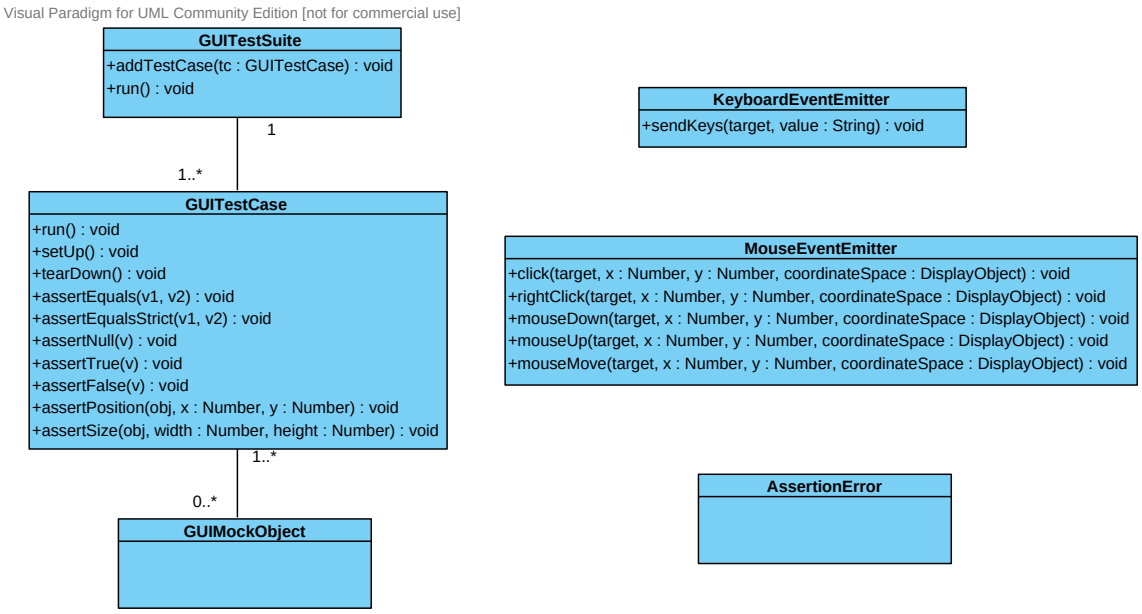

Figure 3.1: Overview of the frameworks classes

Listing 3.1: Framework example

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               xmlns:c="components.*"
               creationComplete="created()">

    <fx:Script>
        <![CDATA[
        import com.ck.gui.GUITestCase;
        import com.ck.emitter.MouseEventEmitter;

        private function created():void
        {
            /* Create test case */
            var tc:GUITestCase = new GUITestCase();

            tc.test_mouseDownEvent = function ():void
            {
                // Emit a mouse down event
                MouseEventEmitter.mouseDown(testClass, 0, 0);
                // Verify that mousePressed is true
                this.assertTrue(testClass.mousePressed);
            }

            tc.test_mouseUpEvent = function ():void
            {
                // Emit a mouse up event
                MouseEventEmitter.mouseUp(testClass, 0, 0);
                // Verify that mousePressed is false
                this.assertFalse(testClass.mousePressed);
            }

            // Run tests
            tc.run();
        }
        ]]>
    </fx:Script>

    <c:TestClass id="testClass" />
</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;
        }

```

```
18          private function onMouseDown(event:MouseEvent):void
19          {
20              _mousePressed = true;
21          }
22
23          private function onMouseUp(event:MouseEvent):void
24          {
25              _mousePressed = false;
26          }
27          ]]>
28      </fx:Script>
29  </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

```
1   tc.test_mouseUpEvent = function ():void
2   {
3       // Verify that mousePressed is still true, indicating
4       // that the state of the widget has not been reset
5       // between the two tests.
6       this.assertTrue(testClass.mousePressed);
7
8       // Emit a mouse up event
9       MouseEventEmitter.mouseUp(testClass, 0, 0);
10      // Verify that mousePressed is false
11      this.assertFalse(testClass.mousePressed);
12  }
```

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 in the *setUp* method and is no longer true.

