Automated Graphical User Interface Testing

Markus Mikkolainen

29.10.2006

Abstract

This paper gives a short introduction to the opportunities and challenges of automating user interface testing, centering mostly on automation tools related to Java. The aim is to find a set of tools which is easily automatable with Ant , which enables us to effectively regression test a graphical user interface , which will require minimal work to maintain and a tool which will hopefully combine all of these traits.

Contents

1 Introduction

The current trend in user interfaces is geared towards graphical user interfaces (GUIs). This presents a problem for designing tests for software, since GUIs are very complex and hence GUI testing is very time consuming. Automation is a requirement for testing any larger graphical user interfaces, but automating GUI tests isn't a straightforward task. Techniques which are familiar from the command line interface (CLI) age, don't translate to the GUI world without problems. For example capture/playback techniques familiar from tools like Expect or Chat, are more difficult to implement in a graphical world. While text is easy to capture,compare and play back, it is more difficult to capture and compare graphics, where fonts and colors may change. Later an example of an action/expect based technique is demonstrated.

There has been some research into how this problem could be either solved or averted. A quick glance to ACM library search on the topic will give some approaches: generate the GUI based on a model, generate tests based on a model, automate test case generation to make it possible to regenerate the tests each time GUI changes and making automated oracles which model the behavior of the user interface. There have also been peeks into generating tests based on artificial intelligence (AI) planning techniques and genetic modelling.

In section 2 we will go through some of the challenges of testing software with graphical user interfaces. In section 3 we will discuss different approaches to automating GUI testing and in section 4 there are some examples of tools which implement some of these approaches. In section 5 there is a small summary and some conclusions.

2 Challenges

Testing a GUI seems to be easy at first. You don't have to write any code , you just click and see if it works. This naive approach works only for GUIs of limited complexity. Even quite simple software have surprisingly complex GUIs and the amount of work required to test the software will grow fast. For example WordPad contains 36 modal windows and 362 events , not counting shortcuts[MSP01]. It is also quite difficult to envision all ways in which a GUI can break, and sooner or later the testers will grow so accustomed to testing the same GUI that they will start to miss opportunities to make it fail.

Also to go by the principle of "test early – test often", an automatic regression test suite is required, otherwise the testing will not be done or simply costs too much.

2.1 Automation is difficult

It is quite difficult to automate something that you have only limited control of and limited knowledge of. In most cases when doing GUI testing a magical way of accessing program state is not available and manipulating the program has to be done with either operating system level tools or even purely graphically. Some tools turn this to their advantage, for example some of them use the X windows protocols which enable them to inject events to the application, and some use platform independent protocols like VNC to access the software under test. Fortunately in the Java case , we have two excellent tools – Java reflection to access program internal state and java.awt.Robot class to inject UI events to the program (and capture screenshots if necessary).

2.2 There is a lot to test

If testing by hand is expensive because there is a lot of test cases, it can also be expensive when automated – in case it is expensive to automate a test case or it is expensive to run the generated test case. The first case is possible if the test case generation has to be done each time the UI changes, and possibly by hand. The second case is possible if the automation tool generates the test cases automatically and possibly generates too many test cases. One of the challenges is ensuring proper test coverage.

2.3 How do we know if the program is in the correct state?

Sometimes (usually) it is impossible or atleast very difficult to know if the program is in a valid state just by looking at the user interface. The simplest example is a program which has just

experienced a failure not detectable in the user interface ,but the internal state is incorrect, eg. a thread has exited because of an exception.

2.4 How do we avoid breaking the tests when making the next iteration?

Changing an user interface , for example adding something, will often break the test cases. This is due to the hierarchical nature of the GUI itself. Even if the modification seems minor code and user-wise , it might break all testcases which depend on the modified parts. For example moving a button to a different place in a different panel, could easily break all test cases related to that button. The amount of broken cases can be very high, up to 74% per iteration [MS03]. Some tools like the one presented in [MS03] will attempt to repair these test cases, and some tools will attempt to avoid the problem by accessing the program under test in a way which gives enough information to recognize components by name[SJ04].

