Boon and Bane of GUI Test Automation

How to get the most from GUI Test Automation

by Mark Michaelis

Synopsis

In recent years the importance of well-designed User Interfaces has increased a lot. If nothing else, it is the rise of Web 2.0 applications, the applications for everyone. Nowadays User Interfaces have to deal much more than before with untrained people sitting in front of their computers.

So it is no wonder that not only the "automation behind the scenes" (Unit Testing for example) gained in importance, but also the automation of User Interface Tests with all its boon and bane.

This article describes the advantages and disadvantages of automatic GUI testing and tells you some lessons I have learned during my career as GUI tester. You will also get to know some selected tools I used for automated GUI Tests. A very personal insight into my work rather than a good comparison of all available GUI testing applications.

Bane

Let me start with the Bane of GUI Test Automation. This was the first contact I ever had with automatic GUI tests. The developers of the company I worked for before were well aware of the importance of GUI Tests, but they also knew of their problems. They used the Netbeans Jemmy Module.

Jemmy

Jemmy is no capture and replay tool, a quite common approach for GUI Testers nowadays. You develop your tests just as JUnit Tests in Java Source Code. This means you are actually blind when developing your tests. This has some disadvantages:

- There is no (implicit) additional testing by the author of the test.
- The author of the test has to know Java (in this case), although he does not need to have this knowledge from the perspective of his domain.
- Maintainability very much depends on the capabilities of the author. This is true as well for the identification of GUI components and for the actual error messages.

The result was obvious: Once developed, the tests were hard to maintain and it was rather hard to find anyone willing to dig into the code.

Nevertheless, experiences at this company paved my way into Quality Assurance. I changed to CoreMedia AG, Hamburg where I had the opportunity, no, the challenge to maintain GUI Tests for another Java Application, created with HP WinRunner, formerly known to be developed by Mercury Interactive.

WinRunner

Unlike Jemmy, WinRunner is a capture and replay tool. This means that while recording you actually see your application and you have an additional test through this for free. WinRunner does an important step: It separates the Map of the User Interface from the actual tests. This results in a database describing your GUI elements like this:

```
FileOpenWindow {
        class: window,
        label: Open
}

FileLoadButton {
        class: button,
        label: Load
}
```

Now the tests will refer to the GUI elements via this map. This has two advantages:

1. Test execution script and GUI Maps can be maintained separately. As a consequence: If only the layout changes, you only have to adapt the GUI Map at one specific location.
2. Depending on the keys (here: FileOpenWindow, FileLoadButton) you choose, your tests might be easier to read.

I will return to this example later. Because it is actually a boon, not a bane. The banes about the tests I adopted were:

- The GUI Maps were actually too rich. Think of a button which should be at coordinate 100,345 and has a label "Load". If one attribute does not match anymore, the test will fail. It would have been enough to just say that the button has the label "Load". That was enough to make the button unique.
- The test execution is described in a proprietary language called TSL (=Test Scripting Language), which in itself has some disadvantages:
    - No developer of the application (implicitly) knows about this special language.
    - GUI test developers again had to dive into a different non-GUI medium.
    - There is only little information about this language you can find e. g. in forums.
- The test scripts were not written to fail. In the case of an error, they did neither give clear statements of what failed or why, nor were they able to leave the application in a stable state so that other tests could continue.
- Tests depended on the results of other tests. This is actually no WinRunner problem, but I got aware of this while debugging the scripts. With one failed test at the root, all successors also failed of course.
- Tests were not rerunnable. Just think of

a read-only file to be created at a specific location. If you rerun the test, it will fail because the file already exists. While this is no problem during automatic test execution, it is a problem when debugging: Testers always have to reset their environment before a test can be rerun.
- Common actions (like "create document") occurred, step-by-step, at multiple locations in the scripts. As the creation steps in the application slightly changed, all of them had to be adapted
- WinRunner is for Microsoft Windows platforms. And most Microsoft Windows applications are not Java applications. This bears (at least) two other disadvantages:
  1. WinRunner cannot run on multiple platforms like MacOS, Linux, etc.
  2. WinRunner has severe problems, especially when it comes to supporting newer releases of Java.

