3 TESTING BASICS FOR UAT

We know from the definition of UAT that the tests we design need to be formal so we need to know how to test formally, even though we will be aiming to generate simple tests based on our common-sense understanding of what the users' needs are. We will also need a range of different kinds of tests to cover all the aspects of UAT, so we will need a range of design techniques.

In this chapter we will build the testing toolkit that we will need so that we can use it effectively when we come to construct the tests for UAT. As we do this we will explain the key terms you need to be aware of and the basic processes, and then we will go on to build a few simple tests as a way of practising the skills.

Topics covered in this chapter

- What is testing?
- Test types
- Testing processes
- Test-case design techniques
- Testing approaches for UAT
- Reviews

WHAT IS TESTING?

What does the word 'testing' conjure up? It might be the image of the dummies used in testing cars to destruction to ensure they are safe to drive, or the kind of rudimentary checks we might do on a spreadsheet we built to use in the office, or the more rigorous testing that professional developers do to ensure their code works before they hand it over. None of these describes the kind of testing we need to do for UAT.

What we need for UAT is a 'formal' kind of testing that will enable us to check that a system does everything it is supposed to do and nothing that it is not supposed to do. We may be accepting something very expensive that should transform our business but, if there are any problems, could damage it. Testing of this kind is serious stuff; it needs to be robust, reliable and rigorous. That is why we opt for formal testing.

Formal testing

Testing is an objective and structured evaluation of a system (or a piece of software) against a standard. Evaluation must be objective rather than subjective so that we can be confident that the results of testing can be relied upon; testing must determine absolutely whether a piece of software meets its specification (the standard) or not. There is no room for personal opinion, judgement or guesswork. Testing needs to be structured so that we can be sure that we have evaluated the system against the requirements completely, in the sense that every requirement has been tested at least once. A structured approach enables us to ensure that every requirement has at least one test.

Also, if we assume that any defects and deficiencies discovered during testing will need to be corrected, testing must also provide objective evidence of what is wrong with the system so that the development team can quickly and accurately identify the sources of defects or deficiencies and correct them. This is another aspect of a structured evaluation.

Testing contrasts with activities like debugging, which are designed to find and eliminate defects during the development phase. In debugging a developer picks inputs that they think might break the system so that they can remove the defects. In effect, because the developer is both the expert and the fixer of bugs at this stage, they can do testing and bug fixes themselves. In UAT, as in all formal testing, the testing and bug-fixing activities are deliberately kept separate. The end-user can find but not fix errors, and the developers can fix errors but not carry out the testing.

The primary objective of UAT is to tell us what the status of a system or a piece of software is, in particular (in our case) whether it is fit for release. To do this testing must be:

- systematic, so we know exactly what has been tested and what has not;
- derived from the specification of what the system is supposed to do, so we can ensure everything that needs to be tested actually gets tested;
- repeatable, so that if we find defects we know the development team can repeat our test and get the same results to help them locate the defect;
- documented, so we can evaluate the quality of the tests.

Testing comes in many shapes and sizes, but all formal testing conforms to these same criteria. These are the characteristics UAT has in common with other types of software testing, but UAT is also unique in some respects.

Test levels

In Chapter 2 we looked at a life cycle for software development called the V life cycle. In that life cycle we have several stages of testing, which are called test levels (because they test software as the system is built up, level by level, from individual units of code). Each test level has its own unique set of objectives.

For example at the unit test level we would look for evidence that the unit functions correctly, at the integration test level we would look for evidence that units can communicate and function together, and at the system test level we would look for evidence that the whole system works as specified.

Each test level is a formal test based on the specification at that level, so unit testing would be based on the unit specifications and so on. The relevant specifications are therefore called the test basis for that level.

At every test level we may have many different aspects of the software or system to consider such as functionality, reliability, usability, security and so on. Each aspect of the system that we test will need its own specific test type. Functional testing is designed to test that functionality at a given level is correct with respect to the test basis. Security testing is designed to test that security requirements have been met at a given level and so on.

UAT is the highest test level because it represents the level at which requirements were formulated. Every level below UAT should have been tested before UAT begins, so UAT is the final check that what has been built meets the requirements of the end-users and sponsor.

TEST TYPES

Test type

A group of test activities aimed at testing a component or system focused on a specific test objective, which means functional test, usability test, regression test and so on. A test type may take place on one or more test levels or test phases.

