The Video Store Revisited Yet Again : Adventures in GUI Acceptance Testing

Johan Andersson and Geoff Bache

Carmen Systems AB, Odinsgatan 9, SE-41103 Göteborg, Sweden
geoff.bache@carmensystems.com

Abstract. Acceptance testing for graphical user interfaces has long been recognised as a hard problem. At the same time, a full suite of acceptance tests written by the Onsite Customer has been a key principle of XP since it began [1]. It seems, however, that practice has lagged behind theory, with many practitioners still reporting weak or no acceptance testing. At XP2003, we presented our successes with text-based acceptance testing of a batch program[2]. In the past year we have extended this approach to apply to a user interface. We have developed an approach based on simulation of user actions via a record/replay layer between the application and the GUI library, generating a high-level script that functions as a use-case scenario, and using our text-based approach for verification of correctness. We believe this is an approach to GUI acceptance testing which is both customer- and developer-friendly.

1 XP acceptance testing

We should be clear what we regard as the primary aims of acceptance tests. These are the standards by which we judge acceptance tests and approaches to acceptance testing:

- The tests should model the actions of a user as closely as possible.
- Writing the tests should be quick, painless and require as few programming skills as possible,
- Running the tests should be as smoothe as possible - press a button and watch them go green/red.
- Maintaining the tests should not be too laborious.
- Tests should be as stable under changes as possible. In particular they should be independent of things like font, user interface layout and internal design.
- Tests should document the features of the system in as readable a way as possible.

Let's also be clear at what we are not aiming for. While the following are worthy aims, they are mainly the responsibility of other practices, for example Unit Testing or the various replacements for it that we described in last year's paper [2].

- The tests should not aim to improve or document the design.
- The tests should concentrate on indicating the presence of errors, not primarily help in fixing them.

2 Introduction

Our open source acceptance testing tool, TextTest [3], has traditionally been a console application that we have used to test UNIX batch tools. Recently, however, we wrote a GUI for it, and wanted to be able to test this GUI using a variation of the same approach. We have come up with an approach to do this that we found to be highly effective. For the sake of this paper, however, we thought that we would use what we have learned to revisit the classic Video Store problem, as this is likely to be more familiar to readers and avoids the meta-situation of programs testing themselves! The Video Store has been used to illustrate a few aspects of XP already, from refactoring to unit testing.[4]

TextTest is written in Python, and its GUI uses the PyGTK library[5]. The examples are therefore taken from this environment.

3 The Theory: Principles of our Approach

3.1 Separating Simulation from Verification

Acceptance testing of GUIs has traditionally been regarded as one activity. Perhaps due to our background with applications that do not have an interactive aspect, we have come to regard it as two, largely independent activities: simulating the interactive actions of a user in some persistent way (e.g. a test script) and verifying that the behaviour is correct when performing these actions. For future reference we refer to these as *simulation* and *verification*.

This simplifies matters somewhat because it removes the need for a tool that does both, decoupling the activities. Each tool can then concentrate on being good at one thing only. Armed with a pre-existing verification tool, TextTest[3], (discussed later) which has proved successful in the world of batch applications, the main challenge of testing a GUI is to find an effective approach to simulation.

3.2 An Agile Record/Replay approach

Record/Replay approaches have a strong theoretical appeal to us. To be able to create tests as a user simply by clicking around the application under test seems to be the easiest imaginable interface. Many tests can be created quickly, it is totally clear to the person creating them what they represent, no (potentially error-prone) code needs to be written per test and the only qualification for writing them is understanding the system under test, which is needed anyway.

Record/Replay tools are nothing new. A wide range of them exist, of which QCReplay[6] is the one we have most experience of. In recent years, a bewildering array of open source varieties for Java have appeared as well[7]. They are generally based on intercepting mouse clicks and keyboard input, recording them in a script, and asserting behaviour by taking screen dumps ("photographing" the screen)

They are not renowned for their popularity in the Agile community, however. They tend to produce long, low-level scripts which are extremely tied to the environment at the time when they were recorded.[8] For example:

1. Taking screen dumps is fragile under changes of font settings or window manager.
2. Moving the mouse across a GUI generates lots of focus-in events, focus-out events, mouse-over events etc. The application is only connected to ('*listening for*') a fraction of these, so they fill up the script with junk.
3. Even relevant events are recorded in very low level terms, with commands like `click(124, 21)`. Change the GUI layout and all bets are off: everything must be re-recorded.