Boon

Discussing the boon of any automatic test first starts at the ROI, the return-on-investment. Shura from Sun has written an interesting article about this topic titled "Automation Effectiveness Formula". Shura sets the investment to be equal with development time, which of course is not true in every way, but it is a good start. Shura points out that there are many variables to take into account when talking about effectiveness:
- The only alternative to automatic testing (if you want to test at all) is manual testing.
- Tests have to be run at least on every release.
- Tests have to run on all supported platforms.

Manual Testing

Shura already stated that the only alternative to automatic testing is manual testing. The scenario I know is that you have a couple of test plans which describe step by step what to do on the user interface. Testers replay these test plans and mark the different test steps as failed or success.
This should be done on all platforms the application supports, and it must be done on every release. This already rings a bell of warning, but let us first face the advantages of manual testing:
- It is obvious that the tester and the customer have the same view on the application. So it is a perfect acceptance test.
- Testers have no problems dealing with changing GUI layouts between versions. And if they have problems, their customers might stumble across them, too.
- Testers can do more explorative testing besides the actual test plan.
- GUI testing can be easily outsourced, as the testers do not need to know more than the customer.
- Manual GUI tests can be done by "Eating your own Dog Food": Use your product. This is the best and cheapest test you will ever get.

However, there are also disadvantages:
- Each release, each platform multiplies the costs.
- Test cases tend to explode easily. You have to manage carefully and use appropriate selection schemes to find the right tests.
- Manual testing between releases (nightly tests) is impractical. So you have no early reports on possible problems. Therefore, time has to be reserved for bug-fixing after the GUI tests are completed. In most cases, this will be short before the actual release date.
- Testers tend to become blind. If one and the same tester executes a test case multiple times, he will very likely start to skip test steps or at least not check the surrounding application not directly involved in the test path. This is especially true if he does the same just on different platforms.
- Manual tests are most likely to be skipped when the project runs out of time.
- The tests are not reproducible, as testers tend to forget the actual steps they have taken until an error occurred. Only recording the manual steps might help here.

Automation

My experiences with Automatic GUI Testing are more like a boon than a bane. Still, there are some WinRunner tests which need to be maintained (and they really break with each new release). They are the bane I am trying to get rid of. The boon started with a reboot from scratch.
This reboot was initiated by a new GUI testing application which was recommended to me. It is QF-Test developed by Quality First Software. This tool finally formed my expectations of a good GUI testing tool.

Expectations of the perfect GUI testing tool

1. Capture and Replay
   This means that you can press the record button, do some clicks and then replay these clicks by the tool. This is perfect to use in short-time projects with a short maintenance phase. It is inappropriate for product development, as your application will change over time and maintenance of such captured scripts is time-consuming. However, you can still use Capture and Replay to get to know the testing tool's language better.
2. Test Scripts saved as text
   This might also only be important for product development, and if a version control system is used. For binaries (such as commonly used Excel tables for data-driven testing) you will have no support to see differences.
3. Testing Language offers Try/Catch/Finally statements
   This is important so that you are able to react to exceptions which might break your tests. If you don't know such a statement:
   - "Try" will contain the code to be executed.
   - "Catch" will contain the code to handle exceptions, such as clicking away error dialogs.
   - "Finall" will contain the code which will always be executed irrespective of whether the try-code was successful or failed with an exception. Typically, some cleanup is achieved. For the CMS Editor tests I wrote, it closes for example documents I opened during the test.
4. GUI elements are recorded as mappings
   This is what I mentioned before: In the testing code, only references to the keys exist. Keys which point to the identifying descriptions of GUI elements.
5. Screenshots can be made
   Ideally they are automatically made in case of failed tests. However, it would be sufficient if you have the chance e. g. to make a screenshot in the catch statement. This really makes the first error analysis much easier.
6. Test scripts can be modelled in a GUI
   This is important because the GUI Testers can use their domain knowledge (of GUIs) to write the tests. And if you can choose your statements at the GUI, they are just easy to learn. All this is possible, because actually most steps for GUI Testing (about 90%) are quite simple and do not require a rich language. Another possibility is to use FitNesse, which allows a high abstraction level from the actual test execution steps. The tests themselves are modelled in a Wiki and are nearly as easy to read and understand as the customer-readable test cases.
7. There is a non-proprietary fallback language
   If there really should be a case where the language elements of the GUI testing tool are not sufficient. you need a fallback language. In case of QF-Test, it's Jython for example (a mix of Java and Python). These scripts should be easy to hide in the test scripts modelled in the GUI, so that only a small selected group of testers need to know about these scripts. Non-proprietary, because it is always good to be able to get help from communities on the web.