Of all the test types that might be defined, two are particularly useful for UAT: functional testing and structural testing.

Functional testing

Functional testing is exactly what it says: testing that required functions are present and that they work correctly.

Functional testing

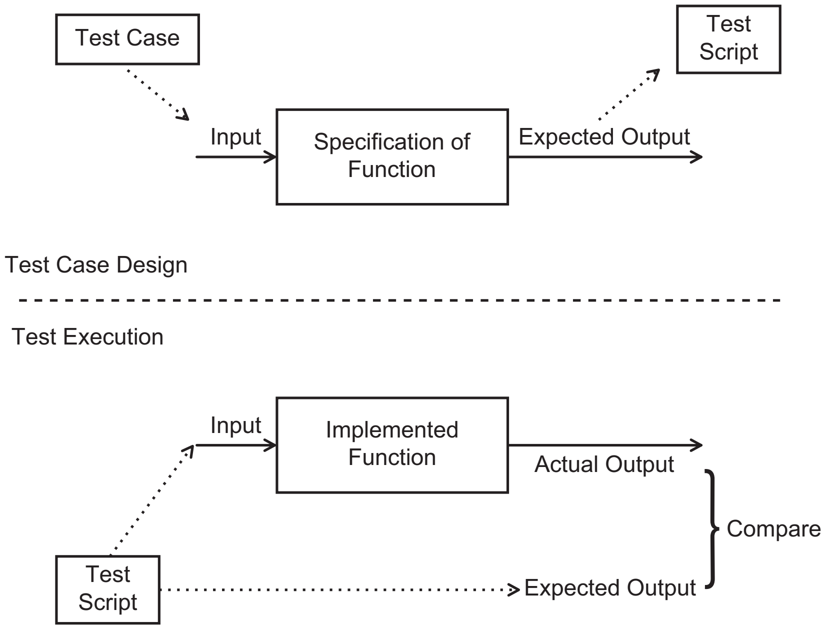
Testing based on an analysis of the specification of the functionality of a component or system. This is also commonly known as black-box testing.

Functional test design technique

A procedure to derive or select test cases based on an analysis of the specification of the functionality of a component or system without reference to its internal structure.

We have to analyse the business requirements to identify all the functional requirements, define test conditions and generate test cases for them. Figure 3.1 shows what a functional test looks like.

Figure 3.1 Functional testing



In Figure 3.1 a functional test case has been created from a test condition. The input data specified in the test case are then applied to the specification of the function to be tested (in the business requirements or in a more detailed specification if one exists) to discover what output(s) that input should generate. These are all recorded in the test script.

When the test is executed the test script is used to identify the required data input and this time the actual output(s) are recorded in the test script. The actual and expected outputs can then be compared to determine whether the test has passed or failed.

The vast majority of test scripts generated for UAT will be functional.

Structural or white-box testing

Structural testing

Structural (white-box) testing is based on an analysis of the internal structure of the component or system.

White-box testing is most often associated with testing code, but it is useful for testing anything that has internal structure. In UAT we need to test system components such as menu structures and we also need to ensure that the system operates correctly with business processes. We can utilise white-box testing in these cases to explore the relevant structures and to ensure that we exercise all of the possible routes through the structure (a form of (structural) test coverage).

If we take a business process as an example, we know that each path through the business process will trigger one or more use cases that will exercise system functions and generate outputs. The business process may then utilise the outputs in some way.

This is an example of end-to-end testing if we initiate the business process to generate inputs to the system and consume outputs from the system. Successful completion of the business process indicates a successful end-to-end test case and within that test case a functional test case will have been utilised to transform data.

Example 3.1

Figure 3.2 shows a simple business process that has three paths corresponding to the responses to the two questions embedded in the process. If the answer to 'Existing Customer?' is 'Yes', the process moves to the next question 'New Account Type?' If this too is a 'Yes', the process requires two actions: 'Generate Authority' and 'Send to Customer' (path 3). Each of these activities will have a use case from which we can generate functional test cases. So an end-to-end test of path 3 involves a user operating a business process using the system and completing that process via two test cases and probably a manual activity to generate a mailing to a customer. The process flows from its beginning to its end.

