Web Applications, Multithreading, Parallel Testing and Multiple Components: Further Adventures in Acceptance Testing

Johan Andersson, Geoff Bache, and Claes Verdoes

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

Abstract. At XP2004, we presented an 'agile record/replay' approach [1] to GUI Acceptance Testing based on recording high level use-cases rather than low-level GUI mechanics. This in turn was built on our textual comparison-based acceptance testing framework for batch applications, TextTest [5]. In the past year we have run a project to attempt to write tests using this approach for three different products produced by Carmen Systems, two of which are systems with Java Swing GUIs and one is a web application [2].
In the process we have produced a Swing equivalent of PyUseCase [6], presented last year. Also, we have met new challenges, listed in the title: the different challenges presented by a web GUI, multi-threaded GUIs and multiple-component systems. We have found that we can fit all of these things comfortably into our existing approach. Along the way, we have also reflected further on what has made our approach both workable and successful: a fully parallel test environment where speeding up the tests is primarily achieved by buying hardware.

Key words: Acceptance Testing, Use-case scenario, Text-based Testing, Web Application, Multithreading, Parallelism, Components

1 Our acceptance testing approach and tools

Our approach is presented more fully in our paper from XP2004 [1] and to some extent XP2003 [4] as well. This is a short summary of the ideas presented there.

1.1 TextTest: Verification by Textual Differences

We verify program correctness by the simple mechanism of comparing plain text produced by a program against a previously accepted version of that text - essentially comparing log files with a graphical difference tool like tkdiff. This is supported by an open source acceptance testing framework, TextTest [5].

We believe that this has some clear advantages over the more common approach of providing a test API into a program against which to write acceptance tests, and writing assertions about the program behaviour by hand in each test:

- Plain text is ultimately readable, ultimately portable and very easy to change [8].
- Writing new tests never involves writing test code. This vastly reduces the maintenance burden of a large test suite [9].
- The tests do not affect the design and cannot hinder its mobility.
- Adding a logging schema to a legacy system is easy and risk-free compared to 'making it testable' by refactoring it until a test API can be added.
- Every logged line is an assertion - so far more is being tested than if each test has assertions added by hand. Tests will find errors that their authors didn't think of.

Of course, the logging schema has to be consciously created and maintained, and should be high-enough level that the tests don't fail for trivial reasons all the time. It's also very useful if it can easily be configured on multiple levels of detail using a tool like those in the log4x family: this facilitates error-finding.

TextTest essentially assumes the system under test is a batch program that will perform a task without human intervention, and then exit.

1.2 xUseCase: GUI usage simulation with Use Case Recorders

When it comes to testing GUIs, we advocate an agile 'record/replay approach' based around automatically recording and replaying a high level 'domain language' script that models the use case that the test writer is performing with the GUI in question. This combines the strength of traditional record/replay approaches (rapid creation of tests, few skills required to create them) with the strength of data-driven approaches (easily tweaked high-level test representations). In short, the core assumption is that recording is great but re-recording is horrible.

This is achieved by providing a GUI library-specific layer that can listen for every event that the application listens for, and can be told by the application what the intent behind the event is in the language of the domain. This means it can record this high-level statement instead of something based on the screen layout or other GUI mechanics.

By recording and then being able to replay such a script, it is possible to make a GUI behave in 'batch mode', so that it too can be tested with TextTest.

Two such libraries currently exist: PyUseCase [6], for the Python library PyGTK [12] (presented last year) and JUseCase [6], for Java Swing, written this year. Both are written according to the 'You ain't gonna need it' [3] principle so neither will support every conceivable widget usage, but both will support the most common usage and are open source and hence extensible.

2 Use-case Recording for Web Applications

The GUI for a web application is presented in an external application: a web browser. This means that recording use cases presents new and different challenges. In particular, we need to decide at what level to record the user/GUI interaction.

2.1 Recording and simulating HTTP traffic

Although recording HTTP traffic is possible, it is not so good from a use case perspective. Firstly, the contents of HTTP requests and responses is typically very detailed, making them oversensitive to changes in the HTML code. Secondly, it is only possible to record transitions between different web pages, not GUI activity within a page, making the recorded material harder to interpret and control.

2.2 Simulating user/browser interaction

To obtain scripts at the use case level, we need to approach the application via the browser. In this domain a number of web application testing frameworks exist, e.g., actiWATE [21], Canoo WebTest [17], HttpUnit [18] and HtmlUnit [19]. They are all in some sense browser simulators, allowing the creation of web page objects that support clicking on links, filling out forms etc. They mostly expect that tests will be written using a provided Java API, although Canoo WebTest makes use of XML scripts instead.

