Acceptance Testing

www.plainprocess.com

gary.mohan@plainprocess.com

This book can be downloaded for free, in PDF format, at:
http://www.plainprocess.com/uat.html

Table of Contents

Introduction

Who Is This For?

This is aimed at end users who need to acceptance test a system. No prior knowledge of testing is assumed and this book has deliberately been kept as concise as possible.

What Does It Cover?

This book covers a simple testing approach, based on techniques that have been in use for decades. It describes how to write a test script and how to design test data intended to test against your acceptance criteria.

Why Bother Testing?

If you rigorously test a system prior to accepting it into the live environment, you achieve two important objectives:

- It has been verified that the system works as specified.
- Problems with the way the system was specified are more likely to have been identified, preventing the system from going live prior to it being genuinely ready.

Definitions

Prior to discussing how to test a system, it is important to define some terms:

Acceptance Criteria: The agreed list of criteria under which the system is deemed acceptable. Typically, this will be an agreement that the system performs the functions described in a series of functional specification documents.

Branching: An approach for storing computer software source code that allows multiple versions of a system to be worked on simultaneously. This matters in acceptance testing, as a particular version will only contain functionality specific to that version.

Bug Report: This is a report raised when documenting that the system is not working to specification. Typically, this is covered under the existing project budget.

Change Request: This is a request to change the behavior of the system, addressing the fact that although the system is working to specification, the specification itself needs to change. Typically, this requires new budget.

Functional Specification: The body of documentation that specifies how the system should behave. For example on a project using an "Agile" approach, this will be a repository of "User Stories". Other formats include "Use Cases". Non-textual formats may also be used, such as diagrams and screen mockups.

Integration Testing: Prior to handing a system over for System Testing, software engineers will have a testing exercise to ensure that the individual modules (units) within the system "integrate". Usually, this is done to find issues around data formats and system state that cannot be easily tested at a module level.

Performance Testing: Testing the system's speed. Strictly, if the functional specification is silent about performance requirements, this is considered "non-functional" testing. Typically, this is done using special tools and hardware. It is intended to find so called "memory leaks" (where the system fails to de-allocate memory after completing a functional operation) or to find unacceptable lags/errors that result from multiple users concurrently operating on the same data.

Regression Testing: An end-to-end test of all test cases defined for System Testing, to de-risk major changes implemented within the system.

System Testing: Testing carried out by an independent test team, to ensure that the system meets end-to-end functional criteria. All aspects of the functional specification will be tested at this stage. This is not the same as acceptance testing and is usually carried out as the step prior to handing the system over for acceptance testing.

Test Case: A low-level testing scenario, tied to the output expected when performing a specific function with specific input data.

Unit Testing: Testing carried out by software engineers on a module-by-module basis. Usually, these tests are implemented in separate software code, distinct from the core system code. Often, running of unit tests is automated to take place across the entire system on a regular basis, as software engineers complete individual parts of the functional specification.

How To Carry Out Acceptance Testing

To acceptance test a system, you will need to:

- Have acceptance criteria in place that allow you to agree with the software development team whether or not the system is acceptable.
- Define how the test environment should be set up, prior to beginning tests.
- Define a series of test scripts, intended to test the entire functional specification of how the system is supposed to operate.
- Test data, associated with the test scripts, designed to test the system's operation.
- A test plan describing the order in which specific test scripts will be run, with specific test data. This plan may document that some tests can be run in parallel.
- Write Bug Reports for test failures.
- Write Change Requests to change the system specification, where the system works as specified, however, the specification is not acceptable for deployment into the live environment.

Acceptance Criteria

Prior to starting testing, you should have criteria in place that allow you to agree whether the system is acceptable or not. In an ideal world, these criteria are agreed at the start of the project. Often, this is not the case and acceptance criteria are agreed late in the project.

It is critical that you have acceptance criteria in place, even if the these criteria are stated as a one-sentence statement like:

> "The system will be accepted into the live environment if it meets all the requirements defined in specification documents 001 to 017."

If you don't even have something as basic as this in place, you are in trouble. You don't have a "meeting of the minds" with the development team on what the system is supposed to do and any testing you carry out will lack authority. This mess could result in you wasting valuable time prior to a delivery deadline, being "forced" into accepting the system into the live environment without really knowing if it will work.

If you end up identifying bugs in the live environment, the development team will be within their rights to say that these are change requests, with potential budget implications and embarrassment for you. You must have a clear definition of what you are supposed to be testing.