If we follow the same testing approach for paths 1 and 2, we will have completed the entire business process; that is we will have achieved 100 per cent structural coverage of the business process, utilising some existing test cases along the way.

A LITTLE QUALITY EXPERIMENT

Imagine we have a system with 100 requirements and we give three testers the task of testing it as well as they can. Tester 1 runs 100 tests, tester 2 runs 200 tests and tester 3 runs 1,000 tests. Which tester did the most effective testing?

The answer is that we cannot tell. Although tester 3 ran the most tests, we cannot be sure what was tested so we do not actually know how effective the tests were.

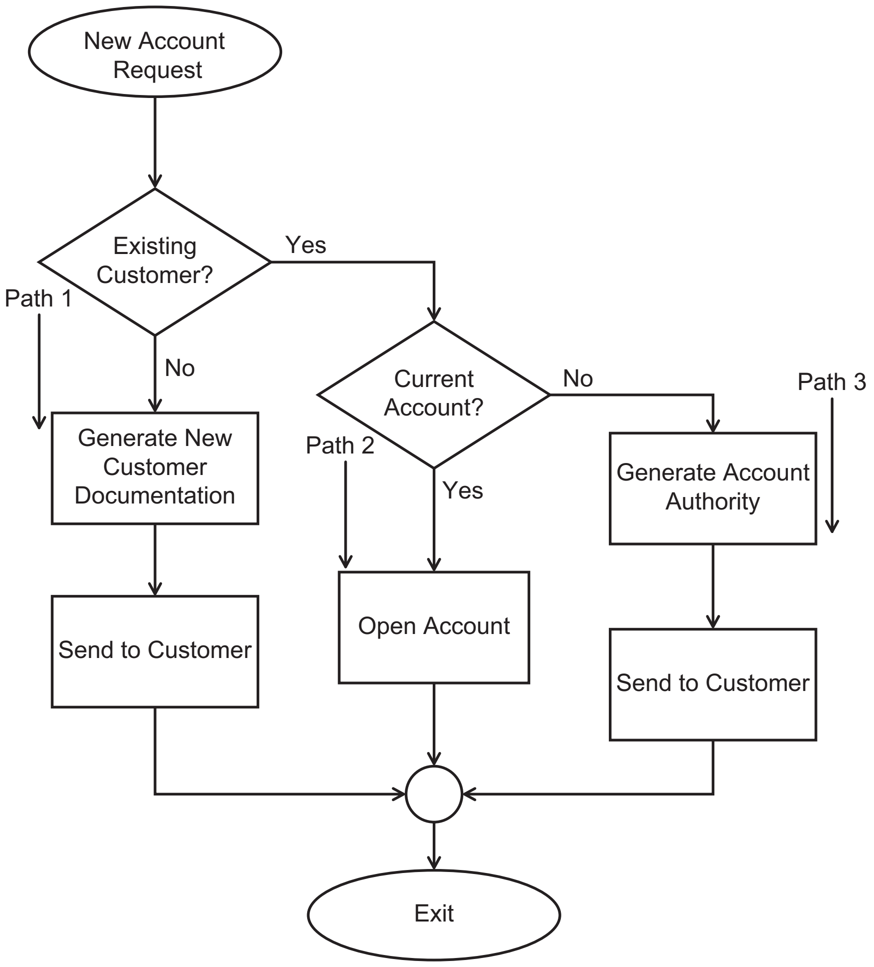
If we specified that every requirement must be tested at least once and we repeated the exercise, the results would be rather different. Tester 1 runs 100 tests, tester 2 runs 200 tests and tester 3 runs 1,000 tests. There is no change except that all three testers now claim that all requirements have been tested. What can we conclude?

If tester 1 has covered every requirement, then that set of tests must be more efficient than the other two testers' tests because it achieved the same result with less effort. But we still do not know whether the three sets of tests were equally effective. The moral of the story is that more does not necessarily mean better and test effectiveness needs to be based on some other criterion. That criterion is called coverage.

Coverage is a measure of exactly what parts of a system are tested by each individual test. If we can demonstrate (this will need some documentation) that our tests have been designed so that every requirement has been tested at least once, then we have achieved 100 per cent requirements coverage and the smallest set of tests that achieves 100 per cent coverage is the most efficient.