2.3 A simple browser based on HtmlUnit

As we already had a use-case recorder for Java Swing (JUseCase), we decided to take one of the Java frameworks and build a simple web browser on top of it, plugging it into JUseCase at the same time. It would then be possible to record and replay use cases for the web application within the browser. Of the available options, HtmlUnit seemed to be the most browser-like framework that was also open source, so we decided to build on that and created 'SimpleBrowser' [7]. (see Figure 1)

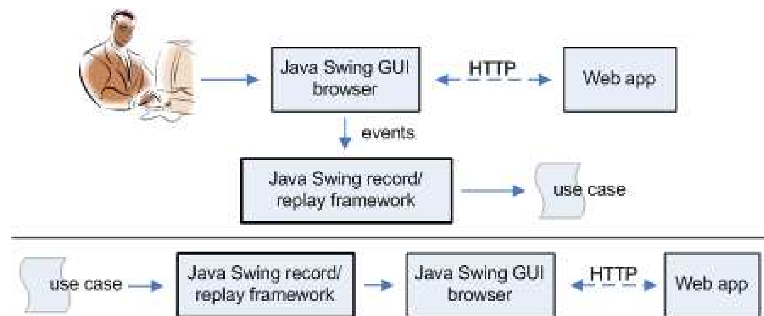

Fig. 1. Use case recording and simulation in a Java Swing browser

2.4 Getting use case-level command names using the 'title' attribute

Plugging SimpleBrowser into JUseCase was not difficult in itself — for each link or form component in the HTML an equivalent component in Swing could be connected up to JUseCase as discussed previously. However, we cannot really hard-code good names for these component events at the browser level, (names that say something about the intent of the application). Since the only connection between the browser and the web application is a number of HTML pages, that's where the use case command names have to come from.

Fortunately, most HTML tags support the 'title' attribute which can be used to define use case command names. This attribute is optional and 'offers advisory information about the element for which it is set' [20]. Using it to set use case command names thus doesn't conflict with its intended use.

The title attribute gives the application developer full control over which use case command names to use at which places, and since the attribute is available for both links and form controls, it provides a consistent mechanism for defining use case command names.

In regular browsers the title attribute is often used in tooltips for the corresponding elements, but this does not conflict with how they are to be used in conjunction with SimpleBrowser - in fact, use case commands often work very well as tooltips.

3 Testing Multi-Threaded Programs

A typical GUI application has the requirement of instant response to its user at all times. It cannot just lock up while some background processing (e.g. loading in a large amount of data) is done. This means that a large number of GUIs have multiple threads, and any acceptance testing approach for GUIs must therefore consider how to handle this problem.

3.1 Waiting during Simulation : Application Events

When we replay a test without human intervention, it may well be necessary to wait for things to happen before proceeding. Otherwise the test will fail because further use case actions rely on data loaded in a separate thread being present. In this case a traditional record/replay tool is basically stuck: it knows nothing of application intent and all it can do is ask the test writer to hand-insert 'sleep' statements into the script after recording it. Needless to say, this is both inefficient and error-prone: it can easily be forgotten, and the required length of sleep is hard to get right. If the test is being written by a customer unaware of the thread structure of the program, these problems are further exacerbated.

Our use-case recorders handle this situation by introducing the notion of an 'application event': the application can simply notify the use-case recorder when a significant event has occurred that is worth waiting for. At places in the code

where such events occur, the programmer adds calls to xUseCase, which will then record a 'wait for <name of application event>' command. During replay the replay thread will halt until the application reaches the point where the application event occurs, i.e., when the use-case recorder is notified of the event having occurred.

For example, assume we have the following use case script from a Swing App, using JUseCase:

```
load movie data into list
select movie Die Hard
```

Also assume that the first command starts a separate thread that loads a large amount of data from a database and displays it on the screen. Unless there is a way of telling the replayer when this has completed, it would perhaps try to select 'Die Hard' before that item was present in the list, causing the simulation to fail. To solve this, the programmer inserts the statement

```
ScriptEngine.instance().applicationEvent('data to be loaded');
```

in his application at the point directly after the loading is completed. The recorded use case will now look like this:

```
load movie data into list
wait for data to be loaded
select movie Die Hard
```

In record mode the applicationEvent method just records the 'wait for' command to the script file. In replay mode, the replayer halts replaying on reading this 'wait for' command, and the applicationEvent call then acts as a notifier to tell it to resume when the data has been loaded.

4 Parallelism: An Economic Way to Make Acceptance Tests Faster

One major obstacle to successful automation of acceptance testing is that realistic tests simply take too long to execute. This often leads to solutions where test stability is traded for speed, meaning a large burden of maintenance for the tests. We believe that, given reasonable economic resources, such speed problems are solved most effectively via parallelism. This opens up possibilities that simply are not there otherwise.

4.1 The Investment of Automated Acceptance Tests

The effort of producing unit tests is reasonably constant over time. Each test consists solely of code written in the same language as the system under test and should depend on a reasonably limited part of the system.

