Putting a Motor on the Canoo WebTest Acceptance Testing Framework

Jennitta Andrea

jennitta@agilecanada.com

Abstract. User acceptance testing is finally getting the attention and tool support it deserves. It is imperative that acceptance tests follow the best practices and embody the critical success factors that have been established over the years for automated unit testing. However, it is often challenging for acceptance tests to be repeatable, readable, and maintainable due to the nature of the tests and the tools currently available for automation. The key contributions this paper makes to the agile community are: first, it provides concrete examples of applying test automation patterns to user acceptance testing, and secondly it provides a description of various extensions to the WebTest acceptance testing framework that facilitate developing automated acceptance tests according to these established best practices.

Key Words: automated testing, Canoo WebTest, framework extension, test automation patterns, testing strategy, user acceptance testing

1   Introduction

Years of test first development using automated unit testing tools and frameworks, like Junit [1], have resulted in a collection of best practices that enable key critical success factors, namely: automated tests that are *repeatable*, *readable*, and *maintainable* [2]. Automated tests create the safety net that enables a team to confidently evolve a system incrementally, and to be assured that it is production-ready. A team must be able to efficiently develop tests that they trust, specifically, tests that are free from side effects, and that perform predictably every time they are executed. Because the tests encode system requirements, it is important that the tests are more readable than the system code itself. When there is as much test code as system code, the tests must be maintainable, so as to continually keep their benefits higher than their costs.

It is imperative that both acceptance tests and unit tests follow best practices and embody these critical success factors. It is often more challenging for acceptance tests to accomplish this because of the nature of the tests and the tools available for automation. Acceptance tests embody real business workflow scenarios, thus all input and

validation occurs as a result of interaction with the user interface, and execution through all of the layers of the application.

This paper describes a series of customizations made to the Canoo WebTest [4] acceptance-testing framework that were implemented during the course of an XP/Scrum project to develop automated acceptance tests that are repeatable, readable, and maintainable.

2. Acceptance Testing Strategy

Automated software testing is not simply about tools and techniques for automating tests. In fact, the choice of tool should be one of the last decisions made when developing a testing strategy because the tool should enable the overall strategy, rather than drive it. While this paper is ultimately about a tool, we are obliged to start with the strategy. Software projects must balance many different, often competing, concerns when developing a testing strategy. The testing strategy as a whole considers the significance of each type of testing, including: unit, acceptance, usability, performance, security, etc. Decisions made about the one type of testing are likely to have an impact on the choices made about another type. For example, a system with an extensive unit test suite will tend to focus the content of acceptance tests on workflow concerns rather than on detailed business rule validation.

The team builds their acceptance testing strategy by considering questions like the following: What is the budget for user acceptance testing? Is the purpose of the test to specify requirements, to assess quality, or both? Who is responsible for writing the test specification? Who executes the tests and assesses failure? How are the tests executed? How often? How are the automated tests kept in sync with the test specifications? What parts of the system does the automated test touch? What techniques should be used? What tools should be used? The answers to one question are often inter-dependent on other ones. Some answers may contradict each other, making the formation of the strategy all the more challenging.

The experience described in this paper is based on a particular project with an acceptance testing strategy driven by budgetary concerns. Specification, automation, and execution of the acceptance tests were allotted an average of twenty hours per week of a single developer's time. The remainder of the strategy, including tool selection, had to facilitate automating as many tests as possible in a short timeframe. Given the size and complexity of the system, this budget was not sufficient to automate acceptance tests for the entire system, so the developer and customer collaborated to define the smallest possible set of representative tests for the highest priority areas of the system. The customer did not have time to automate the tests themselves, but reviewed the automated tests in order to sign off on them. While automating acceptance tests was considered late in the project, the system was built unit test-first, ensuring acceptable test coverage of small-grained business rules. Thus the acceptance tests were required

to focus on overall workflow scenarios, using the user interface in ways that a real user would. After a limited exploration of various techniques and tools, including HttpUnit [6], FIT[5], and a custom framework to generate test code from an excel spreadsheet [10], the Canoo WebTest framework was the tool selected for this project.

3   Canoo WebTest

Canoo WebTest is a layer on top of HttpUnit, where the tests are specified using descriptive XML targets (like clickbutton, verifyitle, etc), and executed via Ant [7]. The key characteristics of WebTest that made it a good fit for this project, are: a scripted approach to testing against the user interface (as opposed to a record-playback approach [8]); a succinct, high level specification language; readable and detailed output reports; and the interpreted execution, which made test development very fluid. The output reports were probably the most important characteristic for this project; not only do they provide visual clues as to the success/failure of each individual step, they also include captured screen shots. This facilitates a slow motion replay of the test, which is used by the customer to verify the correctness of the test, and can be used to debug the test when necessary.