Incidentally the documentation of the tests also enables us to check that the tests work. Around 10 per cent of test failures are typically caused by defects in the test rather than defects in the system, so it is worth checking the tests before we run them.

Figure 3.2 Business process end-to-end testing



In UAT we look especially for business requirements coverage. If we specify 100 per cent test coverage, we mean that every single requirement has one or more tests defined and when those tests are executed we can be certain that 100 per cent of the requirements have been tested. We will later introduce test-case design techniques that enable us to achieve desired levels of test coverage and also test our system efficiently, in the sense that we will use the minimum of time and resources to achieve a given level of test coverage – the Holy Grail of testing.

Bear in mind, though, that one test may not be enough to test a requirement completely, so 100 per cent requirements coverage may mean that every requirement has been subjected to at least one test. If we want to be more thorough (perhaps for the more critical requirements) we may have to design more than one test for some requirements and adjust our coverage measure accordingly.

One question that we will have to wrestle with continuously is, ‘What exactly are the business requirements?’ We need a test basis for UAT and we know that the initial RS is unlikely to be an adequate test basis. We will return to address that problem when we come to planning the UAT exercise.

TESTING PROCESSES

We want to be able to complete UAT in the most efficient and timely way we can. To do that we need two key processes: the FTP, which will help us to ensure we do the right things at the right time, and the test development process, which will be used to ensure we design the right kinds of tests so we get a clear answer to whether the system meets the business requirements and the acceptance criteria.

The fundamental test process

As we defined in Chapter 1, the fundamental test process (FTP) has five steps:

1. test planning, monitoring and control;
2. test analysis and design;
3. test implementation and execution;
4. evaluating exit criteria and reporting;
5. test closure activities.

We will use this simple process to structure the step-by-step guidance in Chapters 6–10. It gives us the sequence of activities we will need to follow to achieve our objective and also provides useful pointers to the kinds of outputs we might need to enable us to make a rational decision about acceptance at the end.

The test development process

The test development process (TDP) describes the mechanism for generating effective tests that achieve a required level of test coverage. The process comes in three stages, each of which is aligned with a step in the FTP.

The TDP has three components: test conditions, tests cases and test scripts. These are all defined in the ISTQB Glossary and reproduced in the box below.

Test condition

An item or event of a component or system that could be verified by one or more test cases, e.g. a function, transaction, feature, quality attribute, or structural element.

Test case

A set of input values, execution preconditions, expected results and execution post-conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.

Test procedure specification

A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script.

Test condition

The purpose of a test condition is to express some aspect of the business requirements (a function, transaction, feature, quality attribute or structural element) in a form from which a specific test or tests can be constructed. A test condition can be either **true** or **false** and the value of the test condition can be determined by running a test case.

Example 3.2
If a feature describes secure login to a system using a username and password, there are a number of test conditions that can be written:

1. If a valid username is entered with the correct password, the user is logged in to the system.
2. If a valid username is entered with an incorrect password, an error message appears.
3. If a non-valid username is entered with a password, an error message appears.

There are other test conditions that could be written in addition to the above three, depending on the requirements of the business.

Notice that each test condition represents a single component of the feature that can be assessed as either true or false. The feature is correctly implemented if and only if all three conditions are true.

For each test condition we could construct a table to identify what the test condition means as follows.

Table 3.1 Test conditions table

Valid username	TRUE	TRUE	FALSE	FALSE
Correct password	TRUE	FALSE	TRUE	FALSE
User logged in	TRUE	FALSE	FALSE	FALSE

Each entry in the table can now be tested with a test case that evaluates to TRUE or FALSE.

Test case

We need to identify inputs and expected outputs associated with each test condition so that we can generate specific tests to determine whether the test condition is true.

Preconditions and post-conditions identify the state the system must be in before the test is executed and the state it will be in after test execution respectively.

One or more test cases can be written for each test condition.

Example 3.3
Here is a set of test cases written for the first test condition from Example 3.2.

Test condition 1: If a valid username is entered with the correct password, the user is logged into the system.