In short, they do not embrace change. They are fun for a while but usually a maintenance headache in the long run.

For this reason, they have been abandoned by many in favour of data-driven approaches, that sacrifice some of the advantages listed initially for the ultimately greater gain of maintainibility in a changing world. We, however, have tried to rehabilitate record/replay in a more agile and maintainable form. In our view this requires a radical change to the way it works.

3.3 Test Scripts as Use-Case Scenarios

We believe that the fundamental difference between acceptance tests and unit tests is that acceptance tests have a customer focus and unit tests have a developer focus. A GUI acceptance test therefore has a lot in common with a Use-Case scenario. It should be a description of an interaction sequence between the user (actor) and the system under test from the point of view of the user. It should not describe what happens internally in the system, instead, as a Use-Case scenario, it should aim to give a user-readable statement of what happens during the actor/system interaction in the high-level language of the domain.

Such a test has two major advantages over the kind of test generated by traditional record/replay approaches. It is easy to read and functions well as documentation of system behaviour. More importantly, it is much more independent of the mechanism by which the use-case has been implemented.

We therefore want the test script that we will record and replay to be a high-level natural language file describing what the user does in the terminology of the domain. This fits well with the chosen verification approach, of comparing high-level natural language files produced by the system.

How can this be done? It is clear that it is not possible to write such a record/replay tool that sits on top of the application, starting, stopping it and recording its events at the system level. We need a layer between the application and the GUI library which can be told something about the terminology of the domain and the intent of the application rather than its mechanics.

4 Applying the theory: Simulation with PyUseCase

We have developed an open source record/replay layer for PyGTK, "PyUseCase"[9], extending it as we have needed to, in the process of testing TextTest

in the past year. While this scripting engine will only be useful to other PyGTK developers, the approach is possible with any GUI library.

To summarise, it differs from other record/replay tools in the following respects:

1. It does not generate scripts in any particular 'language'. What comes out is a high-level use-case description in the terminology of the domain.
2. The relationship between it and the application is reversed. Instead of sitting on top of the application and starting and stopping it, it sits between the application and the GUI library.
3. It is assertion-free, i.e. it is a pure simulation tool. Another tool (e.g. TextTest) is needed for verification.

4.1 Creating a domain-language script

(Note that PyGTK's terminology of 'connecting to signals' may be understood better as 'listening for events' for readers used to other GUI libraries)

Our ideal is to be as close as possible to the terms in the user's domain, and not use the terms of the GUI layout or the mechanics of how it is used. For example, when the user of VideoStore clicks the 'add movie' button, we want the script to simply say

`add movie`

rather than

`click('add movie')` or

`click(124, 21)`

This has obvious advantages. It's about as stable under changes as is possible : it survives as long as the user can in some way add a pre-selected movie at that point. It is not dependent on the user interface layout, the choice of widgets for the purpose of adding movies or the internal system design. It also leaves the reader in little doubt as to what happens at this point.

How does it work? We need our developers to connect the GUI widget signals to script commands at the same time they connect them to the methods that will be called when the user performs some action. This tells the script engine how to record the emission of that signal into a use-case description, and how to convert it back again.

For example, PyGTK programmers might implement the 'add movie' button like this:

`button.connect('clicked', addMovie)`

where `addMovie` is the method that actually performs the change, and `button` is the widget. To use PyUseCase, they would instead write

`scriptEngine.connect('add movie', button, 'clicked', addMovie)`

Instead of connecting the addMovie method directly to the signal emitted when the button is clicked, they connect it indirectly via the script engine, giving the user action a name at the same time. This is not much of an extra burden for the programmers. They just need to give names to everything the user can do via the GUI by adding extra arguments to their 'connect' statements.

