Generative Acceptance Testing For Difficult-To-Test Software

Jennitta Andrea

jennitta@agilecanada.com

Abstract. While there are many excellent acceptance testing tools and frameworks available today, this paper presents an alternative approach, involving generating code from tests specified in a declarative tabular format within Excel spreadsheets. While this is a general approach, it is most applicable to difficult-to-test situations. Two such situations are presented: one involving complex fixture setup, and another involving complex application workflow concerns.

Key Words: automated testing, code generation, domain specific testing language, test automation patterns, testing strategy, user acceptance testing, XML, XSL

1. Introduction: Acceptance Testing Difficulties

As a result of the agile movement, teams now pay more attention to their testing practices, and seek out the advantages of automated testing. Acceptance tests[1] are performed to ensure a system has suitably implemented the user's requirements. The primary objective of an acceptance test is to ensure the core business rules are implemented correctly in the context of the overall application workflow.

Automated acceptance tests are used in a wide range of situations, motivated by a variety of different goals. The purpose commonly mentioned in testing literature is to support custom software development in a test-first manner; the acceptance tests describe the essence of what is to be developed, and objectively signal when it is complete [1]. Application integrators use acceptance tests to specify, manage, and verify outsourced or commercial software components [2]. Acceptance tests are being created after-the-fact for existing legacy systems to support both ongoing maintenance and application renewal [3].

Due to the nature of acceptance tests, a number of difficulties may be experienced when automating them that may not be experienced when automating unit tests for the

[1] Also known as customer tests, functional tests, system tests, etc.

same system. *Direct customer involvement* is crucial to the acceptance testing process. It also raises the bar for the usability of automated testing frameworks to ensure the customers can read and potentially write the automated test themselves. There are additional implications for test management, including: maintaining synchronization between the customer test specifications and the automated tests, and maintaining the tests as the user interface and/or workflow evolve over time. *Poor performance* of an automated acceptance test suite is often an issue because the tests typically operate on the user interface, and involve all of the application layers and other integrated components. Stubbing out a problematic component is a common solution to performance problems, but unless the application was originally designed for testability, selective stubbing is often impossible. *Test data management* is another area rife with difficulties. Acceptance tests are automated to accelerate and standardize regression testing. The key to achieving reliable and repeatable regression tests is the use of unique test-specific data rather than real production data [4]. If the application has not been designed for testability, it is often difficult to create the test data, or to control the execution environment in order to create a specific event.

This paper examines how a particular project overcame these types of difficulties, and developed an innovative acceptance testing strategy: generating the code to automate acceptance tests that are specified in a declarative tabular format. For the project this was not just the simplest thing that could possibly work, it was the *only* thing that could possibly work.

2. Motivation: Invention Through Necessity

We did not set out to create an alternative approach to automating acceptance tests. Our expectation was that an existing framework (e.g.: FIT [5] HttpUnit [6] jWebUnit [7] WebTest [8] to name just a few) would be used to automate our tests. As it turned out, none of these highly capable frameworks was up for the combination of challenges we faced on project Alpha[2].

The first hurdle was the large number of acceptance tests that had to be written in a very short period of time. For a variety of reasons, acceptance testing was considered late in the game; in addition, the critical system features being tested did not have adequate unit test coverage. To compensate for this, and to increase the users confidence in this part of the system, the acceptance test suite was larger than normal to include coverage of the business rules as well as the overall workflow. We started with ~100 tests, and expected this number to increase as subsequent releases introduced new features. The strategy for managing the integrity of such a large suite of acceptance tests was to ensure the test specification created by the business expert was directly executable. We could not afford the time or potential for error associated with translating a user specification into an automated test. A number of frameworks being considered were dropped from the list because coding the tests directly in Java was out of the question.

[2] Not the real name of the project.

The core challenges were architecturally rooted, thus much more problematic. Alpha was part of a family of applications, which together supported a large corporate business process. A number of the other related applications supplied portions of the precondition data referenced in Alpha's acceptance tests. Due to incomplete application integration, and technological and architectural differences between the applications, creating the precondition data required by the tests was complex and convoluted, a common side effect when testability is not a key architectural consideration. No single framework could work the magic required to dynamically set up the precondition data; we definitely needed to think outside of the box.

3. Overview: Code Generation Approach

An outline of the code generation approach is shown in Figure 1. The customer defines their acceptance tests in a tabular format within *Excel spreadsheets* using a formal domain specific testing language. An *XML* representation of the test is created as a result of running a custom macro within the spreadsheet. Members of the development team use *XSL* to transform the XML test specification into an executable *test*, which encodes the detailed steps required to interact with the *system under test*, using the syntax and mechanisms required by the target *testing framework*. An important result of this is that the *front end* can drive many different *back ends*. The XML generated from the spreadsheet can be manipulated by any number of XSL specifications to create automated tests based on any of the available frameworks (e.g., Junit, HttpUnit, jWebUnit,WebTest).

