CHAPTER FOUR

Test design techniques

Chapter 3 covered static testing, looking at documents and code, but not running the code we are interested in. This chapter looks at dynamic testing, where the software we are interested in is run by executing tests on the running code.

4.1 IDENTIFYING TEST CONDITIONS AND DESIGNING TEST CASES

1 **Differentiate between a test design specification, a test case specification and a test procedure specification. (Kl)**

2 **Compare the terms test condition, test case and test procedure. (K2)**

3 **Write test cases: (K3)**

a showing a clear traceability to the requirements; b containing an expected result.

4 **Translate test cases into a well-structured test procedure specification at a level of detail relevant to the knowledge of the testers. (K3)**

5 **Write a test execution schedule for a given set of test cases, considering prioritization, and technical and logical dependencies. (K3)**

4.1.1 Introduction

Before we can actually execute a test, we need to know what we are trying to test, the inputs, the results that should be produced by those inputs, and how we actually get ready for and run the tests.

In this section we are looking at three things: test conditions, test cases and test procedures (or scripts) - they are described in the sections below. Each is specified in its own document, according to the Test Documentation Standard [IEEE829].

Test conditions are documented in a Test Design Specification. We will look at how to choose test conditions and prioritize them.

Test cases are documented in a Test Case Specification. We will look at how to write a good test case, showing clear traceability to the test basis (e.g. the requirement specification) as well as to test conditions.

Test procedures are documented (as you may expect) in a Test Procedure Specification (also known as a test script or a manual test script). We will look at how to translate test cases into test procedures relevant to the knowledge of the tester who will be executing the test, and we will look at how to produce a test execution schedule, using prioritization and technical and logical dependencies.

In this section, look for the definitions of the glossary terms: **test case, test case specification, test condition, test data, test procedure specification, test script** and **traceability.**

4.1.2 Formality of test documentation

Testing may be performed with varying degrees of formality. Very formal testing would have extensive documentation which is well controlled, and would expect the documented detail of the tests to include the exact and specific input and expected outcome of the test. Very informal testing may have no documentation at all, or only notes kept by individual testers, but we'd still expect the testers to have in their minds and notes some idea of what they intended to test and what they expected the outcome to be. Most people are probably somewhere in between! The right level of formality for you depends on your context: a commercial safety-critical application has very different needs than a one-off application to be used by only a few people for a short time.

The level of formality is also influenced by your organization - its culture, the people working there, how mature the development process is, how mature the testing process is, etc. The thoroughness of your test documentation may also depend on your time constraints; under excessive deadline pressure, keeping good documentation may be compromised.

In this chapter we will describe a fairly formal documentation style. If this is not appropriate for you, you might adopt a less formal approach, but you will be aware of how to increase formality if you need to.

4.1.3 Test analysis: identifying test conditions

Test analysis is the process of looking at something that can be used to derive test information. This basis for the tests is called the 'test basis'. It could be a system requirement, a technical specification, the code itself (for structural testing), or a business process. Sometimes tests can be based on an experienced user's knowledge of the system, which may not be documented. The test basis includes whatever the tests are based on. This was also discussed in Chapter 1. From a testing perspective, we look at the test basis in order to see what could be tested - these are the test conditions. A **test condition** is simply something that we could test. If we are looking to measure coverage of code decisions (branches), then the test basis would be the code itself, and the list of test conditions would be the decision outcomes (True and False). If we have a requirements specification, the table of contents can be our initial list of test conditions.

A good way to understand requirements better is to try to define tests to meet those requirements, as pointed out by [Hetzel, 1988].

For example, if we are testing a customer management and marketing system for a mobile phone company, we might have test conditions that are related to a marketing campaign, such as age of customer (pre-teen, teenager, young adult, mature), gender, postcode or zip code, and purchasing preference (pay-as-you-go or contract). A particular advertising campaign could be aimed at male teenaged customers in the mid-west of the USA on pay-as-you-go, for example.