The development team immediately recognized issues related to weak tool support for XML development and cross-technology refactoring, especially when contrasted to the powerful capabilities they exploited when developing Java with Idea [9]. In addition, the noisy XML syntax within the specifications was deemed to hinder readability. This latter concern was not a showstopper because the customer is able to review the output reports instead of the XML test specification.

4   Customizing Canoo WebTest

The examples in this section are fragments from a real acceptance test for a system that is used by hospitals to archive inactive patient records (called volumes). A patient is identified by a chart number, and may have one or more volumes. A volume is essentially a file folder that contains treatment details for a patient. If a patient has not received treatment for five or more years, their volume is considered to be inactive. Before boxes of inactive volumes are physically sent to an off-site storage location, a user enters the data associated with each volume into the Volume Archiving System (VAS). The acceptance test takes the volume and box through their normal life cycle (volume created, added to box, box moved off site, volume removed from box). It also demonstrates the following business rules: (a) creating a new volume will create the associated chart if it doesn't exist, (b) a box may contain duplicate volume numbers, (c) chart numbers are not unique across sites.

This section displays fragments of a single acceptance test, not necessarily in the order they would appear in a complete test. The entire acceptance test, which is too large to

include in this paper, can be found at [11]. The WebTest extensions described here have been submitted to [4].

Critical Success Factor 1: Repeatability

The rhythm of test-first development is: write a test, watch it fail, and then make it pass by developing the missing system capability or fixing the incorrect behavior [3]. Once the test is complete, it should pass repeatedly, whether it executes by itself or within a test suite. When a completed test fails, we need to quickly and accurately pinpoint the cause: did the test uncover a bug in the system, or is the test faulty? We strive to eliminate time spent debugging tests, especially the nasty situations when side effects of one test cause a different test to fail (the interacting test smell [2]). Best practices for achieving repeatability focus on making tests independent. Ideally, a test should operate on its own unique data set, and should clean up after itself (the clean slate approach [2]).

The most obvious shortcoming of the WebTest framework is the inability to manage the test fixture data directly from within the test. By default, one must use mechanisms external to the test to setup the precondition data and remove it when the test has finished. It is generally possible to setup independent test data for a number of tests in advance, however, this approach creates the mystery guest test smell [2], and reduces the readability of the test.

Our solution was to develop various new framework components in Java that enable creating precondition data directly within the test and cleaning it up when the test ends[1]. Example 1 shows the use of the two new Ant tasks: **preconditions** and **cleanup**, which act as bookends to the body of a test. Other custom domain specific Ant tasks are contained within these wrappers, and are responsible for creating or

---

[1] For those familiar with the WebTest framework, the following is a high level summary of the changes made. Text in **bold** font are new classes/methods we introduced; text in *italic* font are existing framework classes/methods. **PreconditionWrapper** extends *StepContainer*, and contains a sequence of steps of type **PreconditionStep** (an extension to *TestStepSequence*). The project specific subclass of **PreconditionStep** overrides the method **doCommonPre-Setup** to perform all of the necessary one-time data base initialization tasks. It then executes each specific fixture setup step, which creates and populates one or more domain objects, and registers them as being ready for persistence. The final task is to override **doCommon-PostSetup** which causes the objects to be stored using a specific persistence mechanism. Following this same pattern, we added **TearDownWrapper** to the framework, which contains a sequence of steps of type **TearDownStep.** Modifications were made to the WebTest framework in order to accommodate these new classes. *TestStepSequence* recognizes the two new wrappers and process them appropriately. *TestSpecificationTask* keeps track of the teardown steps. A finally clause was added to the *Engine's doExecute* method to ensure the teardown steps are executed after a test failure. The final piece is to write custom fixture setup and cleanup steps for the business objects needed by the test. These steps create business objects, populate them and register them for persistence.

deleting a specific type of domain object. To keep the test readable and succinct, the test specification includes only the attributes of the domain objects that are necessary for understanding the tests. All other attributes are generated 'anonymously' within the Java implementation of the custom creation/cleanup Ant task.

```
1.  <preconditions>
2.      <site name="site1"/>
3.      <site name="site2"/>
4.      <volume name="v1c1s1" chart="chart1" site="site1" box="box1"/>
5.  </preconditions>
…….
6.  <cleanup>
7.      <deleteSite name="site1"/>
8.      <deleteSite name="site2"/>
9.      <deleteVolume name="v1c1s1"/>
10. </cleanup>
```

Example 1 Preconditions and Cleanup

Critical Success Factor 2: Readability

It is crucial that all tests are readable, as they are the definitive reference for the system requirements. It is *even more* imperative for acceptance tests to be readable, because the customer is responsible for signing off on them and must fully understand them. A number of best practices help improve readability, namely: write the test declaratively (focus on *what* not *how*); write the test succinctly (include only the details that are pertinent to understanding the test); and make the test unambiguous (ideally, two different people with a similar understanding of the business domain should understand the test in the same way).

