

Acceptance Testing HTML

narti kitiyakara taught himself nearly a dozen programming languages before earning a b.s. and an m.s. in computer science from the american institute of computer sciences. narti joined the nola team in 1998 and is now a lead systems architect in the commercial applications group.

We have been conducting an XP project for nearly two years. During this time, we have experimented with many tools and techniques for acceptance testing. Here we discuss the relative costs and benefits that we've found using a commercial testing tool, manually executed tests, and hand-coded Java tests. We will conclude with a discussion of Avignon, an XML-based extensible scripting language that we developed, which allows us to specify acceptance tests at a high level, in advance, and with relative ease.

Project Background

In late 2000, with two developers and a project manager/customer, we embarked on our first XP project. Our goal was to develop a Web-based J2EE application for commercial release. Over the next eighteen months, the project grew to six developers and consisted of over 650 Java classes, 80 Java Server Pages, and 35 database tables. Our initial process was based on *Extreme Programming Explained*¹ and the developers tried to follow all of the programmer practices, including pair-programming, unit-testing, and iteration planning. Automated acceptance testing, however, was to prove a troublesome issue.

Tools for Testing HTML

Manual Tests

When we first began the project, the customer prepared "stories" of up to thirty typed pages along with detailed acceptance tests. The plan was that the developers would use the tests to get detailed information about the story, but the customer would manually execute the test to determine whether or not to approve the story. The Quality Assurance Department was also supposed to run the previous acceptance tests on a regular basis to make sure that adding the new story had broken no previous stories.

The developers dutifully ran acceptance tests by hand for each completed story, but it turned out that the tests were so detailed that it was easy to fail on trivial points that were not noticed. It also turned out that some aspects of the manual acceptance tests were subject to differing interpretations. Thus the developers would legitimately feel that they'd passed an acceptance test, but the customer would say that they had not. Apart from that,

neither the developers nor the customer nor the QA Department was running the previous tests on a regular basis. It was only by luck, therefore, that the development team would discover that modifying existing code had changed the functionality of the application. This state of affairs led us to purchase a full-fledged acceptance-testing package.

Commercial Testing Tools

In order to try to alleviate the problems of manual acceptance testing, we evaluated several commercial testing packages, including Segue Software's SilkTest, Empirix's eTester and Compuware's QARun. All of the packages were rated on the basis of applicability to projects we were undertaking, ease of use, ability to test multiple server and client environments, and cost. As a result of this evaluation we purchased Compuware's QARun.²

Shortly after the evaluation of automated testing tools, two of us attended ObjectMentor's XP Immersion. It was there that Ron Jeffries impressed upon us the need for automated acceptance tests that the developers could run themselves.³ When we returned from the Immersion, we set about trying to convince our customer of the importance of getting the acceptance tests automated and giving the developers the ability to run them. Unfortunately, it turned out that the customer would find that the scripting facility in QARun was not amenable to writing the acceptance tests in advance.

HTTPUnit

The customer had never disputed that the acceptance tests should be automated, but still felt that they should be done after the fact by the QA Department. At this point, however, QA was still not able to run the previous tests in a timely enough manner to warn the developers of impending problems. So the developers, on their own initiative, started using HTTPUnit⁴ to manually code the acceptance tests. They had discovered the program perusing the XP Web sites. This proved quite effective at stopping us from changing old code, but interpreting the customer's ideas of how to test the details in the visual components was difficult.

Our customer has always been very concerned with the appearance of the application. HTTPUnit provides an API for examining the HTML output of an application, but we found it

awkward to use for the level of detail that our customer wanted. It turned out, however, that HTTPUnit would also expose the generated HTML as an XML document, so we could use XPath⁵ to make very fine-grained tests of the HTML without a great deal of coding effort. Unfortunately, as the tests were being developed simultaneously with the code they didn't help us maintain consistent HTML coding for a given visual effect. They were also quite brittle because the format of the HTML was compiled into the tests. Every change to the appearance of the application required laborious changes to the acceptance tests as well.

Ironically, the very success of the acceptance tests to prevent unintentional functionality changes helped hide their value from the customer. The most visible thing to the customer was the amount of time that it took to code the tests in Java. At that point, though, the developers were still the only group actually automating the test, so the customer couldn't object too strongly. But there was still a feeling that the developers should not be taking up valuable coding time with acceptance tests. For that matter, the developers tended to view the time they were spending on testing as a necessary evil. No one wanted to do away with automated tests that we could run, but they were tedious to write and there tended to be much debate with the customer about what should be tested.

What Can Be Tested?