2.5 GUI testing and test driven development (TDD)

Test driven development is a principle which has popped up with agile processes, especially eXtreme Programming (XP). Without going into more detail, it requires the feature to be testable , before it will be programmed. This will give the programmer a good test coverage and continuous regression. TDD guarantees that only the bare minimum is done to meet the test cases, and that none of the features will break without it being noticed by the test cases. This is basically specification-by-testcase and it will concentrate all verification and validation difficulties in the testcase – specification interface.

While I will not say if this is good or not, TDD requires everything to be easily testable , or atleast testable , before it can be implemented. So to implement GUIs by TDD , GUI code should be easily testable via automated unit tests and the test should be writable before the code which implements the UI. This can present a few difficulties since this will require the programmer to build a test for a gui, which he has not seen. In some cases this is possible like with AJGUITE , which depends only on the names of the UI widget instances and classnames of the program to decide what to access. For some other methodologies like record and playback, this would require scripting the test in advance by hand.

2.6 Have we tested everything?

Test coverage problems have an effect on GUI testing and in some cases automation will make things worse. It might be difficult to answer the questions “Does this set of testcases test

everything that needs to be tested" and "do some of these testcases overlap". Especially so when test cases are automatically generated by the automation suite. Due to the special nature of GUIs , normal code coverage criteria don't fit here, so one has to have new ones specified to examine test coverage in GUIs. There are some definitions and guidelines in [MSP01] related to this.

2.7 Is automation cost-effective?

Automation requires time and there is no use automating things, if it is cheaper to do it by hand. To be effective automated testing must be relatively fast and easy to run , and must require as little work as possible to maintain. One special difficulty of testing is that GUI tests tend to be path-oriented , ie they tend to do multiple operations in a sequence - first to initialize to a state and then to exercise that state. This could easily explode the amount of test cases. The thing that must be considered carefully is how many tests , and which tests should be automated. There are some points raised in [RW06] about the cost-effectiveness of test automation in general.

3 Automation

Multiple avenues of approach have been used in the attempt to conquer the test automation problem. Here some of those approaches will be presented and discussed shortly.

Since maintaining GUI tests is expensive when doing regression testing, some have tried to automate the generation of tests. Other approach to the same problem is to model the test cases so that they can be automatically repaired when changes are done to the UI. Both of these solutions usually are based on the modelling of the UI as a set of events and states , which are then combined into "components" , which can be tested separately. The common feature is that these events sometimes have to be tested in sequence (like cut-paste) to be able to test all of the functionality. Naturally for autogenerated tests this means that the event sequences could grow to infinite length. Some limit has to be imposed and according to studies there is no point in testing very long event sequences except in special cases[MSP01]. What they found out is that event coverage (each event tested atleast once) guarantees very high statement coverage even at sequence length of 1. Lengthening the sequence makes the statement coverage approach 100% slowly.

3.1 Capture and replay (and script)

Capture and replay tools are related to the early CLI tools like Expect. They will use some method to capture a script and then replay the events. They will then either automatically compare the UI state to the captured state , or use some user-supplied method to determine that the testcase has completed successfully. The user-supplied method is usually scripting. In these tools it is usually possible to edit the test script by hand to update the testcase or refine it.

3.2 Test generation via specification

These tools require some form of a specification or model , which they use to generate the test cases and possibly a test engine to run the tests and determine if the program is in a valid state. One example is AJGUITE which requires a specification to be written and then compiled to a test engine. There have also been some studies into generating tests based on UML or other modelling languages, some of which are based on design by contract - like constructions or modelling languages. Some papers introduce ways of modelling the program as a planning-problem, where the UI is modelled as a set of states and an AI planner is used to find sequences to reach given states. This will form a set of test cases to exercise these states.

