Automated Testing
of
Graphical User Interfaces

Davor Babić

March 5, 2013

Abstract

The aim of this paper is to evaluate tools used for testing of graphical user interfaces, in the context of a course (EDA260) at Lund University. It provides a brief overview over some of the existing tools for automated testing of Java-based graphical user interfaces, while focusing on one tool in particular – Abbot. It takes a look at some best-practices and the challenges that have occurred from using Abbot in an education-type setting.

Based on a small pilot study, we will try to evaluate whether the use of these tools should be more widespread and perhaps even taught out in future iterations of the course.

Keywords: tdd, gui, test-driven development, Abbot, Java


1 Introduction

The benefits of writing automated software tests and TDD are today well understood and their use is widespread in our industry[14]. Unfortunately, gui tests remain relatively unused in practice and remain an unexplored area of research[7].

The graphical user interface constitutes, on average, roughly 45-60% of an average codebase[7]. By utilising modern tools and writing extensive gui tests, it is possible to significantly increase robustness standards across an entire system[7].

A team of ten students has as part of the course EDA260, during a period of six weeks used tools for automated gui testing, in order to find out how well they work and what productivity gains can be found by using them.

2 Gui testing

Writing gui tests presents certain challenges that are not present in traditional tests. The first and obvious problem is that a gui inherently responds to user interaction in a different way than code does. To test a gui we need to be able to click, type and have the ability to perform a multitude of other actions.

The traditional solution to this problem has been to utilise the model-view-controller pattern, and create a thin view that contains as little logic (and things that can go wrong) as possible[15]. Or in other words, do everything you can not to have to test the gui. The problem with that approach is of course that it means that while it does test the underlying logic; the view (what the user sees) is still left untested. By using a tool for gui testing we are able to test not only the code that is responsible for handling underlying logic, but also the code that handles presentation.

Instead of leaving the presentation-focused code untested, a better approach is to utilise modern tools such as Abbot and FEST that allow you to simulate having a physical user sitting in front of a computer with a mouse and keyboard.

Due to the fact that tools for automated testing of user interfaces are rarely employed today in the field of software engineering[9], the number of free and open source tools that are available and actively maintained for this task is quite limited. Of the ones that are available, a number of them haven’t been maintained in years. To further complicate matters, support is usually only available for one of the commonly used frameworks for creating gui-components in Java (such as Swing or SWT).
Generally, the tools can be divided into two categories. Those that offer simple record-and-playback functionality and programmatic tools[10].

Record-and-playback means that you have a script editor that lets you press a button that will record everything you do. It will record mouse clicks, key-presses and other actions, and allow you to play them back at a later time. The advantage of this approach is that the tests are very easy to write. The disadvantage is that they are hard to maintain since just adding a new button to an interface could potentially break the test[11].

The other category of tests are the programmatic tests. These can be compared to the more traditional JUnit tests. The main disadvantage that programmatic tests have is that they are hard to write[11]. To be able to write them, the developer will need a tool that will allow him to select components, interact with them and then verify the result in some way[11].

3 Abbot

Abbot is one of the more widely used open source tools that are available today. It has support for both functional- and record-and-playback tests. By default, it offers excellent integration with JUnit[16].

Because components are selected not based on where they happen to be located on the screen, but on a number of developer-selected properties, the resulting tests are less likely to break with new changes to the gui, compared to other tools. Ruiz et al describe Abbot-based tests as “robust because they use numerous attributes to dynamically identify a gui component without depending on any positional attributes.”[11] The following code snippet clearly demonstrates this.

```java
/* example Abbot unit test from the Abbot website */

// Suppose MyComponent has a text field and a button...
MyComponent comp = new MyComponent();
// Display a frame containing the given component
showFrame(comp);

JTextField textField = (JTextField)getFinder().
    find(new ClassMatcher(JTextField.class));
JButton button = (JButton)getFinder().find(new Matcher() {
    public boolean matches(Component c) {
        // Add as much information as needed to distinguish the
        // component
```
return c instanceof JButton &&
((JButton)c).getText().equals("OK");
}
}); JTextComponentTester tester = new JTextComponentTester();
tester.actionEnterText(textField, "This text is typed in the text field");
tester.actionClick(button);
// Perform some tests on the state of your UI or model
assertEquals("Wrong button tooltip",
"Click here to accept", button.getToolTipText());

Abbot is also shipped with the gui-based script editor Costello, which uses xml-files to further simplify the creation and maintenance of tests[16].

