COMP33711: Agile Software Engineering

Writing Acceptance Tests: A Short How-To Guide

Additional (Examinable) Notes for Agile Testing Session (Week 10)

Suzanne M. Embury
November 2012

What Are Acceptance Tests?

This document supplements the material covered in the session on business-facing agile testing (week 10). It describes how to write acceptance tests for user stories, and its contents are examinable. In particular, you may be asked to write acceptance tests in the exam for the course (see the sample exam paper for an example of the kind of question you might be asked on acceptance tests).

Acceptance tests are functional (i.e., end-to-end) tests that describe features the customer is willing to pay for. The name derives from the fact that, in conventional development processes, the acceptance test suite contains the test cases the customer will run on the delivered system to determine whether the asked-for functionality has indeed been provided and, therefore, whether the full price for the software should be paid to the developers. That is, the test suite determines whether the software is "acceptable" to the customer. Many different names are used for this kind of test suite. You may have come across such terms as "user acceptance tests" (UATs) or "customer acceptance tests" (CATs), for example.

In agile projects, acceptance tests play a slightly different but equally important role. Recall that user stories are only one part of the requirements gathered in an agile project. The full requirements are given by what are sometimes called the 3 Cs: Card (i.e., the story card), Conversation (i.e., the conversations with the customer about the story) and Confirmation (i.e., how to confirm that the implementation of the story is complete). This last part, the confirmation, is expressed as a set of acceptance tests. When the software builds without error and passes the acceptance tests in the deployment environment, then we can regard the story as being "(done) done".

What distinguishes acceptance tests from the other kinds of software test we look at in this course unit is that they are business-facing, rather than technology-facing. That is, they are expressed in the language used by the customer, and they are written at an abstract level, so that they are not tightly coupled to the details of the current implementation. The first of these properties is important because it means that acceptance tests can be understood (and even written) by customers, and that therefore all parties can agree on what confirmation of a particular story means. The second property is important because it means that the acceptance tests are not brittle: they do not need to be changed every time some detail of the implementation changes. If we compare this with the kind of tests we write in JUnit (for example), we can see that in contrast the tests are technology-facing: they are expressed using the terms chosen by the developers for classes and methods, rather than in the terms used by Customers (there can of course be some overlap here), and they are tied closely to the details of the current implementation. If a method gains a new parameter, or throws a new exception, then the JUnit test must be updated—even if no real change in functionality has occurred.

A further unusual feature of acceptance tests in agile projects is that they are (as far as practical) automated. This means they can be run frequently and thus give an objective measure of project progress. It also means that they become "living" documentation. Unlike a passive document, which can become increasingly out of step with the code without anyone realising it, automated acceptance tests will fail when they fall out of step with what the code is doing. Thus, there is a real incentive to keep the tests up-to-date as the requirements change.

Writing Acceptance Tests

There are currently two main ways to write acceptance tests: as tables (as exemplified by FitNesse) or as scripts (as exemplified by Cucumber). Both approaches have their advantages and disadvantages, and both will be the best choice in some situation or another. We are going to look at how to write acceptance tests in the form of tables, but many of the basic principles we cover also apply to the scripting approach.

We will illustrate the process of writing acceptance test tables by looking at how we write tests for the following example story from a society event management system:

> As a ticket purchaser, I want to know how much it will cost me to buy a certain number of tickets for a certain event, so that I can make the right decision to purchase or not.

On the back of the story card, the team has written some "conditions of satisfaction" (CoS). CoS are brief statements of the main features that must be implemented for the story to be considered complete (usually written as bullet points on the back of the story card). CoS are a quick way to grab important details from conversations, and are often the starting point in terms of writing acceptance tests. In this case, the CoS written on the card are:

- Society members get 50% discount on all ticket purchasers.
- Society affiliates get 20% discount if no money owing.
- Affiliates get 0% discount if owe money to society.
- Non-members must pay £5 surcharge when booking tickets if money owed to society at time of booking.