Table 3.2 Test cases

Test Case 1	Precondition	User not logged in
	Inputs	Valid username
		Valid password
	Outputs	None
	Post-condition	User logged in
Test Case 2	Precondition	User not logged in
	Inputs	Valid username
		Non-valid password
	Outputs	Error message
	Post-condition	User not logged in
Test Case 3	Precondition	User not logged in
	Inputs	Non-valid username
		Valid password
	Outputs	Error message
	Post-condition	User not logged in
Test Case 4	Precondition	User not logged in
	Inputs	Non-valid username
		Non-valid password
	Outputs	Error message
	Post-condition	User not logged in

Preconditions and post-conditions are useful for sequencing tests, for example if a precondition of a test is that the user is already logged in to the system, it would make sense to ensure it is not run before the test case concerned with logging in to the system.

When a list of all the possible test conditions is written, it may be that some of the test conditions are duplicates. If the outcome (logging in) is dependent on there being both a valid username and a matching valid password, then you could argue that test case 3 and 4 are duplicates because a non-valid username will not allow the end-user to log in, regardless of whether they have a valid or non-valid password. In fact you could argue that a non-valid username cannot have a valid password. However, in our example we want to test that the combinations of the different valid and non-valid entries produce the required outcome. For test case 3 we can enter a valid username that contains an error with the corresponding valid password. Test case 4 can test the equivalent of a hacker trying to gain access to the system by entering a random invalid username and password.

Test script (test procedure specification)

Test scripts provide the mechanism and documentation to run the tests we have defined as test cases. A test case is generic; it defines the nature of input and expected outputs, and provides a template for constructing test scripts with actual data. UA testers need to be armed with a collection of test scripts that they execute in a particular sequence.

In Example 3.2 we have defined a template for successful logins. We could use that template to generate a test script that, for example, logs in 10 valid users (particularly useful if we have a way of automating the input). More interesting, though, is that we could define a test script to exercise test cases associated with all four test conditions and then run that script with a mixture of valid and non-valid usernames and valid and non-valid passwords.

Example 3.4
Here is a simple test script that corresponds to test case 1 above.

Table 3.3 Test script

Login1: Normal user login

Purpose	Users are able to log in with an acceptable user ID and password
Preconditions	User is not logged into the system. Test account has been set up successfully
Test data	User ID: tester1@acme.com
	Password: UAT1
Process steps	1. Click system icon
	2. Enter user ID
	3. Enter password
	4. Click Login

(Continued)

Table 3.3 (Continued)

Login1: Normal user login	
Result	User is logged into the system on the Home page
Post test	User tester1@acme.com is logged in
Notes	

Note that this simple test script can be extended to log in multiple users and extended to include the other test cases.

Example 3.5 – A more extensive test script
This is a test script to test a specific test case uniquely identified as test case 4.11.

4.11 Test script 17 – Check the functionality on the login screen

4.11.1 Test conditions:

1. Check end-users can log on to the system with a valid user ID and password.
2. Check end-users cannot log on to the system with an invalid user ID.
3. Check end-users cannot log on to the system with an invalid password.

Test # 1 – Check the end-user can log on to the system with a valid user ID and password.

Test step	**Test description**	**Expected results**	**Pass/Fail comments and observation numbers raised**
Scenario 1 – Logged out of the system on the start menu This scenario tests: • A valid user ID and password. Expected results: • The end-user is successfully logged on to the system.			
1.	Click the system icon	The login page is displayed. The following fields should be displayed and should be blank: • Username • Password	

(Continued)

Test step	Test description	Expected results	Pass/Fail comments and observation numbers raised
2.	Enter tester1@acme.com in the User ID field	The username should be displayed in the field	
3.	Enter UAT1 in the password field	A * character should be displayed for each character of the password	
4.	Click the login button	The homepage should appear and the username should be displayed in the top right corner of the page	

Note that the terms used in our (real) example are different from the terminology used in this book. In the example the test script is what we have referred to as a scenario and vice versa. This is a very common occurrence in UAT and as long as all the stages that need to be delivered are understood and the term universally used, it is less important that those stages are known by a different name in your organisation. Note also that the script only has three test conditions because in this instance the condition that tests a non-valid username and non-valid password was deemed unnecessary.

Each script contains: test-case number and version, test description, requirement number, tester, process step numbers, process step descriptions, test data to be utilised, expected results, error descriptions, pass/fail results, date tested and comments from the UA tester. The following are the elements that should be included in addition to the above in a scenario containing multiple test scripts:

- scenario name and number;
- scenario description;
- test script names, numbers, versions and dates;
- IDs of the test cases and requirements covered by the test script;
- description of the test cases;
- any prerequisite procedures.