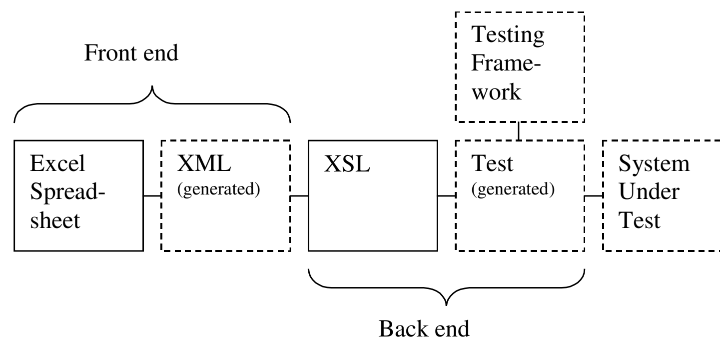

Figure 1 Code Generation Components

Inspired by FIT, our goal was to provide the customer with a declarative, clean, and powerful testing language for defining acceptance tests. The customer already frequently used Excel spreadsheets to specify calculated outcomes, so we decided to use them to document the entire acceptance test. The customer had numerous tests to

write in a short amount of time; human error was minimized wherever possible by taking advantage of Excel's cell cross reference and calculation features.

Figure 2 provides an example 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).

This is one of the simplest acceptance tests for VAS, which demonstrates the following business rules: (a) adding a new volume will create the associated chart and/or box if they don't pre-exist, and (b) volumes for the same chart may be archived in different boxes.

1	**Test Scenario**	1		
2	**Feature Name**	Add Volume		
3	**Description**	Dynamically create boxes as necessary when volumes are added to the system. Volumes for the same chart may be archived in different boxes.		
4	**Preconditions**			
5		**Volume**		
6		Volume#	Chart#	Box#
7		1	1	1
8	**Processing**			
9		**AddVolume**		
10		Volume#	Chart#	Box#
11		2	1	2
12	**Expected Results**			
13		**Volume**		
14		Volume#	Chart#	Box#
15	**unchanged**	1	1	1
16	**created**	2	1	2
17				
18		**Box**		
19		Box#		
20	**created**	2		

Figure 2 Example Excel Spreadsheet Test Specification

This specification is declarative in that it focuses on *what*, not *how*; it gives no hint of the gory details associated with actually executing the test. It is clean because only the

essential concepts are described; we have the assurance that everything else that needs to be taken care of will be. It is powerful because the various table and column headings form a domain specific testing language that is familiar to the customer.

The body of the test (lines 4-20) is divided into three sections:

- **Preconditions** identify the data that is expected to be in the system prior to running the test. The table heading (line 5) defines a business object, and the column headings (line 6) reference attributes of the object. Each row represents a separate object (line 7). Many different business objects can be specified in the preconditions section, following the same pattern shown in the example.
- **Processing** refers to workflow, or user goal level use cases [9](line 9), supported by the system. Each column heading (line 10) represents user input that is required at some point within the workflow. Workflow is processed sequentially in the order specified in this section.
- **Expected Results** are described in the last section in terms of the changes made to the business objects. The appropriate verbs (e.g., created, updated, deleted) are placed in column 1.

The XML corresponding to the spreadsheet is shown in Figure 3. A custom macro within the spreadsheet understands several simple rules about placement and the use of color within a test specification: fields describing the test start in cell (1, 1) and end at the first yellow line; major sub-sections of the test are found in lines highlighted in yellow; domain classes and their attributes are found in lines highlighted in grey; domain objects exist in the uncolored rows. The XML is semantically equivalent to the excel spreadsheet, but in a different (and noisy) format. The macro generically handles test specifications from any business domain, supporting any number of domain classes and associated attributes.

```
1 <test TestScenario="1" FeatureName="Add Volume" Description="Dynamic..." >
2 <Preconditions><classes>
3         <class name="Volume">
4                 <object Volume#="1" Chart#="1" Box#="1"/>
5         </class> </classes></Preconditions>
6 <Processing><classes>
7         <class name="AddVolume">
8                 <object Volume#="2" Chart#="1" Box#="2"/>
9         </class> </classes></Processing>
10<ExpectedResults><classes>
11        <class name="Volume">
12                <object verb="created" Volume#="2" Chart#="1" Box#="2"/>
13                <object verb="unchanged" Volume#="1" Chart#="1" Box#="1"/>
14        </class>
15        <class name="Box">
16                <object verb="created" Box#="2"/>
17       </class></classes></ExpectedResults></test>
```