Testing experts use different names to represent the basic idea of 'a list of things that we could test'. For example, Marick refers to 'test requirements' as things that should be tested. Although it is not intended to imply that everything must be tested, it is too easily interpreted in that way. [Marick, 1994] In contrast, Hutcheson talks about the 'test inventory' as a list of things that could be tested [Hutcheson, 2003]; Craig talks about 'test objectives' as broad categories of things to test and 'test inventories' as the actual list of things that need to be tested [Craig, 2002]. These authors are all referring to what the ISTQB glossary calls a test condition.

When identifying test conditions, we want to 'throw the net wide' to identify as many as we can, and then we will start being selective about which ones to take forward to develop in more detail and combine into test cases. We could call them 'test possibilities'.

In Chapter 1 we explained that testing everything is known as exhaustive testing (defined as exercising every combination of inputs and preconditions) and we demonstrated that this is an impractical goal. Therefore, as we cannot test everything, we have to select a subset of all possible tests. In practice the subset we select may be a very small subset and yet it has to have a high probability of finding most of the defects in a system. We need some intelligent thought processes to guide our selection; **test techniques** (i.e. test design techniques) are such thought processes.

A testing technique helps us select a good set of tests from the total number of all possible tests for a given system. Different techniques offer different ways of looking at the software under test, possibly challenging assumptions made about it. Each technique provides a set of rules or guidelines for the tester to follow in identifying test conditions and test cases. Techniques are described in detail later in this chapter.

The test conditions that are chosen will depend on the test strategy or detailed test approach. For example, they might be based on risk, models of the system, likely failures, compliance requirements, expert advice or heuristics. The word 'heuristic' comes from the same Greek root as *eureka*, which means 'I find'. A heuristic is a way of directing your attention, a common sense rule useful in solving a problem.

Test conditions should be able to be linked back to their sources in the test basis - this is called **traceability.**

Traceability can be either horizontal through all the test documentation for a given test level (e.g. system testing, from test conditions through test cases to test scripts) or vertical through the layers of development documentation (e.g. from requirements to components).

Why is traceability important? Consider these examples:

- The requirements for a given function or feature have changed. Some of the fields now have different ranges that can be entered. Which tests were looking at those boundaries? They now need to be changed. How many tests will actually be affected by this change in the requirements? These questions can be answered easily if the requirements can easily be traced to the tests.
- A set of tests that has run OK in the past has started to have serious prob lems. What functionality do these tests actually exercise? Traceability between the tests and the requirement being tested enables the functions or features affected to be identified more easily.
- Before delivering a new release, we want to know whether or not we have tested all of the specified requirements in the requirements specification. We have the list of the tests that have passed - was every requirement tested?

Having identified a list of test conditions, it is important to prioritize them, so that the most important test conditions are identified (before a lot of time is spent in designing test cases based on them). It is a good idea to try and think of twice as many test conditions as you need - then you can throw away the less important ones, and you will have a much better set of test conditions!

Note that spending some extra time now, while identifying test conditions, doesn't take very long, as we are only listing things that we could test. This is a good investment of our time - we don't want to spend time implementing poor tests!

Test conditions can be identified for **test data** as well as for test inputs and test outcomes, for example, different types of record, different distribution of types of record within a file or database, different sizes of records or fields in a record. The test data should be designed to represent the most important types of data, i.e. the most important data conditions.

Test conditions are documented in the IEEE 829 document called a Test Design Specification, shown below. (This document could have been called a Test Condition Specification, as the contents referred to in the standard are actually test conditions.)