Numbering test scripts is part of the traceability of the test documentation. It serves to measure how many test scripts, test cases, requirements and acceptance criteria have been covered during UAT. Test scripts can also be reused with updated version numbers and dates for the purposes of regression testing.

Test coverage

As we explained earlier, test coverage is an important measure of how much testing has been done.

If a requirement generates 50 test conditions, then those 50 test conditions need to be evaluated to achieve 100 per cent coverage. If 100 tests are run but only one of them relates to a test condition, then only two per cent (1 out of 50) coverage has been achieved. The way test coverage is measured differs from one test-case design technique to another, but in every case the test coverage measure compares the number of tests carried out so far with the number of tests that are possible on the software under test using the given technique. Test coverage is our objective measure of how much testing has been done and therefore how comprehensive the UAT's outcomes are.

A UA test design plan requires a collection of test conditions to be extracted from the requirements to achieve whatever level of test coverage has been specified for the tests. From these a set of test cases is generated that defines all the inputs and expected outputs for all of the tests, and finally a set of test scripts is created that will enable UA testers to execute all the tests and record the results.

Example 3.6
For the login example (3.2) above, based on the assumption that conditions 3 and 4 are not duplicates, the test condition generates four test cases, all of which are needed to completely evaluate the test condition. Each of the four tests would carry 25 per cent of the coverage for that particular requirement and all four would need to be run to provide 100 per cent coverage of the requirement.

TEST-CASE DESIGN TECHNIQUES

Test-case design techniques exist for a wide variety of different approaches to testing a software object, many of which are described in Hambling et al. (2010). Any of these can be used to design test cases and some will certainly improve overall productivity, especially if they are easy to understand and use.

Equivalence partitioning (EP) and boundary value analysis (BVA)

EP and BVA are two related test-case design techniques that are invaluable in the UAT context. Despite the long and technical-sounding names these are very simple and intuitive techniques to use, so it is well worth getting familiar with them.

EP helps to understand what values should be used when data values are in a continuous range, such as all the integers between 1 and 10. It provides a simple way to give us confidence that the system correctly handles all the values within the range, so it gives us efficient yet complete testing.

Here is a simple example. If a data entry field for a surname can contain up to 50 alphabetic characters, there is no benefit in testing all the different combinations of alphabetic characters that can make up a name. We could not even if we wanted to – the number of possible combinations of 50 alphabetic characters is astronomical, and that only accounts for the valid examples.

EP allows us to assume that any valid combination will behave like any other valid combination – so we only need to test one example to give us reasonable confidence that the system will accept any valid combination of characters. We call that very large collection of possible inputs the valid partition. Within this partition any valid combination of characters will demonstrate that data entry will work and therefore represents one valid test case. In this equivalence class, entering 25 valid characters would be as valid a test as entering any other valid combination of characters fewer than 51 characters long, and only one would need to be tested. Note that we have chosen to use the middle of the partition for our valid case in this example.

If there is a valid partition there will also be at least one invalid partition. In fact there are three different invalid partitions that we have to take account of in this case:

1. any combination of 1–50 characters that contains at least one invalid character (for example a numeric character);
2. combinations that contain fewer than one character (which means no characters);
3. combinations that contain more than 50 valid characters.

We will need one example of each of those to make up a complete set of tests of the equivalence partitions.

You need to be aware that EP only saves us time and effort when the inputs are structured into lists, sequences or other collections. It is a limitation, but not that much of a limitation because input data fields are usually structured.

BVA is a natural partner for EP. BVA takes account of the observed characteristic that more application errors occur at boundaries or edges of data than anywhere else. Developers commonly process structured data with loops, and an error in entering or terminating a loop will cause a problem at a boundary (for example the first or last item to be processed in a sequence). It is also true that ‘edge cases’ are those we often get wrong in business processes, so developers may be working from an incorrectly specified boundary value to begin with. For these reasons boundary cases are great places to go looking for problems.

BVA exploits these observations by clustering tests at boundaries, and one good thing about partitions is that they invariably have boundaries. In our EP example above the boundaries are combinations of 1 character and 50 characters. For our purposes in UAT we can test the actual boundary values and one value just outside each boundary. The term ‘just outside’ means outside by the smallest possible amount in the situation. In our case the character combinations can only increase or decrease in length by one, so ‘just outside’ is one outside.