Setting up the Test Environment

Prior to beginning testing, you need to define how the test environment should be set up. This is intended to replicate how the live environment should look prior to deployment of the new system. The definition of the test environment *might* include:

- Software versions. This is particularly important if the system is being developed using a "branching" strategy. The software development team may be working on various versions of the system simultaneously, with specific versions tied to particular release dates.
- Specific usernames and passwords required for logging into the test environment. These may be users with different roles and security privileges.
- A suite of pre-loaded test data.

There are no universal rules around how a test environment should be set up and this depends on what it is you are testing. Like needing defined acceptance criteria prior to starting, having no definition *at all* of how the test environment should be set up usually means that you are already in trouble.

Again, the definition could be one sentence:

> "The acceptance test environment should be loaded with version X of the system, with the test data defined in spreadsheet Y and the login users specified in spreadsheet Z."

Writing a Test Script: The Login Screen Example

Test scripts are a documented series of steps designed to validate the defined functionality of a system. For example, to validate logging on with a test username and password, a test script could follow a pro-forma format similar to:

1. Open Internet Explorer at the address of the acceptance test system: http://uat001/login
2. In the *User* box, enter testuser001
3. In the *Password* box, enter testpassword001
4. Click *Login*
5. Verify that the user's login page has been displayed.

The above example specifies the location of the user acceptance system and a particular test username/password. It is assumed that these have been correctly set up, prior to beginning testing. This example is, however, missing negative test cases.

Negative test cases are situations designed to test the defined error functionality of a system. Alongside logging on a user presenting a valid username/password, the system should also handle cases where data is invalid or unspecified.

The example above could be restated as:

1. Open Internet Explorer at the address of the acceptance test system: http://uat001/login
2. Without entering a username or password, click *Login*
3. Verify that the on screen error message is “Your username and password were blank. Please try again with a valid username and password.”
4. In the *User* box, enter testuser001
5. Click *Login*
6. Verify that the on screen error message is “You must enter a password. Please try again with a valid username and password.”
7. In the *Password* box, enter testpassword001
8. Click *Login*
9. Verify that the on screen error message is “Your username was blank. Please try again with a valid username and password.”
10. In the *User* box, enter testuser001
11. In the *Password* box, enter thisisawrongpassword
12. Click *Login*
13. Verify that the on screen error message is “Either your username or password was incorrect. Please try again with a valid username and password.”
14. In the *User* box, enter thisisawrongusername
15. In the *Password* box, enter testpassword001
16. Click *Login*
17. Verify that the on screen error message is “Either your username or password was incorrect. Please try again with a valid username and password.”
18. In the *User* box, enter testuser001
19. In the *Password* box, enter testpassword001
20. Click *Login*
21. Verify that the user’s login page has been displayed.

The restated test script tests:

- Failure to enter either a username or password.
- Failure to enter a password, with a valid username.
- Entering a correct username, with an invalid password.
- Entering an incorrect username, with a valid password.
- Entering a valid username and password.

Of the five test cases, only the last was a positive test case (i.e., normal day-to-day user behavior). The first four were negative test cases, testing whether

the system copes gracefully with incorrect user behavior. Typically, most test cases will be negative.

Whilst the system should have had these negative test cases tested prior to handing over to acceptance testing, it is important to understand that only testing positive cases during acceptance testing is risky and potentially disastrous. In particular, the system could go into the live environment with incorrect error behavior. It might allow invalid data to be entered by users, without this invalidity having been detected at the data entry stage. It is an objective of acceptance testing to reveal whether the system's error behavior should be changed.

Creating Test Data: The Credit Card Example

A classic testing scenario is validation of credit card numbers. These numbers contain a "check" digit designed to test whether all other digits are correct. Systems accepting credit card numbers are supposed to run this validation check, prior to presenting the number to a bank's system for payment.

A more detailed discussion of the algorithm used for validating credit cards can be found here: http://www.beachnet.com/~hstiles/cardtype.html

A generator for example credit card numbers can be found here: http://www.darkcoding.net/credit-card-numbers/

Below is a table of data, designed to test whether a credit card entry field correctly validates test numbers:

Test Case	**Credit Card**	**Expiry Date**	**Expected Result**
Blank		01/16	Error: "The credit card number cannot be blank."
Too few digits	123	01/16	Error: "There are too few digits in your credit card number."
Too many digits	54561177287864350000	01/16	Error: "There are too many digits in your credit card number."
Non-number characters	abcdef	01/16	Error: "Your credit card number can only contain digits."
Invalid Visa	4829934954827093	01/16	Error: "This credit card number is invalid."
Valid Visa 13 digit	4556974184563	01/16	Moves to confirmation screen.
Valid Visa 16 digit	4929934954827093	01/16	Moves to confirmation screen.
Valid Visa, Invalid Expiry	4929934954827093	01/09	Error: "This card has expired."
Invalid MasterCard	5446117728786435	01/16	Error: "This credit card number is invalid."
Valid MasterCard	5456117728786435	01/16	Moves to confirmation screen.