Producing acceptance tests is inherently a larger undertaking. For a start, much more of the system is being exercised, so they are generally larger-scale

and depend on more factors. Even more importantly, they should be understood (and preferably written) by the customer which means that levels of abstraction above target-language code should be involved. These might be tables in FiT [13], domain-language scripts in Exactor [14] or the recorded use-cases from PyUseCase/JUseCase. [6]

This means that the long-term economics of Acceptance Tests are somewhat different to unit tests. It is very important to save effort by looking at the tests as a group and producing solutions which apply to all conceivable tests, otherwise the effort of maintaining such a test suite may simply not be worth it in the long run.

4.2 Testing with a single CPU

At many companies, particularly those primarily developing on a Windows platform, the developers typically have their own PC under their desk. This means that testing their code before check-in involves using their own development resources and constrains very heavily how long they are prepared to spend running tests. It is all very well to lock up your personal CPU for a few seconds to run some lightning-fast unit tests, but to run acceptance-style tests sequentially for (say) 10 minutes with any regularity is out of the question. This situation constrains process advice: it becomes accepted that acceptance tests do not need to be run before checkin: maybe they are only run nightly, or once per iteration.

Acceptance tests verify that the system as a whole does what the customer wants it to do. This is an extremely valuable thing to get feedback on as often as possible. The principle of 'Test Early, Test Often' [3] (which we would extend to 'as early as possible and as often as possible') should apply at least as much to the tests that the customer owns and cares about. It is much more time-consuming and frustrating to piece together errors after the fact than to catch them before they are released. Also, activities that cease to be part of a daily rhythm become easily forgotten or neglected in practice, especially when under deadline pressure.

4.3 Mocking out time-consuming things

In order to speed up acceptance tests, it is also possible to follow the unit-test habit of creating mock versions of subsystems that take too much time to execute. This is clearly a powerful technique and is needed to some extent, but should not be used too heavily when created acceptance tests. The more you mock out, the more danger you run of creating 'testing holes' where your test system differs from the real system. Also, mocking things effectively can be a considerable effort, both to create the mocks correctly and to maintain them indefinitely.

4.4 Getting the tests to share resources

Another typical solution is to create dependencies between the tests: a common example is to run all tests against a common database and for each test to take

the output of the previous one as its input, with some sort of reset script run at the end. This is usually a bad idea in the long run. Diagnosing failures can be very complicated if the data is wrong at some point, and the whole chain is hard to create correctly, hard to maintain and nearly impossible to understand if you didn't create it. Once there are more than a handful of tests this becomes a big maintenance burden.

4.5 CPU power: cheaper than test maintenance time

We want our acceptance tests to be independent of one another for maintenance reasons. So, it follows reasonably that we should be able to run them in parallel.

In order to do this, we need a central bank of machines which any developer can submit jobs to, and some load balancing and queueing software that can distribute and queue the jobs appropriately. Such software is widely available, for example the free open source Sun Grid Engine [10], and LSF from Platform Computing [11]. TextTest integrates with these products to enable running tests in parallel.

Then, the primary means of speeding up the tests consists of going to the computer store, buying a new CPU and adding it to the central bank. This is a fairly error-free and cheap process compared to employing developers to write and maintain mock subsystems, to figure out that test 46 managed to screw up test 89's resources, or to find a bug after a week's development has passed. As a simple exercise for developers, take your own salary and work out how many hours you can spend on test maintenance before it would have been cheaper for the company to buy a new CPU for parallel usage. If it's much, you're underpaid.

The biggest challenge at this point is usually organisational. Money spent buying computers is a good deal more visible to financial departments than developer time spent maintaining tests, and financial departments do not always realise the value of automated testing.

5 Component based architectures: testing in a distributed environment

Classic software is often of the monolithic type, where all of the application resides on a single computer in a single process. Our testing approach has up until now been designed for this kind of software. In a modern system the architecture is usually more distributed. The application is a system of independent components possibly residing on multiple computers across a network. Components are understood as 'binary units of independent production, acquisition, and deployment that interact to form a functioning system' [15]. In essence, by using components, a decomposed design continues all the way into deployment.

5.1 Testing a family of components

If we think about testing a system of components, two approaches come to mind:

1. Test the entire system using a global test setup and a centralised logging mechanism
2. Regress to the 'unit test' approach and test each component by its API

Although the second approach is viable, we would like to see if we can achieve the advantages of an acceptance test approach in a component-based application too. The first approach would achieve this but may have practical problems. It is not unusual to end up with a problem similar to the 'single developer on windows' with only one test resource. In this case the company might have only one or two physical resources to run the 'whole system' on.

5.2 Automatically created mock components

Another approach would be to introduce mock components. We focus our 'testing attention' to a single component of the system, create mockups of the rest of the components and then run these as if they were the real whole system. The drawback of this approach is of course the added effort of developing and maintaining all the mock components.