Applying BVA to our example would then lead to the following test cases:

1. two values at the boundaries for the positive values: 1 character and 50 characters;
2. two values just outside the boundaries: 0 characters and 51 characters.

You may recall that we already selected a test case with zero characters so we do not need to repeat that. Now the notion of using a value in the middle of the partition as the valid value makes sense; we are going to test at the edges anyway.

We have now whittled down an astronomical set of possible tests to just four tests: an input of 25 valid characters, an input of 25 characters with one or more of them invalid, an empty input field and an input of 51 valid characters. This saving of effort comes with no compromise on the quality of the testing.

There are many other test-case design techniques that you can explore, but EP and BVA on their own will give us plenty of support for the kind of tests we will need to run.

TESTING APPROACHES FOR UAT

We mentioned earlier that UAT is unique in some respects. The uniqueness arises from the U in UAT. UAT is uniquely driven by end-users (or those who will become end-users of the system once it is implemented). In other words the testers are not expected to be software or testing professionals and there is no expectation that they will have any experience of testing. In fact the reverse is true; the end-users' distance from the development specialism and relative closeness to the business specialism give them a unique perspective that is not influenced by what was built or how it was built. Only end-users can be objective in this context and, since they will ultimately operate the system, any intuition or experience they can bring to the exercise will add value.

One other thing makes UAT unique. Unlike all other kinds of testing, which are based on testing outcomes against a specification, UAT is based on three elements:

1. business requirements;
2. business processes;
3. user expectations.

We could argue that business requirements are documented in a specification (the RS) but, as we have already seen, the RS is not necessarily a valid basis for testing. We might also argue that business processes are specified, but that is not always the case. User expectations are not only not documented, they are also subject to change as the users gain experience.

So we need an approach to testing that mirrors these three elements.

Requirements-based test cases

Test cases must cover the business requirements because the requirements are what UAT sets out to test. In order to show that the requirements-based test cases relate to a specific requirement, the test case should contain an ID that links it to a business requirement. There is one layer of complexity that we should add at this point to do with the timing of test design. So far we have made the assumption that the test cases are created at the end of the project as part of the UAT preparation. However, you may find that the test cases were written shortly after the RS and that the existing test cases are out of date and must be updated according to the newly reviewed RS. The disadvantage of using requirements-driven test cases is therefore that if the requirements contain mistakes, the test cases will also be wrong.

Business process-based test cases

Business process-based test cases are test cases written to help make sure that the system that is delivered will work specifically in supporting the business processes. The test cases must be able to show that the requirements have been met in a way that reflects how the organisation is going to use the system. For business process-based testing, the tests must be sequenced to reflect the processes in order to check that they reflect the paths through those processes.

We showed in Example 3.1 how structure-based testing can be used to achieve coverage of business processes.

User interface-driven test cases

User interface-driven test cases are structured around forms or screens that need to be completed. Test cases are based on data entry, interactions via the screen and reporting. In each case these will be related through a scenario so that data are manipulated in a realistic way. User interface-based test cases can be embedded within business process-based test cases where the business process involves data entry, interaction or reporting. User interface testing might include:

1. Tab order – is the tab order correct?
2. Required fields – are the required fields marked and is an entry required?
3. Data-type errors – can the data only be entered in the correct format (dates, numeric, currency)?
4. Save and delete confirmations – does the system prompt the user to save changed data before closing and confirm a delete?
5. Shortcuts – do the shortcuts work?
6. Invalid menu items – are any menu items shown that are not available for the context users are currently in?
7. Links – do links work?
8. Menus – do the correct menu items appear?

Setting priorities – risk-based testing

One final aspect of testing for UAT is that we will be doing it under time pressure because we do our testing just before the system is released. If the system is late reaching the UAT stage we will be under even more pressure. In recognition of that we need a way of doing the best we can with the limited time available. For this we use a prioritisation mechanism that ensures we run the most important tests first, so that any testing that cannot be completed is guaranteed to be less important than the testing we have already done. We call this risk-based testing.