WebTest's steps (e.g. clickbutton, verifytext, etc) and output reporting facilitate developing readable tests, especially compared to writing raw HttpUnit and only having the red/green bar for feedback. In practice, we found that because user acceptance tests capture multi-step workflow, they tend to be fairly long. Even when reviewing the WebTest output report, the reader quickly becomes lost in a forest of low-level tactical details related to using the user interface. They must consciously re-construct the intent of the sequence of steps in order to understand the big picture. A series of simple adjustments to the framework and the report formatter improved this situation greatly.

A new attribute, **description**, was added to the *testSpec* target (see Example 2). This free-form text attribute is intended to capture an overall summary of the test and is displayed at the beginning of the test output.

```
11. <testSpec name="basic volume and box lifecycle" description="This test takes
    the volume and box through their normal life cycle (volume created, added to
    box, box moved off site, volume removed from box). It also demonstrates the
    following business rules: (a) creating a new volume will create the associated
    chart if it doesn't exist, (b) a box may contain duplicate volume numbers, (c)
    chart numbers are not unique across sites."/>
```

Example 2 Description attribute

Another new step container, called **group**, is used to assemble related steps together under a higher-level description (see Example 3). The primary purpose of this container is to enable the steps to be visually grouped together in the output report, giving the reader the big picture; steps 13-20 must be performed in order to create a new volume in VAS.

```
12. <group stepid="create a volume">
13.     <clickbutton label="${mainMenu.button.addVolume}"/>
14.     <verifytitle text="${addVolume.title}*" regex="true"/>
15.     <setinputfield name="${addVolume.field.boxNum}" value="box1"/>
16.     <setinputfield name="${addVolume.field.chartNum}" value="chart1"/>
17.     <setinputfield name="${addVolume.field.volNum}" value="v1c1s2"/>
18.     <setinputfield name="${addVolume.field.date}" value="2003-11-13"/>
19.     <clickbutton name="${addVolume.button.save}"/>
20.     <verifytitle text="${addVolume.title}* regex="true"/>
21. </group>
```

Example 3 Group step

We also added a simple custom step to the framework that corresponds to the fail() assertion from junit, called **forceTestFailure.** This facilitates an active to-do list style of writing tests that ensures the system will remind us when a test is incomplete rather than comments in code or notes on scraps of paper.

Critical Success Factor 3: Maintainability

Test first development yields as much (or more) test code than system code, thus we have to be as concerned (or more) with the maintenance costs of test code as compared to system code. Refactoring is a common practice on agile projects, because the system continually evolves over time as new features are developed. Acceptance tests are modified to reflect changes to business rules and screen details. Maintenance costs can be reduced if the acceptance tests don't break when UI elements merely change position on the screen. While development tools greatly assist the maintenance effort through powerful refactoring features, developers remain responsible for making design decisions that enable system and test code to be modified efficiently. This section contains a series of refactorings that were performed on the test specification fragment shown in Example 3.

A user goal level use case [12], e.g. to create a patient volume, is achieved through a number of detailed interactions with the user interface (entering text into fields, clicking buttons, etc). The acceptance test suite contains multiple instances of the same user goal level use case, so a strategy for code reuse must be devised. The simplest possible thing to try initially was to use XML componentization within the test specification. The WebTest framework was extended with a new step, **storeVariable,** as a simple way to pass parameters to an XML component. Example 4 is the result of refactoring the original test specification fragment (Example 3) to reference a common XML component.

```
22. <group stepid="create a volume">
23.     <storeVariable name="${param.boxNum}" value="box1"/>
24.     <storeVariable name="${param.chartNum}" value="chart1"/>
25.     <storeVariable name="${param.volNum}" value="v1c1s2"/>
26.     <storeVariable name="${param.date}" value="2003-11-13"/>
27.     &createVolume;
28. </group>
```

Example 4: test spec with custom XML component reference

The body the original test specification fragment was moved into the createVolume XML component (see Example 5), with specific values replaced by references to the stored 'parameters'.

```
29. <clickbutton label="${mainMenu.button.addVolume}"/>
30. <verifytitle text="${addVolume.title}*" regex="true"/>
31. <setinputfield name="${addVolume.field.boxNum}"
      value="#{param.boxNum}"/>
32. <setinputfield name="${addVolume.field.chartNum}"
      value="#{param.chartNum}"/>
33. <setinputfield name="${addVolume.field.volNum}"
      value="#{param.volNum}"/>
34.  <setinputfield name="${addVolume.field.date}" value="#{param.date}"/>
35. <clickbutton name="${addVolume.button.save}"/>
36. <verifytitle text="${addVolume.title}* regex="true"/>
```