Everything

At first, after we discovered the value of automated acceptance tests, nothing seemed too trivial to test. Not only were the correct results supposed to be put into the database, but also almost every visual element was checked. Not only were the correct results to be displayed, but every formatting element had to be correct as well. This proved to be very time consuming for the developers and also quite brittle. The acceptance tests generally needed a specific state to be set up in the application before they could be run. This made the developers generate large amounts of code to initialize the state of the application. The customer was also finding that even a small change in appearance required a fair amount of rework in the tests. Switching to a snapshot of the expected HTML would have alleviated this problem, but would have also put us back in the position of needing the application ready before the acceptance test could be done. The snapshot method would have also led to more brittleness in the tests, since the coded version could always choose how to interpret a visual element (i.e., ignoring non-visual elements of the HTML tags, such as `id` attributes, unless they were important in some way).

Functionality Only

When the cost of testing everything became prohibitive, we scaled back testing to only what was necessary to the functioning of the application. Thus, on the HTML side, we tested only things like the `name` and `value` attributes of `input` tags, references for the anchor tags, and basic text of the output. This took some of the

burden off the developers, but did not relieve them of the problem of initializing the application state before each test. It also did nothing to solve a problem discovered by the customer even when "everything" had been tested: visual inconsistency between the different HTML pages.

Visual Consistency

Our six developers could pair in fifteen different ways and each of these possibilities could render the same idea in a different way. Although the developers wanted to maintain visual consistency for the customer, this was difficult to achieve. The customer, rightly, did not want to specify too much detail about which HTML to use to achieve a certain visual effect, but different pairs ended up using different HTML to do so and the results were not always the same. One pair might put a page title into an `H1` tag, while another might put it into a `P` tag with a same font size as the `H1`. This broke down when the customer wanted to define a default CSS style for page titles. The development team had to go back and standardize, after the fact, a common way for doing titles. Detailing the visual elements in an acceptance test before the story was started would have helped matters, but no one wanted to go back to the massive coding involved in testing everything. One proposed solution was to write a separate set of tests that checked the visual appearance of the application without testing functionality beyond page generation. This would help the visual consistency issue, since it focused on one visual element at a time, but it would still be brittle and require a large amount of coding. Before embarking on this testing method, we decided to try another approach.

Origins

One of the proposed solutions for dealing with the general problem of visual consistency was to allow the developers to generate XML that would be transformed into HTML by a standard XSL.⁶ This would have removed the problem of deciding how to generate the HTML on a page-by-page basis, and allowed the developers to concentrate on the higher-level concepts of what should appear on the page. Unfortunately, the customer was concerned that XML/XSL was not, at the time, supported by enough platforms to be portable to a large number of environments. Later, however, one of our developers realized that they could still use this system in the acceptance-testing environment to actually run the tests. It could also remove the burden of coding the acceptance tests from the programmers by allowing the customer to make detailed, yet easy to change, specifications of what was expected.

Avignon is Born

Thus came about Avignon, a combination of an XML-based scripting language and a high-level page description language for executing acceptance tests against an HTML application. The customer also helps produce an XSL that converts the page descriptions into HTML. This ensures that the developers generate the same HTML to produce the same high-level concept. Avignon is a living language that is expanded, in consultation with the customer, whenever necessary.

Although the page comparisons performed by Avignon return to the brittleness of testing everything, it has proved easy enough to change the expected HTML for every page in the application by changing the XSL that generates it for the test. Because pages are described at a high level, it has also become much easier to pregenerate the expected results. The original page description language was designed for easy transformation into HTML. It quickly became apparent, however, that an even more abstract description could be given on a per-page basis. The abstract description consists only of the elements that can vary on the page. It is then transformed into the original page description language, which is itself transformed into HTML. This makes it extremely easy to change the expected results for any individual page or for every page simultaneously. (An interesting side effect has been that the customer can now quickly prototype the HTML pages entirely outside of the application. This allows the customer to quickly determine the desired look-and-feel.)

Implementation

Implementing Avignon is quite simple. A JUnit test finds all of the files in a directory with the pattern “*TestSuite.xml.” It passes each of these files through a SAX-based XML parser that validates the syntax of the file and fires an event for the start and end of each XML element. Each element must have an associated Java class matching *ElementNameHandler*.class that implements the *AvignonTagHandler* interface. This interface defines two methods:

```
void start(AvignonTestState state,
Attributes attribs)
and
void end(AvignonTestState state)
```

The *AvignonTestState* class allows each tag to access its parent tags, add messages to the error log and request Web pages from the actual application.

Scripting Language

At this time there is no formal schema or DTD for Avignon, but a brief overview follows.

Test Definition Elements

These elements allow the user to integrate the acceptance tests with our ISO 9001 framework and to initialize the application state before a test.