It is also possible to use more complex operations, e.g. press mouse button, move mouse, release mouse button. To do this the operations available in MouseEventEmitter can be combined, as can be seen in listing 3.5.

Listing 3.5: Advanced operations

```
1   tc.test_dragging = function():void
2   {
3       // Press mouse button
4       MouseEventEmitter.mouseDown(this.testClass, 0, 0);
5
6       // Move mouse 100 pixels in x and y direction (right and down)
7       MouseEventEmitter.mouseMove(this.testClass, this.testClass.x + 100,
8       this.testClass.y + 100, this.testClass.parent);
9
10      // Release mouse button
11      MouseEventEmitter.mouseUp(this.testClass, 0, 0);
12
13      // Verify position of the display object
14      this.assertPosition(this.testClass, 100, 100);
15  }
```

The last argument in MouseEventEmitter.mouseMove indicates which coordinate space the specified x and y values are in. If not specified the values are local to the target widget.

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 applications can become a very tedious 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 necessary to have a structured development process. When developing a GUI application it is necessary to be in close contact with the customer, to constantly give them a feel for the application and to get feedback on what was not not user friendly, and what were missing, that they were expecting to be able to do. An iterative SDLC model is needed and a possible choice could be Agile or Extreme Programming. In addition it is necessary to test application behavior, and this in a way that does not require a lot of manual labor every time a test has to be done. By incorporating TDD into the development process, the developer is forced to think of the actual behavior of a unit, and thereby also think of how the units interface should be.

Another addition that can be used in conjunction with TDD is MVC, which means that the application is split into three layers, the models which are data or business logic, the views which are the graphical user interfaces and the controllers which communicates user interactions to the model and the view. With this abstraction it is possible to completely separate the graphical user interface from the application logic. This makes it possible to make changes to the user interface without breaking the business logic. The downside is that the available frameworks for MVC does not agree on the implementation, and that most of them has a very steep learning curve. So when switching from one MVC framework to another, there is a risk the developer has to start from scratch.

A small proof of concept for a unit testing framework for GUI applications, has been created in Adobe Flex and ActionScript. It 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 mouse and keyboard interaction 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.

From my own experience it is not always recommended to apply TDD and writing a lot of tests. It depends on the project size, the larger the project, the more you may benefit from a structured approach. It also depends on the number of co-developers, the more developers on a project, the more benefit you will gain from using a structured approach. Also when working together with multiple developers it is of great importance to create tests, this way the other developers can test if a component behaves incorrectly after they have made changes to it, and if it does behave incorrectly they can use the failed tests to see why it has broken.

In a company situation it will not always be possible to apply the methods mentioned in this paper, since parameters as budget and time also will play a role in choosing the model.

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] Y. Cheon and G. Leavens. A simple and practical approach to unit testing: The jml and junit way. *ECOOP 2002–Object-Oriented Programming*, pages 1789–1901, 2006.

[6] J. Deacon. Model-view-controller (mvc) architecture. *JOHN DEACON Computer Systems Development, Consulting & Training*, 2005.

[7] Alan E. Dillman. Selecting an sdlc methodology, 2003.

[8] 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.

[9] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. *Extreme programming examined*, pages 287–301, 2000.

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

[11] 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.

[12] Nayan B. Ruparelia. Software development lifecycle models. *SIGSOFT Softw. Eng. Notes*, 35(3):8–13, May 2010. ISSN 0163-5948. doi: 10.1145/1764810.1764814. URL http://doi.acm.org.zorac.aub.aau.dk/10.1145/1764810.1764814.

[13] Jason Snyder, Stephen H. Edwards, and Manuel A. Pérez-Quiñones. Lift: taking gui unit testing to new heights. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 643–648, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0500-6. doi: 10.1145/1953163.1953343. URL http://doi.acm.org.zorac.aub.aau.dk/10.1145/1953163.1953343.

[14] Yonglei Tao. Component- vs. application-level mvc architecture. In *Frontiers in Education, 2002. FIE 2002. 32nd Annual*, volume 1, pages T2G–7 – T2G–10 vol.1, 2002. doi: 10.1109/FIE.2002.1157950.