From these details (the story, any CoS that are given and the conversations with the customer), we can extract the general shape for the acceptance tests. Acceptance tests have the same basic components as any test case, as discussed in the pre-reading for the Agile Testing session: a starting state, a set of input values, details of the test to be executed, the expected result if the functionality is correctly implemented and any tidy-up instructions. We'll look first at how to specify tests that don't require any particular starting state or tidying up. The functionality to be tested is broadly indicated by the story itself. So, our main task is to work out what the inputs to the acceptance tests should be, and what the expected outputs are.

If we concentrate just on the story itself, we can see that two inputs are mentioned: the number of tickets required and the event for which the tickets are required. The output is the total cost for the tickets. For table-based acceptance tests, we need inputs and outputs which are atomic values (strings, integers, etc.) that can be entered into a table, rather than complex values like classes and objects. The "number of tickets" input quantity and the "total ticket cost" output quantity both fit this requirement - we can enter an integer into the table for the number of tickets, and a 2-place decimal for the total cost. But what about the event input? An event is a complex object, not an atomic data type. We could use the event name, as a shorthand, but the key information we actually need about the event is not its name but the cost of tickets for the event. A quick conversation with the customer reveals that, in this first version of the system, we do not need to support variable ticket prices for events – all tickets for a given event will have just one price. So, a single input giving the price per ticket for the event should be sufficient for our acceptance test. This gives us an initial table structure as follows:

what is the total ticket cost		
Number of tickets	*Price per ticket*	*Total Cost?*
1	£10.00	£10.00
4	£11.00	£44.00

The first line tells us the functionality that will be exercised by the test. The next line gives the column headings, showing the input and output quantities. Here, we adopt the convention used by the FitNesse acceptance test framework to indicate which are input columns and which are outputs: the output column names end with a question mark, and the input column names do not.

The remaining rows of the table are for the test cases themselves—one row is equivalent to one test case. In this case, we've just given two very simple contrasting cases, the meaning of which are hopefully self-evident. Some acceptance test frameworks advocate the use of a scenario or template to help customers read the test case tables, when they are more complex. In this case, for example, we might use the following scenario:

> When a ticket purchaser wants to buy <number of tickets> tickets for an event at <price per ticket> per ticket, the total cost should be <total cost>.

If we now look at the CoS, we get more details about the behaviour required to implement this story, which change our understanding of the tests to be written. As well as adding up the base ticket prices, we need to apply certain discounts and surcharges to the total, depending on the relationship of the ticket purchaser to the society, and her/his payment history. This means that we need more inputs to fully specify the test. The code which implements this behaviour will probably take as input a Person instance or similar, but we can't give a complex type as an input to a table-based acceptance test. Instead, we look for the simple attributes of person that give the raw information we need: in this case, the membership status of the ticket purchaser and the amount of money owed by the purchaser to the society. So, we need to expand our table structure to the following:

what is the total ticket cost				
Membership type	*Amount owed to society*	*Number of tickets*	*Price per ticket*	*Total Cost?*

The scenario for reading the table rows now becomes:

> When a ticket purchaser who is <membership type> of the society, and owes <amount owing> to the society, wants to buy <number of tickets> tickets for an event at <price per ticket> per ticket, the total cost should be <total cost>.

The next task is to add in enough contrasting test cases to illustrate the full behaviour required for the story. Below, we fill in six rows of the new table structure, giving a set of test cases which (largely) cover the range of behaviours described by the story, the conversation with the customer and the conditions of satisfaction.

what is the total ticket cost				
Membership type	*Amount owed to society*	*Number of tickets*	*Price per ticket*	*Total Cost?*
a member	£0.00	2	£10.00	£10.00
a member	£0.01	2	£10.00	£10.00
an affiliate	£0.00	2	£10.00	£16.00
an affiliate	£0.01	2	£10.00	£20.00
not a member	£0.00	2	£10.00	£20.00
not a member	£0.01	2	£10.00	£25.00
not a member	£0.00	0	£0.00	£0.00
not a member	£0.01	0	£0.00	£0.00

