Automating Acceptance Tests for GUI Applications in an Extreme Programming Environment

Malte Finsterwalder
University of Hamburg
Bundesstrasse 74
20144 Hamburg
Germany
+49-40-41338955
finsterwalder@acm.org

ABSTRACT
Extreme Programming (XP) is focused on the fast and reliable delivery of high quality business value to the customer. In order to ensure that the expectations of the customer are met, the customer specifies acceptance test criteria. In XP, ideally, all tests should be automated, but it is not always worth automating every test. In particular the creation of automated acceptance tests for interactive graphical applications is far from trivial. I will make suggestions as to what to consider when automating acceptance tests. GUI Capture/Playback Test Tools try to cater for the above mentioned difficulties by providing facilities to exercise a GUI automatically and they include validation checks on the GUI level. In this paper, I will address problems with these tools and show how to avoid them. I will discuss ways of how the test tools can be incorporated into an XP project. Possible alternatives for test automation will also be described.

Keywords
Acceptance Test, Automated testing, GUI testing, Test Tools, Extreme Programming

1 INTRODUCTION

Acceptance Tests, being one of the core practices of XP, play a vital role in an XP project. They test whether the developed application delivers all the functionality the customer has requested. Acceptance tests need to be repeatable and should be run at least once a day. This way, any defects that have been introduced into the application can be detected and corrected as soon as possible. Tests which are performed manually, are not likely to be run this often. When the outcome of a test is manually verified by inspection, it could happen that errors are being overlooked. In order to prevent this from happening, the tests need to be automated as much as possible.

However, it is difficult to automate tests that involve GUI-intensive interactions. To test the application in its entirety, tests should actually exercise the GUI of the application and verify that the shown results are correct. In order to verify these results, different attributes of the shown GUI widgets, like color or text, must be examined. But automated control and inspection of GUI widgets is not easily possible. GUI test tools try to address this problem by providing mechanisms for capturing GUI interactions while a user exercises the application. The tools play the interactions back later and verify the outcome by inspecting GUI widgets or database entries.

However, the abilities of GUI test tools are limited. Especially when the GUI is changed, the tests must be adapted. It can be a great effort to automate some tests. Therefore the tool should be used with certain guidelines in mind, which are explained later.

This work is based on my experience with TeamTest by Rational and WinRunner by Mercury Interactive.

2 ACCEPTANCE TESTS

In the beginning of XP, acceptance tests were called functional tests (cf. [1]). Since the entire developed application is being tested by acceptance tests, the test literature also refers to them as system tests (cf. [2]). These are technical terms specifying what to test: The functionality of the whole system.

It is important to understand that in XP, acceptance testing is not just about finding out whether the application works as expected or not. In XP, the customer writes down small user stories to capture the requirements. For each user story the customer specifies acceptance tests as well. These tests are implemented and run frequently during the development process. By doing so, the customer gains confidence in the functionality of the developed application. The amount of successful tests is also used as a measure of progress. As the application is more and more completed during the course of the project, the number of successful tests should steadily increase.

At the end of an iteration a story is only accepted by the customer, if the acceptance tests for this user story demonstrate that the developed application is providing the requested functionality correctly. Thus the term “acceptance” tests. If some tests for a story fail, the story will have to be reworked in a later iteration.

3 TEST AUTOMATION

In XP, acceptance tests should be automated. But it is not always worth automating every test. As Brian Marick writes (cf. [9]):

Key issues in automated testing are:

- *deciding which tests are worth automating.*
- *finding and implementing an automation approach that insulates tests against change.*

Sometimes the extra effort put into automating a test does not pay off. This effort, the test's usefulness as well as its lifetime differ greatly from the technique that is used to implement the test. There are two major concerns when choosing a technique for automating a test:

- how expensive is it to implement the test with this technique?
- how will the test behave when the application changes and how likely is it to break?