If we have an understanding of what aspects of the system are most important and might cause a serious problem if they were not effective, that is they represent risk, we can prioritise tests by risk level and test the highest risk areas first. We can begin by identifying the risk level for each requirement or group of requirements and putting them in priority order. That is at least a practical starting point for developing test cases. Risk-based testing can be used in conjunction with other approaches. For example we can apply requirements-based testing as a way of achieving requirements coverage and then apply risk-based testing within the requirements-based tests to ensure we test the most important areas first.

One final thought on test approach

Although we are interested in whether the software works or not, the developers and testers will already have tested this; specifically whether the software performs according to the technical specification. Our concern is not about compliance with the technical specification. If the system does what we want and need it to do while missing some detailed part of the technical specification then that will be an outcome that we should report, but it may not be something that, for us, is a 'show-stopper'. If, on the other hand, the system met every single item of the technical specification and the RS, but we found it to be cumbersome in use or it failed to support key business processes, then we would have cause for concern.

That is the nature of the end-user perception. We test to ensure we get what we need, even if that is not what was specified. The customer is always right.

REVIEWS

One further testing technique that we will need to be familiar with is the technique of reviewing.

Review

A review is an evaluation of a product or project status to ascertain discrepancies from planned results and to recommend improvements. Examples include management review, informal review, technical review, inspection and walk-through.

As the definition indicates, there are many kinds of review but the purpose is always the same: to evaluate and improve. We will use reviews as a key tool for evaluating and improving RS and test conditions, test cases and test scripts. We will concentrate on a very simple and informal kind of review, a variant of the walk-through mentioned in the definition, which will use the following basic process:

- Read the document individually and identify any problems or questions.
- Meet with the author of the document and other reviewers so that the author can 'walk through' the document and answer any questions.
- Capture any unresolved questions or problems in a document to give to the author after the review.

The evaluation is achieved by each reviewer contributing comments to enable the author to identify things that need to be changed or corrected; the improvements come from the changes the author will make following the review meeting.

Reviews provide benefits to a team over and above the improvements to documents that can be made. For example:

- Reviews are a team event so they help with the process of team building and getting to know each other.
- All participants are aware of the status of the reviewed document and the improvements made so reviews help to keep everyone involved and informed.
- The opportunity to ask questions of the document's author offers learning opportunities so that team members can build their knowledge of the system and the tests.
- Responsibility for the quality of the reviewed document is shared so everyone has a stake in quality.
- There is an opportunity to learn from more experienced colleagues so knowledge and experience can be pooled effectively in the team.

We will introduce a walk-through into the planning stage of UAT preparation by reviewing the RS in Chapter 6.

CHAPTER SUMMARY

This chapter has introduced some key testing processes and techniques for generating effective test cases, demonstrating how they can be adapted and focused on the specific needs of UAT.

Reviews in general, and the walk-through technique in particular, have been introduced as a valuable technique for evaluating and improving documents and also contributing to the team's development.

After reading this chapter you should be able to answer the following questions:

- What steps do I have to follow to ensure my UAT is complete?
- What steps do I have to follow to build an effective set of tests for UAT?
- How can I use reviews to ensure the documentation I have acquired is fit for purpose as a test basis?
- What techniques are available and how should I use them?

What have you learned?

Test your knowledge of Chapter 3 by answering the following questions. The correct answers can be found in Appendix B.

1. Which of the following best describes test coverage?

   A. Test coverage is the ratio of functions tested to the total functions
   B. Test coverage is a count of the number of tests that have been run
   C. Test coverage is a measure of the number of different things tested by a test
   D. Test coverage is the scope of testing

2. Which of the following is a benefit of using reviews to evaluate documents?

   A. Reviews give testers the opportunity to criticise developers' work
   B. Reviews are inexpensive to perform
   C. Reviews find all the defects in a document
   D. Reviews encourage teams to work together

3. What does the BVA test-case design technique test?

   A. BVA tests that users cannot enter non-valid data
   B. BVA tests that a system recognises whether data are within a specified range
   C. BVA analyses how the system performs when it is subjected to extreme conditions
   D. BVA generates test cases that should all fail

Some questions to consider (our responses are in Appendix B)

1. Your organisation is reluctant to allow UA testers to be part of a review process. What would be the best way to overcome that reluctance?
2. You are being offered a training course before starting work as a UA tester. What would be your requirements for a one-day course? Suppose you could have three days of training. What changes would you make to your requirements?