Example 5: the custom XML component

While this was a simple and workable solution, it falls short of being maintainable. The same tools cannot be used to refactor XML acceptance testing components and Java unit tests and system code. The second approach was a natural progression towards this end, namely turn the XML components into custom action steps (i.e., Ant targets), written in java. . Example 6 is the result of refactoring the previous test specification fragment (Example 4) to reference a custom action step that embodies the desired behavior. The **storeVariable** attribute is no longer necessary, as the parameters are passed to the Java implementation via the specified attributes (e.g., boxNumber).

```
37. <createVolume boxNumber="box1" chartNumber="chart1"
          volumeNumber="v1c1s2" archiveDate="2003-11-13"/>
```

The createVolume XML component (see Example 5), is replaced with a custom Java class (see Example 7), that encodes each detailed step as method calls that are implemented in the **CustomActionStep** super class.

```
38. public class CreateVolumeActionStep extends CustomActionStep {
39.     private String archiveDate, boxNumber, chartNumber, volumeNumber;
40.     private static final String STEP_ID = "create volume";
41.   public void doExecute(Context context) {
42.     verifyParameters();
43.     super.doExecute(context);
44.     clickbutton(context, Properties.mainMenu.button.addVolume);
45.     verifyTitle(context, Properties.addVolume.title);
46.     setinputfield(context,Properties.addVolume.field.boxNum, getBoxNum ());
47.     setinputfield(context,Properties.addVolume.field.chartNum, getChartNum());
48.     setinputfield(context, Properties.addVolume.field.volNum, getVolu-
    meNum());
49.     setinputfield(context, Properties.addVolume.field.date, getArchiveDate());
50.     clickbutton(context, Properties.addVolume.button.save);
51.     verifyTitle(context, Properties.addVolume.title);}
```

Example 7: specific custom action step

The **CustomActionStep** superclass provides Java access to each of the WebTest steps (see Example 8 for the implementation of the verifyTitle step). Each method integrates with the WebTest reporting infrastructure, so that the output reports look the same as they did previously.

```
52. public abstract class CustomActionStep extends GroupWrapper {
53.     public void verifyTitle(Context context, String title) {
54.        VerifyElementText step = new VerifyElementText();
55.        getStepSequence().getSteps().add(step);
56.        step.setStepType("verify title starts with");
57.        step.setType(ELEMENT_TYPE_TITLE);
58.        step.setText(title);
59.        step.setRegex(true);

60.        try {
61.           step.notifyStarted(context);
62.           step.doExecute(context);
63.           step.notifyCompleted(context);
64.        } catch (Exception e) {
65.            throw new StepFailedException(e.getMessage(), step); } }}
```

Example 8: generic support for custom action steps

5   Conclusions

The key drivers from this particular project's acceptance testing strategy that guided the tool selection decision were: the customer must be able to read the test specification in order to sign off on it, the acceptance tests must capture significant system workflow scenarios and must use the user interface as the primary touch point, and the tests must be developed as quickly as possible.

After a short and limited tool evaluation period, the Canoo WebTest framework was deemed the best choice for satisfying these key drivers. Was it the only choice? No, the list of acceptance testing tools and frameworks is impressive, covering the full spectrum of techniques. Was it a good choice? Yes, the framework is solid and feature rich. While it was missing some key capabilities to meet our standards for developing repeatable, readable, and maintainable acceptance tests, the framework proved to be easily extended. Was it worth the extra effort? Yes, the introduction of framework support for custom precondition, action, and cleanup steps enabled the team to develop a domain specific testing language. The elements from this testing language formed the building blocks for quickly developing user acceptance tests.

References

1. Junit, http://www.junit.org/index.htm.
2. Meszaros, Gerard and Shaun Smith "Test Automation Manifesto", XP Agile Universe Conference, 2003.
3. Beck, Kent, *Extreme Programming Explained,* Addison Wesley, 2001.
4. WebTest, http://webtest.canoo.com/webtest/manual/WebTestHome.html
5. FIT, http://fit.c2.com
6. HttpUnit, http://httpunit.sourceforge.net
7. Ant, http://ant.apache.org
8. Meszaros, Gerard, et al al "Agile Regression Testing Using Record & Playback", XP Agile Universe Conference, 2003.
9. Idea, http://www.jetbrains.com/idea/

10. Andrea, Jennitta, "Generative Acceptance Testing for Difficult-to-Test Situations", XP2004 Conference, 2004
11. http://agilecanada.com/wiki/Wiki.jsp?page=JennittaAndrea
12. Cockburn, Alistair, *Writing Effective Use Cases*, Addison Wesley, 1997.