Of course, there are many more expectations, such as the quality of the generated reports, availability of a debugger or that tests can be started from the shell (for automatic nightly tests). But these seven checks I would take with me the next time I had to evaluate a GUI-Testing Tool. Needless to say that QF-Test passes all these checks with ease.

Restart from Scratch

The restart from scratch was important and easy. We had existing WinRunner tests which already covered some aspects of the application under test. This gave us time to concentrate on a new concept and on the learnings from previous GUI test banes.
- **GUI Maps:** All *needed* GUI elements got collected into extra files which could

be easily shared between all tests. Only the ones needed, this is important: Unused mapped GUI elements just blow up your files and once you need them it is most likely that they won't match any existing component anymore. And instead of relying on coordinates, component hierarchies etc. I gave every tested GUI element a name via setName(). A nice anecdote is that even a partner of CoreMedia got aware of this when he had to test extensions to the GUI. He was very pleased by this fact, told it to one of our developers who then told it to me. As you can see QA processes sometimes have surprising results.

- **Framework:** To get around duplicated code, I decided to build up a framework. The goal was to create a framework to get as fast as possible to the test case you want to execute. If you need some kind of document to write to, no matter what it is: the framework will offer a function for this. The result is quite obvious: Nowadays I can create many new test cases within minutes. Not much more time than a manual tester would need – for executing it **once** on **one** platform.
- **Facade:** For some checks the old and the new tests had to dig deeper beneath the GUI and access the API of the application directly. In the old tests this had not been made obvious to the application developers. And when developers changed the API, the GUI tests had to be adapted too. For the new tests I bundled this GUI Test API into a Facade. On refactorings with some IDE support, they will be automatically adapted to internal API changes. On the QF-Test side, I created a stub for this facade which made using the facade inside QF-Test rather transparent.
- **Fail-safe:** Especially for newly created regression tests, it was ensured that they failed with unfixed versions and that they left the application in a stable state.

Web-Test-Automation

I have to add this section, as I mentioned web testing as an important requirement for the future where web applications are increasing and even Office applications can suddenly be run within your browser.

At CoreMedia we are currently using Sahi which allows to write the tests in JavaScript, which of course has the benefit that you have full access to your web application. The problem of Sahi is that it is easy to write unmaintainable code, as there is no concept of GUI-Maps. But with some coding guidelines this can be enhanced. They include such things as to bundle access to GUI elements and keep them separate from the test scripts, and to write the top-level-code in a way which makes it easy for a human reader to see the relation to the user test cases.

So a test like this:

```
enterSearchTerm("Sahi");
submitSearch();
checkResultsContain("Sahi");
```

is easier to read than a test like this:

```
_setValue(_textbox("q"), "Sahi");
_click(_submit("Google Search"));
_assertNotNull(_link("Sahi"));
```

(taken from an example provided by Stringy-Low in the Sahi Forum).

A good alternative seems to be Selenium, which uses FitNesse at highest level to describe the tests. It was recommended to me quite often. And I am currently keenly following the further development of QF-Test, as they will also support web testing with the new 3.0 release.