Often the easiest and least expensive way is to implement the tests in the same language as the developed application itself. That way there is no initial cost for the developers for learning a new language or using a new test tool. It is easy to control the application and access all parts of it for verifications. When the interfaces that are used by the tests change, for example during refactoring (cf. [3]), the tests can easily be kept up to date by altering them with the interface they use. That way the tests are less likely to break.

The disadvantages are that not everything can easily be tested within the applications. Questions like "Is the right dialog box shown?" are not easily answered. The tests are written on the same level of abstraction as the application itself. There is no special support for testing purposes. All the tests need to be programmed by hand which can be a major effort and requires programming skills. A testing framework and auxiliary functions need to be implemented to facilitate the testing effort.

To guide the decision which tests are worth automating, Brian Marick writes up three criteria (cf. [8]):

1. *Automating this test and running it once will cost more than simply running it manually once. How much more?*
2. *An automated test has a finite lifetime, during which it must recoup that additional cost. Is this test likely to die sooner or later? What events are likely to end it?*
3. *During its lifetime, how likely is this test to find additional bugs (beyond whatever bugs it found the first time it ran)? How does this uncertain benefit balance against the cost of automation?*

Item 1 reminds one to think about the extra cost that is involved in automating a test. Only when you have a feel for how much extra effort you have to invest in the automation of a test, you can decide if this extra effort will pay for itself. Especially in the light of refactoring, as it is incorporated in XP, item 2 and 3 get a different emphasis. Since the code is evolving and being refactored all the time, the lifetime of an automated test is likely to be shorter. But at the same time, the tests will be refactored as well in order to cope with the changes of the underlying application, which is extending their lifetime again. It is an extra cost to keep the tests up to date during refactoring.

As the code is constantly refactored, it is quite possible that defects appear in areas of the software which worked properly before. The likelihood of an automated test to find a defect not just on the first run, but also on a consecutive run, is much higher. Thus regression tests are more important when refactoring is used.

4 GUI CAPTURE/PLAYBACK TEST TOOLS

GUI Capture/Playback Test Tools support application testing. They capture user interactions done to the application under test while they are being performed. The user interactions are recorded in scripts. The scripts can be played back repeatedly simulating the user's interactions captured in them.

Some of the benefits of using these test tools are:

realistic The test tools exercise the application in exactly the same way as a real user would do. The running test is said to be as realistic as possible.

capturing The tests can be created by capturing the normal usage of the application. Tests do not need to be programmed by hand.

data-driven test design The same tests can be run repeatedly with different values. The different values can be given e.g. in an Excel-spreadsheet.

comparison support The tools support the comparison of all kinds of attributes of GUI widgets, database entries and bitmaps with stored correct values.

large test suites Tests can be grouped into test suites which can be exercised together unattended.

synchronization The tests can be synchronized with the application by the use of wait conditions and timers.

scripting The test tools use flexible high level script languages with special support for accessing all kinds of GUI widgets.

bug tracking Bug tracking and reporting is included in the tools.

test planning The tools incorporate test planning and management.

On the other hand, the major drawback is that the tools exercise the application through the GUI. The GUI usually changes as time goes by. Changes in the GUI can break the tests. When the GUI of the application under test is changed significantly, the tests break down and need to be either fixed or completely reworked. Also, not all interactions can be

captured properly. Sometimes it is difficult for the tools to detect and to control certain GUI widgets, especially custom made widgets. The test tools can be extended to cope with the custom built GUI widgets, but this can be a lot of work. The attributes that can be inspected are limited as well. The standard comparison functions can only compare the attributes with previously stored values or a range of possible values. More complex comparisons have to be programmed by hand.

The normal way to create a test with a test tool would be to capture each test independently. This way, every test that needs to open a file includes a portion that captures the GUI interactions needed to open a file. This results in a lot of duplicate code. When the way to open a file is changed, for example because the menu layout is rearranged, all the tests need to be adapted to cope with the change in the application under test.