3.3 Other methods

There have been forays into generating tests using genetic algorithms. The idea is based on similar modelling as in the model based approach where the UI is modelled as states and events which transition between these states. A criterion is established for "good" testcases and they are allowed to come forth by using randomness and genetic selection based on this criterion. According to some papers this is quite successful in modelling novice user behavior and will find dead states and unexpected combinations of actions , which are often untested by other methods.

There is also the notion of "monkey testing" ie. instead of using thousand monkeys to write shakespeare , we will use one automated and very fast monkey to click through the UI in an attempt to generate a critical failure [RW06]. This is useful for smoketesting applications since it will find the worst bugs, the ones which you will not want to pass to the customer in a supposedly finished product.

4 Tools

Here I will present a some tools and discuss some of their pros and cons. I picked tools which were in production use somewhere and which are atleast somehow representative of the techniques listed in this paper.

4.1 Marathon - an example of record/playback/script

Marathon is based on running the program under marathon, recording a testcase by actually clicking on the gui and generating a test script out of this. After the test script is generated ,it is possible to run it and add assertions to assert the state after the test. The scripting language used is Jython , which is a java-based version of python capable of running within a JVM. While this tool is in the most popular record/playback category of tools it will probably not require as high maintenance as the more basic tools since it is able to access the components via reflection. It will also make transitioning from testing-by-hand to automated tests quite easy. However it requires some very basic programming skills from the testers to be able to assert the final state, unless it is quite plainly visible in the UI. Marathon is also capable of running in a batch mode and integrating to Ant. I was unable to get Marathon actually working on my system so all of this is based on second hand information.

4.2 AJGUITE - an example of specification to test engine using reflection and specification

AJGUITE is the tool developed on top of Jemmy in [SJ04]. The idea behind AJGUITE is to write a test specification by hand in an XMLish language. This language will then be compiled to code which uses the Jemmy library to access and manipulate the SWING GUI in the java program under test. It will perform a set of operations and assertions to check that the GUI is able to do what the specification says, for example that after a button is pressed it stays down or after a field is cleared it is really clear. This idea sounds very sensible and there are some benefits to this approach. While the test engine will have to run in the same JVM as the program under test , it will also have access to almost any data via reflection and it is also able to manipulate the objects themselves instead of just system level events and graphics.This means that there is a larger possibility that the test framework will interfere with the program ,but it gives a lot more information about the test subject. One of the larger benefits is that the test engine can access the objects by name instead of screen position. This means that if you move an object it will not break the tests immediately.

4.3 Abbot/Costello

Abbot is a framework for driving java UI components programmatically. Costello is a script editor and a launcher which accompanies Abbot. They are based on the same capture/playback/script principle that marathon uses , but Abbot can also be used programmatically. Abbot seems to be as aware of the inner workings of java and the application under test as Marathon is. The interface is a bit crude but I managed to get this one running.

5 Summary

While GUI test automation has come a long way from the “click until you find something” phase, it seems there is still a lot to be done. There are multiple good techniques which are quite easily adoptable but it seems that tools havent caught up with the developed techniques. Most if not all of the more finished tools I have seen seem to use the capture/replay - strategy and thus require quite a lot of work to maintain the test cases between iterations which change the user interface.

There are some tools that support other strategies , but they seem not to be ready for prime time. According to some of the papers listed in references[SJ04], most of the commercial tools do the capture/replay. While Jemmy (the library AJGUITE is based on) development has continued and future development directions were presented in the paper[SJ04], I didn’t see any continued development for AJGUITE.

References

[MS03] Memon, A. M. and Soffa, M. L., Regression testing of guis. *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA, 2003, ACM Press, pages 118–127.

[MSP01] Memon, A. M., Soffa, M. L. and Pollack, M. E., Coverage criteria for gui testing. *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA, 2001, ACM Press, pages 256–267.

[RW06] Ramler, R. and Wolfmaier, K., Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, New York, NY, USA, 2006, ACM Press, pages 85–91.

[SJ04] Sun, Y. and Jones, E. L., Specification-driven automated testing of gui-based java programs. *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, New York, NY, USA, 2004, ACM Press, pages 140–145.