4 Alternatives

4.1 Comparison

The following section will provide a brief overview over some other tools for gui testing. It will also describe some of the more interesting ones in greater detail.

<table>
<thead>
<tr>
<th>Name</th>
<th>License</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abbot</td>
<td>Open Source</td>
<td>RP&amp;P</td>
</tr>
<tr>
<td>QF-TEST</td>
<td>Commercial</td>
<td>RP</td>
</tr>
<tr>
<td>IBM RFT</td>
<td>Commercial</td>
<td>RP</td>
</tr>
<tr>
<td>Marathon</td>
<td>Both*</td>
<td>RP</td>
</tr>
<tr>
<td>FEST</td>
<td>Commercial</td>
<td>P</td>
</tr>
<tr>
<td>UISpec4J</td>
<td>Open Source</td>
<td>P</td>
</tr>
<tr>
<td>TestNG</td>
<td>Open Source</td>
<td>P</td>
</tr>
<tr>
<td>SwingUNIT</td>
<td>Open Source</td>
<td>P</td>
</tr>
<tr>
<td>Jacareto</td>
<td>Open Source</td>
<td>RP</td>
</tr>
<tr>
<td>Jemmy</td>
<td>Open Source</td>
<td>P</td>
</tr>
</tbody>
</table>

RP = Record-and-playback
P = Programmatic
* = Jalian Systems provides both a commercial and an open source version of Marathon

4.2 Rational Functional Tester

The Rational Functional Tester is a commercial solution developed by IBM and has support for not only Java, but also Web 2.0, SAP, Siebel, Windows
Forms and terminal-based applications[17]. The Rational Functional Tester is mainly based on the record-and-playback approach[?].

Unfortunately, for a lot of use cases IBM’s solution is prohibitively expensive, costing between £2600 and £10000[17], depending on features and level of support. IBM offers demo and trial versions of the product on their website.

4.3 FEST (Fixtures for Easy Software Testing)

FEST is one of the more popular frameworks, and has seen a lot of use by large companies (such as Google, Oracle and IBM) in production. It is trivially easy to integrate FEST with JUnit and TestNG[18].

5 The pvg project

Abbot was chosen because it has been recommended in the scientific literature[11] and because it had support for both playback-and-record and programmatic tests. It was further thought that the gui-tool Costello that ships together with Abbot, would lower the amount of training required to get started with gui-testing.

Compared to several other toolkits, that haven’t been updated for some time, Abbot still seemed to be actively developed. The SourceForge project page reported that it had been updated in 2013 and the project website was last updated in 2011. This was another factor that worked in Abbot’s favor.

During a planning meeting in preparation for the second iteration, two team members were given the task of researching Abbot and Costello. They presented their results on the project’s wiki and then briefly during a quick stand-up meeting. Their wiki-article contained detailed instructions on how to get both Abbot and Costello running and was based on Abbot’s manual and various tutorials on the Abbot website.

The team presented both Abbot and Costello, but they decided against using Costello because they didn’t want to have to run yet another testing tool in addition to the standard JUnit tests, and Abbot could be more easily integrated into the existing JUnit-testing infrastructure and work process than Costello.

At this point, large parts of the graphical user interface had already been written, so what was left was merely the task of writing the tests for it. When the team tried to follow their own guide, they ran in to a lot of trouble. They tried going back to the guides and API documentation that is available on Abbot’s website. The official documentation proved both inaccurate and
inadequate and they were unable to get any tests working for a period of five hours. This led to large amounts of frustration, but they were eventually able to get the tests working.