IEEE 829 STANDARD:
TEST DESIGN SPECIFICATION TEMPLATE

Test design specification identifier

Features to be tested Approach

refinements Test identification Feature

pass/fail criteria

4.1.4 Test design: specifying test cases

Test conditions can be rather vague, covering quite a large range of possibilities as we saw with our mobile phone company example (e.g. a teenager in the mid-west), or a test condition may be more specific (e.g. a particular male customer on pay-as-you-go with less than $10 credit). However when we come to make a test case, we are required to be very specific; in fact we now need exact and

detailed specific inputs, not general descriptions (e.g. Jim Green, age 17, living in Grand Rapids, Michigan, with credit of $8.64, expected result: add to Q4 marketing campaign). Note that one **test case** covers a number of conditions (teenager, male, mid-west area, pay-as-you-go, and credit of less than $10).

For a test condition of 'an existing customer', the test case input needs to be 'Jim Green' where Jim Green already exists on the customer database, or part of this test would be to set up a database record for Jim Green.

A test case needs to have input values, of course, but just having some values to input to the system is not a test! If you don't know what the system is supposed to do with the inputs, you can't tell whether your test has passed or failed.

Should these detailed test cases be written down? They can be formally documented, as we will describe below. However, it is possible to test without documenting at the test-case level. If you give an experienced user acceptance tester with a strong business background a list of high-level test conditions, they could probably do a good job of testing. But if you gave the same list to a new starter who didn't know the system at all, they would probably be lost, so they would benefit from having more detailed test cases.

Test cases can be documented as described in the IEEE 829 Standard for Test Documentation. Note that the contents described in the standard don't all have to be separate physical documents. But the standard's list of what needs to be kept track of is a good starting point, even if the test conditions and test cases for a given functionality or feature are all kept in one physical document.

One of the most important aspects of a test is that it assesses that the system does what it is supposed to do. Copeland says 'At its core, testing is the process of comparing "what is" with "what ought to be" '. [Copeland, 2003] If we simply put in some inputs and think 'that was fun, I guess the system is probably OK because it didn't crash', then are we actually testing it? We don't think so. You have observed that the system does what the system does - this is not a test. Boris Beizer refers to this as 'kiddie testing' [Beizer, 1990]. We may not know what the right answer is in detail every time, and we can still get some benefit from this approach at times, but it isn't really testing.

In order to know what the system *should* do, we need to have a source of information about the correct behavior of the system - this is called an **'oracle'** or a test oracle. This has nothing to do with databases or companies that make them. It comes from the ancient Greek Oracle at Delphi, who supposedly could predict the future with unerring accuracy. Actually her answers were so vague that people interpreted them in whatever way they wanted - perhaps a bit like requirements specifications!

Once a given input value has been chosen, the tester needs to determine what the expected result of entering that input would be and document it as part of the test case.

Expected results include information displayed on a screen in response to an input, but they also include changes to data and/or states, and any other consequences of the test (e.g. a letter to be printed overnight).

What if we don't decide on the expected results before we run a test? We can still look at what the system produces and would probably notice if something was wildly wrong. However, we would probably not notice small differences in calculations, or results that seemed to look OK (i.e. are plausible). So we would conclude that the test had passed, when in fact the software has not given the correct result. Small differences in one calculation can add up to something very major later on, for example if results are multiplied by a large factor.

Ideally expected results should be predicted before the test is run - then your assessment of whether or not the software did the right thing will be more objective.

For a few applications it may not be possible to predict or know exactly what an expected result should be; we can only do a 'reasonableness check'. In this case we have a 'partial oracle' - we know when something is very wrong, but would probably have to accept something that looked reasonable. An example is when a system has been written to calculate something where it may not be possible to manually produce expected results in a reasonable timescale because the calculations are so complex.

In addition to the expected results, the test case also specifies the environment and other things that must be in place before the test can be run (the preconditions) and any things that should apply after the test completes (the postconditions).

IEEE 829 STANDARD: ¡
TEST CASE SPECIFICATION TEMPLATE
[

Test case specification identifier	Output specifications
Test items	Environmental needs
Input specifications	Special procedural requirements
	Intercase dependencies

The test case should also say why it exists - i.e. the objective of the test it is part of or the test conditions that it is exercising (traceability). Test cases can now be prioritized so that the most important test cases are executed first, and low priority test cases are executed later, or even not executed at all. This may reflect the priorities already established for test conditions or the priority may be determined by other factors related to the specific test cases, such as a specific input value that has proved troublesome in the past, the risk associated with the test, or the most sensible sequence of running the tests. Chapter 5 gives more detail of risk-based testing.