To mitigate this problem, it is helpful to use "framework-based test design" (cf. [6]). With this approach, a small framework specific to the application under test is constructed inside the test tools scripting language. For every feature that can be exercised in the application under test, a small function is constructed which is written in the test tools scripting language. When the function is called, the feature is exercised. Instead of including all the GUI interactions needed to open a file in each script, a simple function call can now be included. This way the tests stick to the "Once and Only Once"-rule (cf. [1]). When the application changes, only one place in the code needs to be adopted to cope with the change.

This implies that the test scripts are programmed by hand. Only the functions for the framework can be captured. But in most cases, the captured scripts need to be enhanced manually to create usable functions. The captured functions can be used as a starting point and are then refactored to what is needed. The benefit of easily capturing tests is lost.

5 HOW TO USE THE TEST TOOLS WITH XP

One of the XP mantras is to "do the simplest thing that could possibly work". Using the test tools may not be the simplest thing. They require quite some time getting used to. If tests are created before the application which is to be tested, and if "framework-base test design" is used, some of the benefits that the tools advertise are lost. For reasonable testing efforts, the naive capturing of user interaction as advertised by the tool vendors, as well as the mechanisms provided for verification of correct results are often not sufficient. Therefore manual programming of test scripts is often required.

It is said that acceptance tests should be specified by the customer and that it would be best if the customer developed the tests himself. The distributors of GUI Capture/Playback Test Tools claim that it is possible for an unexperienced person to use the test tools for the recording of simple tests. From my experience, this is not easily possible. The recording process is very fragile and requires a lot of planning and understanding: At least a GUI prototype is needed, extra functionality is not as necessary. It is desirable, however, to sit down with a customer and to capture some tests which are used as a starting point and are then refined. The customer tells the tester what interactions to do and what results he would expect. The interactions as well as the verifications are then captured.

Since the tests are rather fragile when the GUI of the application changes, it may become a lot of work to keep the tests synchronized with the GUI of the application. Therefore it is better not to build all the tests depending on the GUI of the application.

In a well-designed system, there should be a thin layer which does only the GUI handling. Apart from this, there is very little functionality in this layer. The underlying functionality is realized in different parts of the application. The Façade-Pattern (cf. [4]) can be used to wrap the underlying functionality. The GUI-layer accesses this functionality through the façade only. If this is done throughout the application, most of the functionality can be tested from within the application code by calling methods in the façade. This could be done e.g. by using one of the xUnit testing frameworks (cf. [10]). These tests cannot verify, of course, that the GUI itself works correctly. This is, where the test tools can support the testing effort. If the underlying functionality has already been tested extensively, it suffices to create tests for just a few features with the test tools, showing that the GUI is attached correctly to the underlying functionality. Additionally, some manual testing of the application could support this process before delivering it to the customers.

The creation of a thin layer for GUI handling can become difficult, especially for Web applications. Web applications generate HTML-output which is then loaded and interpreted by a Web browser. Often additional infrastructure, like Java Server Pages (JSP), is used to facilitate the generation of HTML-output. In such a setting, a major part of the underlying functionality can still be tested through a façade. However, the process of generating the HTML-output, which is partly done by the infrastructure, can not be tested this way, but only by sending an HTTP-request to the application. The returned HTML-output can then be compared with a saved HTML-file which is known to be correct. A problem with this approach is that HTML-files are often changed to adjust the design or to change included text and graphics, even though the underlying application does not change. In such a complex environment, it is difficult to test whether a Web page functions correctly, or not, without exercising the Web page inside an actual Web browser. This holds even more true when active components like JavaScript are embedded into the Web page. The test tools can control a real Web browser in order to support the testing of Web applications. Small changes in the design of Web pages are ignored by the tools.