6 Survey & interview results

During the six weeks that the project was ongoing, the students in Team 12 were regularly interviewed and asked to respond to some survey questions. The result of these will be posted here.

As can be seen from this chart, the team members had a very negative view of gui testing in the context of their own project.

When asked, almost all of them felt that they got very little out of writing the tests. The tests broke fairly often and a lot of time and effort had to be spent on fixing them. Sometimes they worked only intermittently and in those cases it was very hard to track down why exactly it is that they weren’t working. The team felt that they might as well have used that time to test the gui manually.

It was initially thought that these results might be entirely due to a lack
of training or familiarity with Abbot. This theory was later abandoned when it was found that it was the team members with the most experience and proficiency in working with Abbot, that were the most negative. The people that answered that their opinion of the tests were “neutral” were generally the people that hadn’t worked on them yet.

The gui tests became known for being difficult to work with, so people actively started to avoid them. This might explain why so many team members answered that they hadn’t worked on the gui tests. It is also possible that the early problems that the group had with getting the gui tests rolling, lead to a negative feedback loop.

Abbot’s FAQ page recommends that one should primarily use scripts, since those are easier to edit, lower complexity and make it easier to reference components. This recommendation was not followed during the project, where the team made extensive use of ComponentTextFixture instead. It is possible that they would have had a different view, had they followed this recommendation.

Most of the issues that the team was having with gui testing can be summarised with the following points:

The tool that they were using was difficult to use and had poor and at times outdated documentation. This increased frustration and made them feel that the cost-to-benefit ratio of using Abbot was very low. It is possible that a better tool would have changed their estimate of the costs involved
with writing gui tests.

We didn’t spend enough time training everyone on these tools. One of the benefits of pair programming is that it lets the programmers pick up knowledge from each other as they work[19]. Pair programming doesn’t seem to help when people don’t want to learn. In hindsight, one can conclude that we relied too heavily on pair programming as a teaching tool. The fact that the official documentation left a lot to be desired further aggravated the issue of lack of proficiency.

7 Best practices

During the project we found some best-practices that generally are a good idea to follow when writing gui tests. These will be presented in this section.

The first of these is to use unique names for components and make sure not to look up components by their text-values or position, as these are easily and often subject to change[3][11].

Another important thing to consider and always keep in mind is timing. The whole point of these tools is that they can click through a user interface much faster than a human; but that also means that the test might break if your interface takes too long to respond[3]. The PVG-group had issues with this on more than one occasion, with the result that test were passing and failing in an intermittent manner.

Ruiz et al. suggest that one should practice TDD when writing gui tests[11]. During the PVG-course, we found this to be difficult. One of the main benefits of practicing TDD is that you can use the test to guide design[2]. Since the MVC-pattern dominates the design landscape when it comes to graphical user interfaces[1], the team felt that when writing gui tests they had lost one of the main advantages of practicing TDD – the design part. This made the team feel that if one followed Ruiz’s second design tip (i.e. keeping a clear separation between the model and the view), that TDD was not as necessary as it is for the rest of the code base.

Additionally, it is a good idea to only focus on testing expected behaviour[11] and above all avoiding to write tests for what can be considered default component behaviour.

8 Conclusions

The conclusions we have drawn from performing this study is that automated GUI testing is inappropriate in the context of a PVG-project. The team
members themselves saw very little benefit from using Abbot and had trouble imagining situations where it might be useful.

It is possible that the advantages of automated GUI tests only become readily apparent for larger projects, where it is unfeasible for the developer’s to manually click through all aspects of the user interface in order to make sure that nothing is broken between every build. In order to properly test Windows 95, Microsoft had to release 400 000 beta copies[7]. Perhaps proper use automated gui testing practices would have been appropriate in that context?

Further studies, using different tools and providing more extensive training, are needed before these results can be seen as settled.

References


[18] https://code.google.com/p/fest/