Test cases need to be detailed so that we can accurately check the results and know that we have exactly the right response from the system. If tests are to be automated, the testing tool needs to know exactly what to compare the system output to.

4.1.5 Test implementation: specifying test procedures or scripts

The next step is to group the test cases in a sensible way for executing them and to specify the sequential steps that need to be done to run the test. For example, a set of simple tests that cover the breadth of the system may form a regression suite, or all of the tests that explore the working of a given functionality or feature in depth may be grouped to be run together.

Some test cases may need to be run in a particular sequence. For example, a test may create a new customer record, amend that newly created record and

then delete it. These tests need to be run in the correct order, or they won't test what they are meant to test.

The document that describes the steps to be taken in running a set of tests (and specifies the executable order of the tests) is called a **test procedure** in IEEE 829, and is often also referred to as a **test script.** It could be called a manual test script for tests that are intended to be run manually rather than using a test execution tool. Test script is also used to describe the instructions to a test execution tool. An automation script is written in a programming language that the tool can interpret. (This is an automated test procedure.) See Chapter 6 for more information on this and other types of testing tools.

The test procedures, or test scripts, are then formed into a test execution schedule that specifies which procedures are to be run first - a kind of super-script. The test schedule would say when a given script should be run and by whom. The schedule could vary depending on newly perceived risks affecting the priority of a script that addresses that risk, for example. The logical and technical dependencies between the scripts would also be taken into account when scheduling the scripts. For example, a regression script may always be the first to be run when a new release of the software arrives, as a smoke test or sanity check.

Returning to our example of the mobile phone company's marketing campaign, we may have some tests to set up customers of different types on the database. It may be sensible to run all of the setup for a group of tests first. So our first test procedure would entail setting up a number of customers, including Jim Green, on the database.

> Test procedure DB15: Set up customers for marketing campaign Y. Step 1: Open database with write privilege Step 2: Set up customer Bob Flounders
>
> male, 62, Hudsonville, contract
>
> Step 3: Set up customer Jim Green
>
> male, 17, Grand Rapids, pay-as-you-go, $8.64
>
> Step 4: ...

We may then have another test procedure to do with the marketing campaign:

> Test procedure MC03: Special offers for low-credit teenagers
> Step 1: Get details for Jim Green from database Step 2:
> Send text message offering double credit Step 3: Jim Green
> requests $20 credit, $40 credited

Writing the test procedure is another opportunity to prioritize the tests, to ensure that the best testing is done in the time available. A good rule of thumb is 'Find the scary stuff first'. However the definition of what is 'scary' depends on the business, system or project. For example, is it worse to raise Bob Founders' credit limit when he is not a good credit risk (he may not pay for the credit he asked for) or to refuse to raise his credit limit when he is a good credit risk (he may go elsewhere for his phone service and we lose the opportunity of lots of income from him).

IEEE 829 STANDARD:
TEST PROCEDURE SPECIFICATION TEMPLATE

Test procedure specification identifier

Purpose

Special requirements

Procedure steps

4.2 CATEGORIES OF TEST DESIGN TECHNIQUES

1 **Recall reasons that both specification-based (black-box) and structure-based (white-box) approaches to test case design are useful, and list the common techniques for each. (Kl)**
2 **Explain the characteristics and differences between specification-based testing, structure-based testing and experience-based testing. (K2)**

In this section we will look at the different types of test design technique, how they are used and how they differ. The three types or categories are distinguished by their primary source: a specification, the structure of the system or component, or a person's experience. All categories are useful and the three are complementary.