For a specific test case, a mock component must interact with the 'component under test' as if it was the 'real' component. So even if we cannot create a mock component automatically in general, it is possible to create it automatically for a given test case. The approach is to run the 'real' system when recording a test-case, with all the real components active. We then aim to let the test framework intercept and store all 'inter-component' interactions where the 'component under test' takes part. For each real component 'talking' to our test component we then automatically create a mock component that given the specific interaction in the test case will respond in exactly the same way as the real component did in the 'record run'. These mock components are then stored along with the test case.

5.3 Intercepting component interaction

Component interaction inherently varies from application to application. We do not propose to turn TextTest into something that could intercept any component interaction in any application. We do however aim at making it able to intercept some common component interaction protocols and utilize that ability for multiple applications. An application with higher demands can always rely on the framework property of TextTest and write an extension module specific to that application.

As we use the intercepted network traffic for playback in the automatically created mock components, there is a need for this traffic to be fairly deterministic and interpretable. As with our approach for GUI testing we assume we have a fair amount of control of what and how the components we want to test in our system are designed and implemented.

5.4 Limitation of our approach

There are some principle limitations:

- As with our GUI test approach we test what we develop and nothing else.
- A component-based testing approach does not cover 'whole system' integration tests.

and some current limitations as we have only begun exploring this:

- We only intercept inter-component communication across a network.
- We have only intercepted non-binary traffic, such as XML.

The network traffic interception approach (with its limitations, complexities and opportunities) is commonly used in VPN solutions, some applications in this context can be found in [16].

Note that we assume that if a binary component is loaded into a process it will be tested along with other components in that process. We have no goal that each component should be testable in isolation: our aim is to test as big a part of the system as practically possible.

6 Conclusion

The acceptance testing approach advocated has shown itself to be versatile and applicable well outside of the realm of the batch applications where it began. Use-case recording has proved itself for 'fat client' GUIs in both Java and Python. We believe we have now found a viable approach for testing web applications, multi-threaded applications and multiple-component systems. At the same time, we acknowledge our debt to the CPU power we have available through a fully parallel test setup, and we encourage anyone with one CPU per developer to consider alternatives.

References

1. Andersson, J. and Bache, G.: "The Video Store Revisited Yet Again: Adventures in GUI Acceptance Testing" in Proceedings of the 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2004). Germany, 2004.
2. See also the full report of this project, covering the first half of this paper in more detail Verdoes, C.: "Use case recording and simulation : Automating acceptance tests for GUI applications". Chalmers University of Technology, Sweden, 2005.
3. Beck, K.: Extreme Programming Explained. Addison-Wesley, 1999.
4. 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.
5. TextTest is free and open source. It can be downloaded from `http://sourceforge.net/projects/texttest`

6. Both PyUseCase and JUseCase are free and open source. They can be downloaded from `http://sourceforge.net/projects/pyusecase` and `http://sourceforge.net/projects/jusecase` respectively.
7. SimpleBrowser will be released as open source pretty soon...
8. The Power of Plain Text is discussed at length in chapter 14 of Hunt, A. and Thomas, D.: The Pragmatic Programmer - from Journeyman to Master. Addison-Wesley, 2000.
9. Brian Marick expands on the burdens of test maintenance somewhat, talking about a 'two-year itch' when maintenance of code-based tests becomes prohibitive. `http://c2.com/cgi/wiki?TwoYearItch`
10. Sun Grid Engine is free and open source. It can be downloaded from `http://gridengine.sunsource.net/`
11. LSF (not open source) from `http://www.platform.com`
12. PyGTK is freely available from http://www.daa.com.au/ james/pygtk/. It comes as standard with Red Hat Linux versions 8.0 and onwards.
13. Framework for Integrated Test at `http://fit.c2.com`
14. Exactor framework at `http://www.exoftware.com/xp_tools.htm`
15. Szyperski, Clemens: Component Software, Beyond Object-Oriented Programming. Addison-Wesley, 1998.
16. AppGate is a VPN product that also uses the technique of network traffic interception, albeit for other purposes than test. Referred to in paper below as 'Proxy services'. `http://www.appgate.com/knowledge_center/20_White_Papers_and_Other_Documentation/wpaper_40.pdf`
17. The Canoo WebTest tool for automated web application testing is open source and can be downloaded at `http://webtest.canoo.com/`
18. The HttpUnit Java library for automatic simulation and testing of web applications is open source and can be downloaded at `http://httpunit.sf.net/`
19. The HtmlUnit Java library for automatic simulation and testing of web applications is open source and can be downloaded at `http://htmlunit.sf.net/`
20. The HTML 4.01 specification, `http://www.w3.org/TR/html4/`
21. The actiWATE web application testing environment can be downloaded at `http://products.actimind.com/actiWATE/index.html`