Figure 3 Example Generated XML

While this is one of the simplest acceptance tests for VAS, the actual steps required to carry out the test (either manually or automated) are significantly more complicated. Automated tests contain the details of the application workflow and proper test data management, and are typically divided into four sections:

- **Fixture Setup** (lines 4-7): Create a unique volume object with the specified attributes (note, the business logic will also cause the chart and box to be created). All remaining required attributes are generated as unique, 'anonymous' values [4]. Persist this object in the database.
- **Exercise System Under Test** (lines 9-11): Ensure the objects that will be created by the test (lines 15 and 20) do not exist prior to running the test. Navigate through the screens to accomplish the user goal of adding a volume to a box. The values listed on line 11 are used as inputs as appropriate within this workflow. For this application, this simple workflow involves logging in, and navigating through 3 different screens.
- **Results validation** (lines 12-17): Navigate through the application to ensure that: the objects that should have been created actually exist, the objects that should have been deleted no longer exist, and the objects that should have been updated have the new values. Each of these validations involves multi-screen navigation starting from the main screen.
- **Fixture teardown** (lines 4-7, 12-17): Remove any objects created in the preconditions (line 7). Remove any objects created as a result of exercising the System Under Test (line 15, 20).

4. Project Alpha: Difficult Fixture Setup

As described in Section 2, the motivation for this code generation approach was to find a solution to the complicated test fixture setup problem experienced by project Alpha. Test data creation was a multi-step process performed partially using: Alpha's user interface, another application's user interface, Alpha's java API, and a series of carefully hand-written SQL scripts aimed directly at several databases (a simplified outline is shown in **Figure 4**). Multiple XSL code generators were developed to take information from the XML specification and transform it into a specific step in the data loading process. The standard acceptance-testing tool for the project was QA Run [10]. The testing team developed a customized script-based interface to supplement QA Run's standard record-playback interface.

This is a case where the *front end* and the *back end* of the tests are radically different. Because the data creation process was so complex and time consuming, the code generated to handle the fixture setup stage pre-loaded all of the data for an entire test suite while still insuring that each test operated on it's own unique data set (private fixture data [4]). The key critical success factor for pre-loading private fixture data is to follow a naming convention that ensures each test references only its own data. The

test scenario and feature name from the header part of the test specification (lines 1-2 of the spreadsheet; line 1 of the XML) were encoded into the test data during the code-generation process.

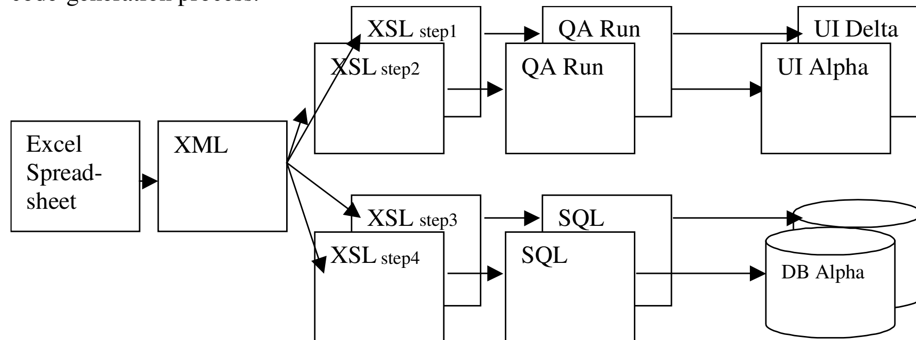

Figure 4 Multiple Targets for Fixture Setup

This turned out to be a remarkably elegant and efficient solution that resolved both the technical challenges and the usability requirements. The business experts were able to quickly develop acceptance tests, given the declarative, and domain-oriented specification language. This approach addressed all of the problems and constraints related to loading the test data, and did so in an error-free and consist manner. Ultimately, less time was spent automating the large number of acceptance tests using the code generation approach, as compared to having to hand-code each test from a customer specification.

5. Project Bravo: Complex System Workflow

The results from project Alpha were so encouraging, that a second project was immediately sought out to verify the general applicability of the approach. Project Bravo[3] also considered acceptance testing late in the game, but was very different from project Alpha in a number of ways. First, a disciplined (unit) test-first process was followed during application development. As a result, the system was very testable; none of the fixture setup difficulties experienced by Alpha were experienced on this project. Because the intricacies of the business rules were well covered by the unit tests, far fewer acceptance tests were needed as compared to Alpha. The team had been using HttpUnit to unit-test the user interface, so this was the target tool for the second code-generation case study (see Figure 5).

[3] Not the real name of the project.