Conclusion

Automation of GUI tests is a bane if you do not respect some rules, particularly with regard to maintainability. However, it is always a boon compared to manual GUI testing which simply costs too much money if the tests are executed regularly and perhaps even on multiple platforms.

My recommendation: Evaluate the application you want to use to write tests carefully, and carefully design your tests.

I hope I could give you some insight into the rules I found for myself, which you can find in the box "Mark's Rules for GUI Testing Automation".

Bibliography

- JavaWorld: Automate GUI tests for Swing applications by Ichiro Suzuki <http://www.javaworld.com/javaworld/jw-11-2004/jw-1115-swing.html>, 2004-11-15
- NetBeans: Jemmy Module <http://jemmy.netbeans.org/>; last release January 2006
- Automation Effectiveness Formula by Shura <http://jemmy.netbeans.org/AutomationEffectiveness.html>
- Fitness Acceptance Testing Framework <http://fitnesse.org/>
- QF-Test by Quality First Software <http://www.qfs.de/>
- Sahi Web Automation and Test Tool <http://sahi.co.in/>
- Selenium – Web application testing system <http://selenium.openqa.org/>

Mark's Rules for GUI Testing Automation

These are the rules derived from the boons and banes of this article. These are my very own rules – but you are free to adopt them.

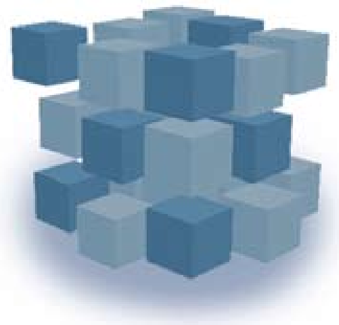
1. Write your tests so that they are readable from top-level like a use case. This makes it easier for others to understand your tests, to get an overview of the use cases covered, and of course it eases the debugging in case your tests signal a failure.
2. GUI-Maps: Never address GUI components directly. Always use some kind of mapping.
3. Do not be too specific in describing your GUI components: Less is (often) more.
4. GUI-Element-IDs: If your application's development language supports setting aids for components, use them and tell your developers to set them. For Java you should use setName() for example, for Web applications the ID attribute.
5. Write "Tests to Fail". Be aware that your tests will fail sometimes. And then they have to deal with it: Report the failure, end the test in the best possible way and pave the way for the other tests to follow. In short: Cleanup!
6. (Of course) do not make tests dependent on the results of other tests.
7. If possible, create tests which can be run multiple times without the need to reset your application.
8. Use Capture & Replay only in short-time projects or for getting to know the testing application.
9. Use Data Driven Testing whenever possible.
10. Don't store any automation control statements in binary objects such as Excel tables.
11. If you ever need to access the API of the application from the tests: Use a facade!
12. Build up a framework to cover common actions and to hide them from the actual testing code.

Biography

Mark Michaelis is ISTQB Certified Tester and Software Engineer Quality Assurance at CoreMedia AG, Hamburg since 2005. His main focus is on automation of GUI Tests and automatic setup of test environments.
After studying computer science at the University of Ulm and writing his diploma thesis on the automatic navigation of robots through reinforcement learning, he started as Java Developer at H.U.T GmbH in Hildesheim in 2000.
In 2003, he changed to Gentleware AG, Hamburg developing on Poseidon for UML (a UML modelling tool) and got deeper into testing automation through JUnit and Jemmy.

expecco

Taking the stress out of test automation!



eXept SOFTWARE AG

eXept Software AG, Tel. +49 7143 88304-0, info@exept.de, www.exept.de

Testing Services and Outsourcing

RSI Test Factory

Tradition:
15+ years of experience
500+ highly qualified professionals
3 million+ hours in QA projects

Test and Project Methodology:
International standards compliance
Web Portal for Project Portfolio Management
Customizable approaches:
business, requirements, risks...

Best Practices:
Highly reusable testware
Large scale production capacity
Charging models: FPA, UUCP, test case...
Competitive costs

www.rsinet.com.br