This is basically the only API to PyUseCase. The syntax varies a bit for different widgets, and for more complex widgets like list views you need to tell it how to parse the arguments for selecting rows, etc. You also need your application to know about record and replay mode, so it can forward these things to PyUseCase.

Note that we only tell the script engine about signals we are connected to anyway. This means that any signals we aren't connected to won't be recorded by the script, whatever the user does with his mouse.

5 Verification with TextTest

TextTest and its usage were discussed in some detail in last year's paper[2]. The basic idea is the same, though it has gained many more features and users since then, including a GUI.

Essentially, the developers ensure that the system writes a plain-text *'behaviour file'* describing what it is doing. This file will contain all information considered useful to the customer: internal state, parsing and response to user actions, text that has appeared on the screen. Verification is achieved by the customer saving this file (and any other text-convertible generated files considered relevant) at the point he is happy both with what he is able to do with the system and how the system responds to his actions. Note that this is not a 'system diagnostic' file and should be free of statements that only have meaning to developers. Developer-statements should be written to a different file, which can also be generated and saved, but whose contents will not be viewed by the customer. By convention the 'behaviour file' is simply written to standard output.

A test-run then consists of replaying what the customer did and checking the system's text output for any differences from when the customer approved it. Differences will show up as test failure, though they may be saved and turned into the new correct behaviour if the customer approves the change. In conjunction with a simulation tool, this can be used on a GUI just as easily as on a batch application.

This has several advantages over requiring the customer to select assertions to make per test. In essence, many more verifications can be made, at a level of detail largely determined by the developers, who have a better overview of this. The customer has one less thing to worry about, and cannot "forget" to make some vital assertion. He can concentrate on using the system in an appropriate way and looking out for correct responses from it.

The tests consist only of automatically generated plain text. This removes the need to write any code per test. Your tests then depend on your program's behaviour file, but not on its internal design. This means refactoring will not disturb the acceptance tests, and you will not end up needing to maintain a lot of test code once you have a lot of tests. Bugs in your test code will not be hiding bugs in your real code.

Also, a customer without development skills can interact with the behaviour file, even if he isn't writing it. It is written in natural language and describes

in words what is happening. He can spot if the important number he saw on the screen didn't appear in the behaviour file, for example. If the verification is implemented as a load of Java test code, he can only hope it does what he intended when writing the test.

6 Customer-developer interaction

We have developed a test-first approach to using these tools. This requires close interaction between the customer and the implementing developers. The process looks something like this. (See the appendix for examples of it in action!)

1. The customer does the simulation to record the test. He does as much as he is able of what he wants to be able to do, generating a use-case script and a behaviour file that records system responses.
2. The customer can force-fail the test by editing the use-case script (giving the system some 'command' it does not yet understand). This tells the developers to add some user capability.
3. The customer can also force-fail the test by editing the behaviour file, if the system responded incorrectly or incompletely. This tells the developers to change the behaviour.
4. The developers take this test and implement the functionality, taking care to make the system output descriptions of important new system actions to the behaviour file.
5. The customer repeats the simulation with the new improved system (if needed). When he is happy, the new test is added to the suite of acceptance tests.

In this way development can be considered to be 'driven' by acceptance tests, in that tests describing work to be done are provided by the customer before that work is begun by developers. However, we have found this process most practical for small incremental user stories, which are hopefully the daily stuff of XP projects. Where the user wants completely new screens or totally different behaviour, it's more practical to describe this in words to developers and only try to create acceptance tests when some attempt has been made to provide the functionality. This is also likely to be the case in the very early stages of a project when there is not so much around to write tests on yet. It is still possible to use the approach for larger steps: but it requires a bit more of the test writer and is more prone to tests needing to be re-written when the functionality is present.

With this process in place, we have also experienced less of a need for unit tests. See our XP2003 paper[2] for details.