For each user story, there need to be acceptance tests. These tests should be implemented no later than in the iteration where the user story is implemented. The acceptance tests are needed at the end of the iteration in order to verify that the user story is finished and fulfilling its requirements correctly. It is even better if the acceptance tests are implemented before or shortly after the part of the application that supports it. Then they can be used by the developer during development to confirm that he has finished implementing the user story. The tests can be used in the iteration as a measure of progress.

It is not possible to record a test when the underlying application does not yet support the feature which is to be tested. At least the GUI needs to exist. If the functionality behind the GUI does not yet exist, it takes extra effort to specify the expected results. The test tools are not optimized for this type of testing. They are optimized for testing an already performing application.

Despite these obstacles it is possible to create the tests almost at the same time as the application itself. When "framework-base test design" is used extensively, the test themselves can be programmed in advance. The tests are written using "programming by intention" (cf. [5]). While doing so, functions which are assumed to exist are being used.

If not many tests are automated with the test tool and the GUI is not likely to change often, it may be enough to just capture the user interactions necessary for the tests and not to invest much extra effort. If the tests break because the GUI of the application has changed, the tests are repeated with the new GUI and this is captured again. This keeps the initial investment into the creation of tests to a minimum, but increases the cost if the application under test changes. The additional cost increases with the number of tests created with the test tool.

If you do not want to invest in test tools, William Wake describes how GUI testing in Java can be done inside the application code using jUnit (cf. [5]). In his paper, he describes how to access the GUI widgets from within the Java-Application. This practice is transferable to other languages as well.

6 CONCLUSION

The focus of XP is to travel light and to do only things that help develop high quality software. Automated acceptance tests are one of the things that are helpful.

For most of my testing it was sufficient to code the tests in the same language as the application, calling the application code through a façade. This implies that the application is designed so that no business logic is mixed with the GUI handling code. The verification that the thin GUI layer is working well can often be done manually. If the tests which are to be performed manually are described in prose text, it can be assumed that the tests can be performed several times with little variation. Verifications of correct results may be automated with small test applications, so that oversights by the tester are minimized. This requires discipline by the tester. Especially for GUIs that change often it does not pay off to invest in GUI test automation.

If GUI test tools are used, they should be used sparingly. I would automate only the most important tests which are to be run often in order to see whether the basic functionality of the application is working correctly. In my opinion, these automated tests should be supported by a suite of tests which are performed manually on a regular basis, for example twice per iteration. Since manual inspection is cumbersome and error prone, small check programs that do the verifications should be created.

Only what is supported by the tool should be automated. If the test tools fail to handle certain GUI widgets, time to extend the tools should not be invested, unless the extension is trivial or it is essential to have automated tests for this part of the application. If the verification mechanisms do not suffice, I would not use the tools.

REFERENCES

[1] Kent Beck. *Extreme Programming Explained.* Addison-Wesley, 1999.

[2] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns and Tools.* Addison-Wesley, 1999.

[3] Martin Fowler. *Refactoring.* Addison-Wesley, 1999.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *Design Patterns* Addison-Wesley, 1995

[5] Ronald E. Jeffries, Ann Anderson, Chat Hendrickson. *Extreme Programming Installed.* Addison-Wesley, 2000.

[6] Cem Kaner. *Improving the Maintainability of Automated Test Suites.* Paper Presented at Quality Week 1997. <http://www.kaner.com/lawst1.htm>

[7] Malte Finsterwalder *Studywork: Erstellen und durchführen von Anwendungstests mit GUI Capture/Playback Testtools.* <http://www.bigfoot.com/ kroeger/publications.html>

[8] Brian Marick. *When Should a Test Be Automated?* Paper presented at Quality Week 1998. <http://www.testing.com/writings/automate.pdf>

[9] Brian Marick. *A Survey and Discussion of Automated Testing.* <http://www.testingcraft.com/automated-testing-survey.html>

[10] The xUnit Testing Frameworks can be found at: http://www.XProgramming.com/software.htm