Notice the values chosen for each row in the table. None of them have been selected at random. Instead, we have chosen values that either cover the set of possible values in some column, are at a boundary where the behaviour of the system changes, or else are values for which the right answer is clear and obvious. For membership type, we made sure we had rows that include all possible membership types. We chose to

calculate the total for orders of 2 tickets in these test cases to show that the discounts/surcharges apply to all tickets in the order, and not just to the first one bought, as well as to indicate that the ticket costs are summed to create the total. In the case of non-members, we chose also to show the result when 0 tickets are ordered, so that we can show the behaviour of surcharges at this important boundary case. The "amount owed" values were chosen to show the boundary at which the discount/surcharge rules change (any amount of debt to the society results in a different behaviour for certain membership types/order amounts, no matter how small). Finally, we chose the value £10 for the ticket prices as this value is a plausible one for the application domain and is easy to create 50% and 20% discounts for, manually.

Adding a Starting State to Acceptance Tests

The example just presented showed how we can use tables to write a collection of acceptance tests for a single functionality in a compact and (hopefully) readable way. The example we looked at was a simple one, which required just a single table to describe the full tests. In general, however, we'll need to use multiple tables to describe test cases for a single piece of functionality. This might be because a quite different set of inputs is needed to get the software into the right starting state for the test than is needed for the test itself, or because the test requires a more "script-like" than declarative style (do this, then do this, then do this). Another reason for using multiple tables is to help make the test more readable to (and writable by) end-users, who can find the bare-bones style of the example given in the previous section hard to follow because of the lack of context.

The solution to all these problems is to create a test that consists of a sequence of tables. For example, we might encode our society event ticket cost example using the following three tables:

some people		
Name	*Membership type*	*Amount owed to society*
Fred Smith	a member	£0.00
Julie Jones	a member	£0.01
Mabel Able	an affiliate	£0.00
Terri Took	an affiliate	£0.01
John Jakes	not a member	£0.00
Bill Beans	not a member	£0.01

some events		
Event name	*Event date*	*Price per ticket*
2013 League Finals	06/06/13	£50.00

what is the total ticket cost			
Member name	*Event name*	*Number of tickets*	*Total Cost?*
Fred Smith	2013 League Finals	2	£50.00
Julie Jones	2013 League Finals	2	£50.00
Mabel Able	2013 League Finals	2	£80.00
Terri Took	2013 League Finals	2	£100.00
John Jakes	2013 League Finals	2	£100.00
Bill Beans	2013 League Finals	2	£110.00

In this formulation of the ticket total tests, we add two tables to populate the system with some people to buy tickets and an event for which tickets can be purchased, before the table which executes the tests. These two new tables don't have any output columns. This indicates that their job is simply to execute the functionality referenced by the table, but not to place any assertions on the outcome. (Some people recommend having a boolean output column for such tables that indicates that the data has been successfully added. This would certainly be needed when the functionality under test is the ability to add new people or events, in this case. But we can dispense with it when we are simply getting the system into the desired state.)

Hopefully, it is clear from this example why some people (especially customer team members, who have no specific training in writing acceptance tests) might find this formulation of the tests more readable than the one given in the previous section. In this version of the tests, we are clearly talking about what happens when some (imagined) people book tickets for some (imagined) event. However, it is important to bear in mind that all this additional information doesn't actually add anything to the test cases themselves. Both versions (the single table version and the three-table version) test exactly the same tests. And when customer comfort is not an issue, the single table version is to be preferred, since it describes the test *and only the test*. All these extra details about person names and event dates are irrelevant to the core logic of the tests, and therefore risk making the tests more brittle when included. (We would certainly be working hard to avoid them if these tests were unit tests, rather than customer-facing acceptance tests.)

In some cases, however, we really need additional tables to express the test we wish to code. This is the case, for example, when the functionality we are testing operates over a large collection of values and not over a single value. How can we express our collection as a value in a column of a table? If the collection is small, we can sometimes get away with using a textual encoding of the collection, such as "{boat, train, tram}". Or, if the collection is known to have a fixed maximum size, we can add that number of columns to the table and leave some columns blank in rows that describe collections with fewer than the maximum number of items. However, neither of these are general solutions that can work in all cases.