7 Other benefits of the record/replay layer

The fact that our record/replay tool sits between the application and the GUI library means it is a part of the application, rather than an optional extra for the testers. This opens up some interesting possibilities for using it for other things than directly recording and replaying tests.

7.1 Refactoring the tests

Everything possible has been done to keep the scripts short, high-level, and change-resilient, staving off the evil day when they get too hard to manage easily by pure record/replay. But applications get big and complex, and maybe that day will come anyway. As we don't have a language with syntax, we cannot take the approach of refactoring out common code by hand. We need some other way of updating a large number of tests when their use-case scripts prove to be insufficiently resilient.

Fortunately, we have the possibility to run in record and replay mode simultaneously. This enables us to automatically update a great deal of tests very quickly by telling the script engine to keep the old names for 'replay' only, while introducing the new ones for 'record'. This will work well where use-case actions disappear or change description. It works less well when new use-case actions need to be introduced to a lot of pre-existing tests, or when one conceptual 'use-case action' starts to require several clicks. This requires another approach, which we have called "GUI Shortcuts".

7.2 GUI Shortcuts: Towards a Dynamic User Interface

The record/replay layer is available at any time to any user of the system. This raises the possibility that individual users can personally tweak the user interface and eliminate repetitive actions by making use of the record/replay capabilities.

Most people have at one time or another ended up using a GUI in a repetitive way. They generally do not need all of its capabilities, and may have to make 5 or so clicks just to reach the screen they usually work with. Or for example, who hasn't at some time or other been frustrated by constant pop-up dialogues that demand "Are you sure you want to do this?" or something similar. This can be minimised by good user interface design, but fundamentally applications have to be configurable for their power users, and this can make them unwieldy for their novice users.

The user can simply record a "shortcut" for his repetitive actions. He goes into record mode at the appropriate point, records his usual five clicks (or OKs all his annoying pop-ups), and then gives the script he has recorded a name. A new button appears at the bottom of his screen, which he can click whenever he wishes to repeat what he did. This will save him time and repetitive work.

In the case of maintaining scripts when a user action starts to require more than one click, you can rely on the fact that shortcut names are recorded in scripts if they are available. Therefore, you would record a shortcut for the single click in the old system, run in record and replay mode simultaneously as described previously to insert the shortcut into all tests, and then simply re-record the shortcut (by hand) in the new system.

8 Conclusion

We feel that true Acceptance testing of GUIs can best be achieved by trying to make record/replay approaches more 'agile'. This in turn is best achieved by an

approach that separates simulating user actions from verifying system behaviour and uses co-operating, but separate tools for these things.

Simulation of user actions will be most change-resilient if it records use-case descriptions that are independent of the mechanics of the GUI, and this can only really be achieved by a record/replay layer between the application and GUI library, rather than one that sits on top of the application. PyUseCase is such a tool that works for PyGTK applications.

Verifying system behaviour is best done by a tool that compares automatically generated plain text. Organised plain text is easy to update and maintain and is independent of the system's internal design. TextTest is such a tool that will work for a program written in any language.

9 Appendix: Examples from the VideoStore

9.1 Step by step: fixing a bug in VideoStore

Let's suppose that the system allows the user to add two movies with the same name. This isn't good, so we as customer want to create a test for it. Here's what we would do.

1. Open TextTest's test creation functionality for the VideoStore application.
2. Enter 'DuplicateMovieBug' as test name, describe problem in description field. Create test.
3. Press 'Record Use-Case' button. TextTest will then start VideoStore in record mode, which forwards this mode to PyUseCase. We use the GUI to enter a movie 'Star Wars', add it twice, and then quit.
4. The test now contains a use-case script generated by PyUseCase. It looks like this:
   ```
   set new movie name to Star Wars
   add movie
   add movie
   quit
   ```
5. We now have the chance to edit this script, but it describes what we did and reproduced the bug, so we don't need to.
6. Press 'Run Test' button. TextTest now starts VideoStore in replay mode (using our generated script), and collects VideoStore's behaviour file. It looks like this:
   ```
   'set new movie name to' event created with arguments 'Star Wars'
   'add movie' event created
   Adding new movie 'Star Wars'. There are now 1 movies.
   'add movie' event created
   Adding new movie 'Star Wars'. There are now 2 movies.
   'quit' event created
   ```
   The 'event created' lines are created by PyUseCase when it successfully replays a script event. The 'Adding new movie' lines are simple output statements from VideoStore describing what it is doing.