In this section, look for the definitions of the glossary terms: **black-box test design techniques, experience-based test design techniques, specification-based test design techniques, structure-based test design techniques** and **white-box test design techniques.**

4.2.1 Introduction

There are many different types of software testing technique, each with its own strengths and weaknesses. Each individual technique is good at finding particular types of defect and relatively poor at finding other types. For example, a technique that explores the upper and lower limits of a single input range is more likely to find boundary value defects than defects associated with combinations of inputs. Similarly, testing performed at different stages in the software development life cycle will find different types of defects; component testing is more likely to find coding logic defects than system design defects.

Each testing technique falls into one of a number of different categories. Broadly speaking there are two main categories, static and dynamic. Static techniques were discussed in Chapter 3. Dynamic techniques are subdivided into three more categories: specification-based (black-box, also known as behavioral

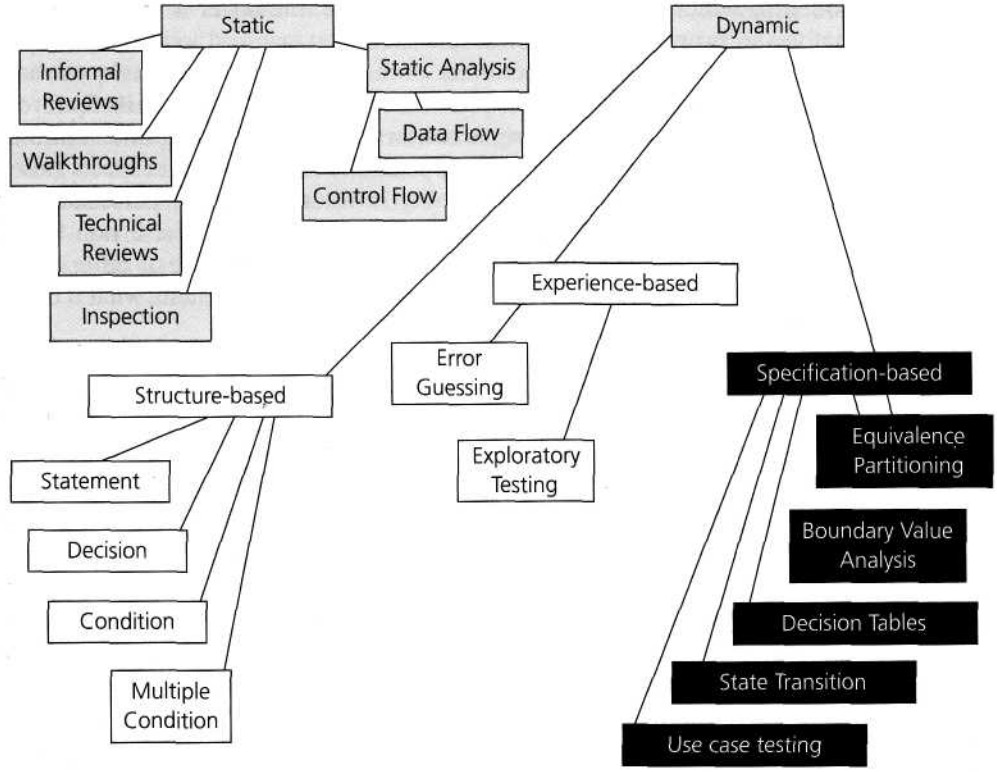


FIGURE 4.1 Testing techniques

techniques), structure-based (white-box or structural techniques) and experience-based. Specification-based techniques include both functional and non-functional techniques (i.e. quality characteristics). The techniques covered in the syllabus are summarized in Figure 4.1.

4.2.2 Static testing techniques

As we saw in Chapter 3, static testing techniques do not execute the code being examined and are generally used before any tests are executed on the software. They could be called non-execution techniques. Most static testing techniques can be used to 'test' any form of document including source code, design documents and models, functional specifications and requirement specifications. However, 'static analysis' is a tool-supported type of static testing that concentrates on testing formal languages and so is most often used to statically test source code.