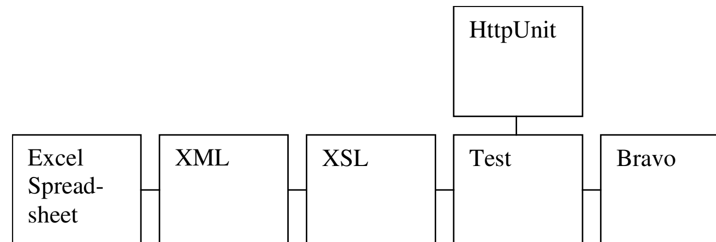

Figure 5 Code Generation for project Bravo

While Bravo did not share the difficulties of Alpha, there were different project characteristics that made this approach compelling as an option. Bravo is very rich in features, supporting many detailed business workflows. Because the declarative nature of the spreadsheets focuses on the business intent (what), not the detailed steps within the workflow (how), the test specifications will remain quite stable while the actual system evolves over time. In addition, the highly collaborative and fluid design approach taken by the team results in user interface and workflow details that are not solidified until relatively late in the sprint. Thus the *front end* of the tests can be defined early on, and the *back end* of the tests can be developed later once the details have been worked out. Furthermore, as the system evolves over time, test maintenance will typically be localized in a small number of XSL components; once the code for the acceptance tests are re-generated they are all brought up to date.

While this approach worked well the second time, it wasn't necessarily the best strategy for Bravo. The customer was not responsible for actually writing the tests, so it was not as crucial to have a user-friendly specification. As a result, the XSL code generation layer was deemed to be extra overhead, requiring specialized skills and tools that the team had not needed thus far. Because the number of acceptance tests automated on Bravo was quite low (in the order of ten to twenty), hand-coding the tests in a more direct fashion was acceptable. The team ultimately decided to use an extensively customized WebTest framework [11] for hand-coding the acceptance tests.

6. Conclusions

While there are many excellent acceptance testing tools and frameworks available today, this paper presents an alternative approach, involving generating test code from acceptance tests specified in excel spreadsheets. This approach has been tried on two substantially different projects, providing insights about its applicability. Although this is a general-purpose approach, it is not a silver bullet. In particular, the project must weigh the extra cost of developing the code-generation layer against the resulting benefits. In particular, if there are a manageable number of tests, and an existing acceptance testing framework can be used directly, then the code-generation approach would likely introduce unnecessary overhead. Thus, the most basic criterion for appli-

cability is that the situation is too difficult to test using an existing framework directly. One or more of the following project characteristics further increase the appropriateness of this approach:

- The customer writes the acceptance tests themselves and needs a simple, domain specific testing language to express the concepts clearly.
- Acceptance tests act as requirements and are focused on capturing strategic concepts (e.g., overall business rules and relationships) rather than tactical details (e.g., application steps to enact the workflow). Decoupling the specification from the automation makes this separation of concerns possible.
- A large number of tests must be automated in a short amount of time. The tests contain calculations and interrelationships between the data that a spreadsheet supports well.
- The user interface evolves over time, and test maintenance is a concern.

Does the automated acceptance testing world need yet another approach? For a difficult-to-test project like Alpha, the code generation approach worked exceptionally well, in a situation where nothing else could have possibly worked.

Acknowledgements

It's been my privilege for many years to work with and be mentored by Gerard Meszaros and Shaun Smith on test automation best practices. The work described in this paper would not have been possible without the courage, insights, and contributions of members of the *Alpha* project team: Bryan Ambrogiano, Kevin Holroyd, and Bud Newman. The work was significantly improved by the insights and contributions of the *Bravo* project team: Amy Law, Robert Purdy, Lynne Ralston, and Ross Taylor.

References

1. Beck, Kent, *Extreme Programming Explained,* Addison Wesley, 2001.
2. Andrea, Jennitta, "An Agile Request For Proposal (RFP) Process", ADC, 2003
3. Meszaros, Gerard, et al "Agile Regression Testing Using Record & Playback", XP Agile Universe, 2003
4. Meszaros, Gerard and Shaun Smith, "Test Automation Manifesto", XP Agile Universe, 2003
5. FIT, http://fit.c2.com
6. HttpUnit, http://httpunit.sourceforge.net
7. jWebUnit, http://jwebunit.sourceforge.net/
8. WebTest, http://webtest.canoo.com/webtest/manual/WebTestHome.html
9. Cockburn, Alistair, *Writing Effective Use Cases*, Addison Wesley, 1997.
10. QA Run, http://www.compuware.com/products/qacenter/qarun.htm
11. Andrea, Jennitta, "Putting a Motor on Canoo WebTest Acceptance Testing Framework", XP2004 Conference, 2004.