7. We can now edit this as well. System behaviour was wrong, so we do so, replacing the second 'Adding new movie' line with a suitable error message.
8. Now we're done. The test is handed over to the developers, who can run it and will be given failure on the line we edited. They can then fix the problem, and get VideoStore to send the error message to both the behaviour file and the screen.

9.2 Step by Step : Adding new functionality to VideoStore

Let's suppose that we want to be able to sort the list of movies alphabetically. This functionality doesn't yet exist.

1. Open TextTest's test creation functionality for the VideoStore application.
2. Enter 'SortMovies' as test name, describe functionality in description field.
3. Press 'Record Use-Case' button as before. TextTest will then start VideoStore in record mode. We enter two movies 'Star Wars' and 'Die Hard'.
4. These are in the wrong order, so we want to sort them. But we can't do that yet. We quit.
5. TextTest then shows us the script it has generated. It looks like this:
```
set new movie name to Star Wars
add movie
set new movie name to Die Hard
add movie
quit
```
6. We now have the chance to edit this script. We wanted to do something we couldn't, so we add a line `sort movies` before the quit command.
7. Press 'Run Test'. TextTest now starts VideoStore in replay mode, using our script and collects the behaviour file. It looks like this:
```
'enter new movie name' event created with arguments 'Star Wars'
'add movie' event created
Adding new movie 'Star Wars'. There are now 1 movies.
'add movie' event created
Adding new movie 'Die Hard'. There are now 2 movies.
ERROR - 'sort movies' event not understood.
'quit' event created
```
8. We can now edit this as well. System behaviour was wrong, so we edit the file, replacing the 'ERROR' line with 'I'd like to press a sort button here. It should sort the movie list into alphabetical order' (or whatever, just to make a difference appear on this line when the test is run)
9. Now we're done. The test is handed over to the developers, who can run it once again and will be given failure on the line we edited. They can then add the sort button, and probably a little print-out to the behaviour file saying what order our beloved movies are in. When they swap order suitably, the developers return the test to the customer.
10. The customer can now review what happens. If he is happy that the system behaves correctly, and that both user and system actions are correctly recorded in their respective files, he saves the new behaviour and checks it in to the acceptance test suite. If not, the process iterates.

References

1. Beck, K.: Extreme Programming Explained. Addison-Wesley, 1999.
2. Andersson, J., Bache, G. and Sutton, P.: "XP with Acceptance-Test Driven Development: A Rewrite Project for a Resource Optimization System" in Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2003). Italy, 2003.
3. TextTest is open source and can be downloaded from http://sourceforge.net/projects/texttest
4. An entire chapter on writing a Video Store GUI with unit tests is present in Astels, D.: "Test-Driven Development: A Practical Guide" Prentice Hall, 2003 A discussion of refactoring with the same problem can be found in van Deursen, A. and Moonen, L.: "The Video Store Revisited - Thoughts on Refactoring and Testing" in Proceedings of the 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002). Italy, 2002.
5. PyGTK is available from http://www.daa.com.au/ james/pygtk/. It comes as standard with Red Hat Linux versions 8.0 and onwards.
6. http://www.centerline.com/productline/qcreplay/qcreplay.html
7. At least 6 record/replay tools for Java can be found at http://www.junit.org/news/extension/gui/index.htm
8. The tool 'Android' gives a beautiful example of the kind of low-level script you get from recording a test that does 1 + 2 = 3 in xcalc. http://www.wildopensource.com/larry-projects/article1.html
9. PyUseCase isn't formally released at time of writing, though it hopefully will be by the time of XP2004. It is in any case bundled with TextTest as TextTest itself uses it for its own testing.