The table does not document every possible permutation of data that the user could enter, however, it does test for these cases:

- Failure to enter a credit card number.
- Too few digits.
- Too many digits.
- Non-digit characters.
- An invalid Visa number.
- A valid 13-digit Visa number.
- A valid 16-digit Visa number.
- Entering an expiry date prior to the current date (the eighth case has a 2009 expiry date).
- An invalid MasterCard number.
- A valid MasterCard number.

Of the ten test cases, three are positive and seven are negative. The negative test cases are there to generate, at least once, the defined errors expected from the system.

Creating good test data is contextual, in terms of what the system is supposed to do. It is better that this data be designed by someone with good business knowledge, rather than relying on a third party. Typically, it is easier to document test data in a tabular / spreadsheet format, rather than listing it as a series of narrative steps in a test script.

The use of a test data table can be integrated into a test script by stating at a particular step "Enter the suite of test data found in spreadsheet X, validating that the system's output is as defined in the spreadsheet." The final column in the table notes expected output. When documenting expected output, it is only necessary to state what the user is expecting to see. There is no need to go into every last detail.

Creating an Acceptance Test Plan

Once the test scripts and test data are documented, these need to be aggregated together into an Acceptance Test Plan. This can be as simple as a small spreadsheet listing which test scripts are to be run in which order, with whichever test data.

If acceptance testing is expected to be intensive, given the sheer amount of work to be undertaken, it may help to figure out whether some or all of the tests can be run in parallel or a non-linear sequence. If this is possible, delegating among a group of people may speed up testing. If the test plan is documented as a spreadsheet, tabs can be created for each parallel stream.

At this stage, it may also help to have the test scripts and test data peer reviewed by someone within the same team. It is also useful to double-check whether test scripts have been defined for all of the system's functionality, to ensure that acceptance testing contains no gaps.

Raising a Bug Report

A bug occurs when the test system does not behave *as defined*. If during testing you discover a bug, you need to create a bug report, documenting at a minimum:

- The test system that the bug was found in (e.g., "uat001", "uat002"), if multiple acceptance test environments are available.
- The date and (preferably) time that the bug was found.
- The test script that was run.
- The step within the test script where the problem occurred.
- The test data used.
- The system's actual output, alongside what the output was expected to be.
- Whether the test was repeated to confirm the error.

After the bug report goes back to the development team, they will attempt to repeat the problem. If the problem is repeatable, the onus will be on them to resolve the issue. If it is not repeatable, investigation may be made of the acceptance test environment to determine whether the issue is intermittent, needing very specific conditions to replicate the issue.

The response to the bug report could be that the system is working as specified. This could result in needing to change your acceptance test scripts or test data, to reflect this fact. If the specified behavior is unacceptable in terms of deploying the system into the live environment, you need to raise a change request.

Raising A Change Request

If the system's behavior needs to be changed, you need to raise a change request, documenting at a minimum:

- Current behavior.
- Change to existing requirements.
- Priority. Whether this change is critical and immediate or can be deferred.
- Expected delivery date.

Raising a change request will almost certainly have budget implications. If large numbers of change requests need to be raised, it is likely that these will

be put together into “logical” groups, to ensure that changes to the same area of functionality happen at the same time.

Who Should Perform Acceptance Tests?

It is questionable practice to hire a third party to carry out acceptance testing. For a business unit commissioning a system, this is highly tempting, especially when enough money is available for bringing in the third party. The temptation is compounded when people in the business unit are in high-pressure jobs, with little room for other responsibilities.

This temptation may not only be short sighted and risky, it could also be expensive. Bringing in a third party requires that their personnel understand your business requirements to a level similar to an expert user within your business unit. This could prove to be a sharp and expensive learning curve, where people who know little or nothing about how you work will bill you for days of consultancy time, attempting to understand your environment. There is no guarantee that their understanding is correct, particularly if they have no educational or experience background in your business domain. For example, if the system to be tested runs complex actuarial calculations, there is no point asking someone to test the system who cannot understand the calculations.

Even in the short run, it could be far cheaper and less risky to release an expert user within the business unit to work only on acceptance testing. Typically, it will be much easier to train this person about the basics of testing, compared to bringing in a third party. Once this person is past the initial learning curve, they will become more productive on current and future projects. The expert user will be far more efficient at raising bug reports and negotiating changes to the system, given that they actually know what they are talking about and what the business unit wants.