TestSuite

This is the top-level element for any Avignon test suite file. It has one attribute, *unitName*, which defines the name of the unit being tested. We use this name to help track test results. When this tag ends, it records all of the test results in a database in accordance with our ISO 9001 policies. This element contains any number of *AcceptanceTest* elements.

AcceptanceTest

This element defines an actual test to be performed. It also has one attribute, *testName*, which it gives to its parent *TestSuite* element along with a pass/fail test result. This tag is also responsible for restoring any application state that has been modified by the test. This element may contain one *DatabaseState* element and any number of other test command elements.

TestScript and ScriptParam

These elements allow the user to execute a separate file containing an Avignon script. The *TestScript* element takes a name attribute that tells it what file to execute. Before executing the script, however, it will do a textual substitution of the information provided by the *ScriptParam* elements contained within. This allows the user to execute the same script with different data without having to actually duplicate the script.

DatabaseState & DatabaseTable

These elements allow you to perform low-level database operations before a test is run. This requires the customer to have knowledge of both SQL and the application database, but it is easy enough to add application-specific tags, implemented either with an XSLT that converts them into the low-level tags or with Java handlers that simplify the initialization of the database for the customer.

The *DatabaseState* is used simply to group *DatabaseTable* elements. Each *DatabaseTable* element is responsible for saving the state of the table specified in its attribute. Each *DatabaseTable* element may also have zero or more of the elements below in any combination.

DatabaseInsert

This element allows you to define a SQL statement of the form `INSERT INTO table_name [(columns)] VALUES (values)`. The table name is defined by the surrounding *DatabaseTable* element, *columns* is defined by an optional attribute to *DatabaseInsert* tag. Finally, *values* is defined by a required attribute to the *DatabaseInsert* tag. You format both *columns* and *values* in such a way that the SQL statement will work correctly when it is executed.

DatabaseUpdate

This element allows you to define a SQL statement of the form `UPDATE table_name SET field=value[,field=value]* [WHERE condition]`. The value of *condition* is defined by an optional attribute to this tag, while the set clauses are defined by one or more sub-elements. These sub-elements, *UpdateField*, each take two parameters: a field name and the value to set it to.

DatabaseDelete

This element allows you to define a SQL statement of the form `DELETE FROM table_name [WHERE condition]`. Again the *condition* comes from an optional attribute to this tag.

Test Command Elements

These elements allow you to perform operations through the application's Web interface. Each of these elements take an optional attribute, `pageDescription`, that allows you to specify an XML description of what the resulting page should look like. The comparison is done by generating the HTML for the actual page and the expected HTML (transformed from the XML description) and comparing them, without regard to white space. The comparison is done without regard to white space because it proved too difficult to generate the same HTML, including non-significant white space, as was expected. Given that the possibility of incorrectly generating significant white space was small, We chose to ignore white space altogether. Each of these elements also has an optional `preTranslation` attribute. If this attribute is present, the XML specified by `pageDescription` is first transformed by the style sheet defined in `preTranslation` and then transformed into HTML.

MenuClick and PageClick

These two elements are essentially the same, but the former operates on the application's HTML menu while the latter operates on the content frame of the application. Both use HTTPUnit to click a link on the page given its text.

SubmitPage

This tag allows you to fill out an HTML form on the current content page and submit it back to the application. It takes the value of the submit button as an attribute and fills out the input values in tab order (unless otherwise specified) with values given by its sub-elements, `InputValues`. Each `InputValue` element must have a `value` attribute and may have either a `name` or a `position` attribute. If no `name` or `position` is specified, the value goes into the current input box and the focus is moved to the next input box. If a `name` or `position` is specified, the focus is first moved to the correct input box, the value is put into that box and the focus moves to the next input box.

DatabaseAssertion

This element allows you to specify that the application's database must be in a given state for the test to pass. You specify three attributes, a `tableExpression`, an optional `whereCondition` and an `expectedCount`. The system generates a SQL statement of the form: `SELECT COUNT(*) FROM tableExpression [WHERE whereCondition]`. The assertion passes if the result of this statement matches the expected count.

User-Defined Elements

The preceding elements represent the code of the Avignon language, but it was never meant to be static. Customers are encouraged to create their own elements. This can lead to some interesting element names (an *EggClick* element, for example), but has proved reasonably successful when the customer consults with the developers about how to phrase what he wants to do, such as whether attributes or sub-elements would be easier. It has also helped to reduce the developers' misunderstandings of what the acceptance tests were doing. Instead of having to describe the whole

test to the developers, the customer now needs to concentrate only on the new elements in the test.

For each new element the customer defines, the developers must add a Java class implementing the `AvignonTagHandler` interface. This class implements the customer's intentions for the tag, be it an assertion or some action command. Because the handlers are dynamically loaded, no recompiling or relinking of the Avignon system is necessary when adding new elements.