Instead, we can create a separate table to describe the population, and use that to provide the starting state for the test itself. For example, suppose that our customer tells us that it is now important to support different ticket prices for events. Now we need some extra tables to set up some ticket prices, as well as having the notion of an order or basket, which describes the set of tickets a particular person wishes to buy. The following tables show what I think is the minimum amount of information needed to describe a good set of acceptance tests for calculating the total ticket cost, under these semantics. (If you can see a better way to represent these tests, please let me know.)

some ticket prices	
Ticket code	*Price*
A	£1.00
B	£5.00
C	£12.00

some basket entries		
Basket ID	*Ticket Code*	*Number of tickets*
1	A	2
1	B	3
2	A	2
3	A	1
3	B	2
3	C	3

what is the total cost of a basket of tickets			
Basket ID	*Membership type*	*Amount owed to society*	*Total Cost?*
1	a member	£0.00	£8.50
1	a member	£0.01	£8.50
1	an affiliate	£0.00	£13.60
1	an affiliate	£0.01	£17.00
1	not a member	£0.00	£17.00
1	not a member	£0.01	£22.00
2	not a member	£0.00	£2.00
3	not a member	£0.00	£47.00
4	not a member	£0.00	£0.00
4	not a member	£0.01	£0.00

The first of these tables sets up some ticket codes, with different prices. The second sets up some baskets with orders for various ticket codes. The third associates the baskets with purchasers with varying types of membership and amount of debt to the society, and describes the actual assertions that make up the test. Note that only this last table has an output column (with a question mark at the end of the column name).

There are several points to note about this acceptance test design:

- Unlike the previous multi-table test example, in this case, we need to use multiple tables just to express even the minimal test cases. There's no way we could write tests for baskets that include more than one ticket code in a single row of the table, without making hard assumptions about how many ticket codes a single basket can contain.

- The final table contains more rows than before, because now we are testing a more complicated functionality (applying discounts/surcharges to different ticket prices, rather than just to one) and so we need more test cases to cover the full set of different behaviours.

- As the behaviour under test gets more complicated, it becomes harder to know what values we should choose for the test cases, and how many test cases we need. Why did I choose to have 3 ticket codes, rather than 2 or 4? Why did I use a single basket for most of the test cases? This is the challenge of acceptance test design: trying to cover all the important cases, without including lots of redundant tests. However, in my opinion, it is more important to have a sensible reason for the choices you have made, than to have made the absolute best choices. That is, it is better to reach a sensible (defensible) position quickly, than to spend a long time trying to find the perfect set of tests. After all, with this approach, it is extremely easy to add another test case, just by adding some more rows to the tables.

- I made the following choices in designing these test cases. I wanted to use a single basket to test all the discount/surcharge cases we tested in our original version of the tests. This is basket 1. It contains more than one ticket code (to show that the discounts apply to all ticket codes ordered, and the surcharges just once, regardless of how many ticket codes are ordered). I chose 2 and 3 as the ticket quantities to avoid the edge effects that come into play when 1 is chosen, and also to show that different quantities can be ordered for the different ticket codes. I also wanted to write tests that show what should happen when just one and more than two ticket codes are ordered in a single basket, so I created baskets 2 and 3 (respectively). I wanted to keep this group of tests simple, so chose a non-member without society debt for these tests, so that no discounts or surcharges needed to be applied. Finally, I wanted to add tests showing the behaviour for an empty basket, including the case for a non-member with a debt to the society (which shows that the surcharge is not applied

until some tickets are actually added to the basket.) These choices seemed to me to cover the main cases, with a small number of baskets and ticket codes. There are obviously other sets of choices that could be equally good, or even better.

- I didn't include any tests here with invalid baskets (for example, multiple basket entries with the same ticket code, or basket entries with an invalid ticket code or a negative ticket quantity). Dealing sensibly with invalid baskets is an important part of the functionality, and should be tested. But not in the same set of tests that we are writing now, to test the calculation of the total basket cost. It is essential when writing acceptance tests that we try to focus the tests on a small independent piece of functionality. If we lump lots of functionality into the same test, it may look superficially as though we are getting a good deal – we get to test several pieces of functionality for the effort of writing and executing a single test. However, this is a false economy. On the one hand, it makes it difficult to make good choices for the number of test cases required, and their attribute values. On the other, it makes failure of an individual test case very difficult to interpret. Which of the functionalities tested by the failed test has been implemented incorrectly? What exactly is the error? We want failure of an acceptance test to mean something concrete and specific: e.g. that there is a problem with applying discounts for affiliates, and not that there is something wrong with some aspect of basket handling.

If the customer next tells us that she wants events to have their own set of ticket codes and prices (to support events at different venues, perhaps), we would need to add yet more tables, and more rows, to cover this additional functionality. You might want to think about how the table structures would need to change to take this more complex functionality into account, as an exercise in applying this approach.

Automating Acceptance Tests Using FitNesse

So far, we have discussed how to express acceptance tests using tables but we have said nothing about how they are automated—that is, how the tests can be executed automatically, by (for example) your team's continuous integration and test harness. The tests can obviously be executed manually, by having some member of the team (or an external test team) work down the tables, entering the values in the columns and recording the results. This is much less useful, however, than being able to execute the tests as often and whenever we wished, regardless of whether the human resource needed to execute them is available.

FitNesse (and similar acceptance test automation frameworks) allow us to do just that. FitNesse provides a wiki for specifying executable acceptance tests. The tables describing the tests are added to wiki pages by using either the special table markup or copy-pasting from a spreadsheet. Each wiki page containing tables can be "run" – i.e., the tests the page describes can be executed against the software under test. Once executed, the wiki page is redisplayed, with the output cells coloured green or red, depending on whether the test passed (meaning that the software returned the value given in the output column) or failed (meaning that the software returned something different, or failed to return at all). For failed tests, the actual value returned is also shown, as well as the expected value given in the table.

But how can a table on a wiki page be "executed" like a test? For this to happen, FitNesse needs to be told how the values in the tables relate to the actual code that can be run. This is done in a Java class (for Java versions of FitNesse and the underlying test engine), called a "fixture class", which should be somewhere on the classpath. The name of this class should be the same as the string given in the first row of the test table, when spaces are removed and converted to camel case. For example, our test table called "some people" has a fixture class called SomePeople, while the main test table called "what is the total ticket cost" causes the execution mechanism to look for a fixture class called WhatIsTheTotalTicketCost.

FitNesse fixture classes are so called because they have the task of setting up the fixture for the tests, but they also take on the job of executing the tests, and of figuring out whether the test passed or failed. Fixture classes must be instantiable, and have one field for each of the columns in the table, because the execution engine will create one instance of the fixture class for each row of the table. The values from the table row will be copied into these fields, by the execution engine. Fixture classes must also have methods defined on

them for each output column in the table, with the same name as the output column. For example, our WhatIsTheTotalTicketCost fixture should have a method called totalCost() defined on it. This method uses the values in the fields (from the input columns in the table) to pass as parameters to the software under test. It gets a result value from the software, which it compares with the value loaded into the field corresponding to the method's particular output column, and returns the result to FitNesse, for display.

And Finally...

We have given a brief description of how to design acceptance tests, and an even briefer overview of how such tests, expressed as tables, in a form readable and writable to the customer, can be executed automatically. As with so much of software engineering, you will learn far more from your first attempts to write acceptance tests than from reading documents like this. If you would like more information about FitNesse, you can find documentation, downloads, and example tests and fixtures, in the FitNesse User Guide at fitnesse.org. You can also find another full worked example of acceptance test design in the sample exam paper/marking scheme for the course unit, given on Moodle. Or, you are welcome to contact me with specific queries.

To close, there is one last important point to make about acceptance tests, in the context of agile projects. If you look at the test cases we have been writing, you will see that each row of each assertion table corresponds to an end-to-end slice through the functionality of the system as a whole. In other words, when we take our stories (which should be high-level end-to-end slices) and break them down into acceptance tests, what we are doing is creating a set of even more fine-grained slices of value-laden functionality. We can then concentrate our efforts on implementing these very fine-grained slices, knowing that if we manage only to make one more acceptance test pass in an iteration, we will have delivered a unit of useful value. When all the tests pass, we have the required confirmation that we have implemented the larger end-to-end slice described by the story. Thus, acceptance tests form the detailed specification for the system that the stories outlined, but in an executable form.