An Avignon Example

The following test scripts sets up the test information and calls the "CreateScript.xml" script:

```
<?xml version="1.0" encoding="UTF-8"?>
<TestSuite name="Composer Information">
  <AcceptanceTest name="Add New Composer">
    <TestScript name="CreateScript.xml">
      <ScriptParam
name="ComposerListing"
value="EmptyComposerListingPage.xml"/>
      <ScriptParam
name="DatabaseScript"
value="NoComposerSetup.xml"/>
      <ScriptParam name="DisplayName"
value="TestDisplay1"/>
      <ScriptParam name="ActualName"
value="TestActual1"/>
      <ScriptParam name="ResultPage"
value="Test1AddedForm.xml"/>
    </TestScript>
  </AcceptanceTest>
</TestSuite>
```

The "CreateScript.xml" file itself is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ScriptBlock>
  <DatabaseState>
    <TestScript name="#DatabaseScript#"/>
    <DatabaseTable name="T_SEQUENCES">
      <DatabaseUpdate
where="SEQUENCE_NAME='COMPOSERS' ">
        <UpdateField
name="SEQUENCE_VALUE" value="-1000"></
UpdateField>
      </DatabaseUpdate>
    </DatabaseTable>
  </DatabaseState>
  <MenuClick itemText="Browse Composers"
pageDescription="#ComposerListing#"
preTranslation="ComposerBrowserTransform.xml"></
MenuClick>
  <PageClick itemText="Add Composer"
```

```

pageDescription="EmptyAddScreen.xml"
preTranslation="ComposerInformationTransform.xml"/
>
  <SubmitPage
pageDescription="#ResultPage#"
preTranslation="ComposerInformation
Transform.xml">
  <InputValue name="DisplayName"
value="#DisplayName#" />
  <InputValue name="ActualName"
value="#ActualName#" / >
  <InputValue name="DateOfBirth"
value="#Born#" />
  <InputValue name="DateOfDeath"
value="#Died#" />
  </SubmitPage>
</ScriptBlock>

```

When the test is run, it will call yet another test script (defined by a parameter) to set up the database, then click on Browse Composers, checking the results against the HTML generated by the EmptyComposerListingPage.xml data file and the "ComposerBrowserTransform.xml" XSL. It will then click the link to add a composer and submit the resulting page with the given input values. When the test finishes, it restores the original state of any database tables that were modified inside the DatabaseState element.

A Solution for Everyone

We tried many forms of acceptance testing, from manual tests to fully automated systems. The manual tests were unsatisfactory all around, they were subject to differing interpretations and expensive to execute. The hand-coded tests reduced the costs of running the tests but increased the costs of creating them. The developers, knowing how often it saved them from making unintentional changes to the application's functionality, felt that they were very worthwhile, but this value was somewhat hidden from the customer. It also left too much of the interpretation of the tests in the hands of the developers.

We also tried a commercial tool for acceptance testing. This would have helped the customer code the tests himself, but the nature of the tool meant that the tests would have to be written after the stories were completed, thus depriving the developers of the opportunity to run the tests on the current story.

Avignon is an attempt to provide a satisfactory solution to both the customer and the developers. The use of XML/XSL has kept the cost of implementing Avignon quite low, yet it gives the customer a relatively easy way to specify tests in advance without ambiguity.

This allows the developers to work with confidence that they're aiming for the right target and know how far off the mark they are while they're coding. In the near future I'd expect to see Avignon brought into the realm of testing more traditional user-interfaces and

perhaps even take the burden of UI coding off the developers' shoulders.

Reprinted with permission from Don Wells and Laurie Williams, editors, Extreme Programming and Agile Methods - XP/Agile Universe 2002. Second XP Universe and First Agile Universe Conference, Chicago, IL, USA. August 4-7, 2002. Proceedings. Lecture Notes in Computer Science Vol. 2418

References

- ¹ *Extreme Programming Explained*, Kent Beck (Addison-Wesley, 2000)
- ² For more information about *QARun*, see Compuware's web site: <http://www.compuware.com/products/qacenter/qarun/detail.htm>.
- ³ Also expressed in: *Extreme Programming Installed*, Jeffries, Anderson and Hendrickson (Addison-Wesley, 2001)
- ⁴ For more information about HTTPUnit, see <http://www.httpunit.org>.
- ⁵ For more information about XPath, see <http://www.w3c.org/TR/xpath>.
- ⁶ For more information about XSL, see <http://www.w3c.org/Style/XSL>. Neil Bradley's *The XSL Companion* (Addison-Wesley, 2000) is also a good introduction.