4.2.3 Specification-based (black-box) testing techniques

The first of the dynamic testing techniques we will look at are the specification-based testing techniques. These are also known as **'black-box'** or input/output-driven testing techniques because they view the software as a black-box with inputs and outputs, but they have no knowledge of how the system or

component is structured inside the box. In essence, the tester is concentrating on what the software does, not how it does it.

Notice that the definition mentions both functional and non-functional testing. Functional testing is concerned with what the system does, its features or functions. Non-functional testing is concerned with examining how well the system does something, rather than what it does. Non-functional aspects (also known as quality characteristics or quality attributes) include performance, usability, portability, maintainability, etc. Techniques to test these non-functional aspects are less procedural and less formalized than those of other categories as the actual tests are more dependent on the type of system, what it does and the resources available for the tests.

Non-functional testing is part of the Syllabus and is also covered in Chapter 2. There are techniques for deriving non-functional tests [Gilb, 1988], [Testing Standards], but they are not covered at the Foundation level.

Categorizing tests into black and white-box is mentioned in a number of testing books, including [Beizer, 1990] and [Copeland, 2003].

4.2.4 Structure-based (white-box) testing techniques

Structure-based testing techniques (which are also dynamic rather than static) use the internal structure of the software to derive test cases. They are commonly called **'white-box'** or 'glass-box' techniques (implying you can see into the system) since they require knowledge of how the software is implemented, that is, how it works. For example, a structural technique may be concerned with exercising loops in the software. Different test cases may be derived to exercise the loop once, twice, and many times. This may be done regardless of the functionality of the software.

4.2.5 Experience-based testing techniques

In **experience-based techniques,** people's knowledge, skills and background are a prime contributor to the test conditions and test cases. The experience of both technical and business people is important, as they bring different perspectives to the test analysis and design process. Due to previous experience with similar systems, they may have insights into what could go wrong, which is very useful for testing.

4.2.6 Where to apply the different categories of techniques

Specification-based techniques are appropriate at all levels of testing (component testing through to acceptance testing) where a specification exists. When performing system or acceptance testing, the requirements specification or functional specification may form the basis of the tests. When performing component or integration testing, a design document or low-level specification forms the basis of the tests.

Structure-based techniques can also be used at all levels of testing. Developers use structure-based techniques in component testing and component integration testing, especially where there is good tool support for code coverage. Structure-based techniques are also used in system and acceptance

testing, but the structures are different. For example, the coverage of menu options or major business transactions could be the structural element in system or acceptance testing.

Experience-based techniques are used to complement specification-based and structure-based techniques, and are also used when there is no specification, or if the specification is inadequate or out of date. This may be the only type of technique used for low-risk systems, but this approach may be particularly useful under extreme time pressure - in fact this is one of the factors leading to exploratory testing.

4.3 SPECIFICATION-BASED OR BLACK-BOX TECHNIQUES

1 Write test cases from given software models using the following test design techniques. (K3)

a equivalence partitioning;
b boundary value analysis;
c decision tables;
d state transition testing.

2 Understand the main purpose of each of the four techniques, what level and type of testing could use the technique, and how coverage may be measured. (K2)

3 Understand the concept of use case testing and its benefits. (K2)

In this section we will look in detail at four specification-based or black-box techniques. These four techniques are K3 in the Syllabus - this means that you need to be able to use these techniques to design test cases. We will also cover briefly (not at K3 level) the specification-based technique of use case testing. In Section 4.4, we will look at the K3 structure-based techniques.

In this section, look for the definitions of the glossary terms: **boundary value analysis, decision table testing, equivalence partitioning, state transition testing** and **use case testing.**

The four specification-based techniques we will cover in detail are:

- equivalence partitioning;
- boundary value analysis;
- decision tables;
- state transition testing.

Note that we will discuss the first two together, because they are closely related.

4.3.1 Equivalence partitioning and boundary value analysis

Equivalence partitioning

Equivalence partitioning (EP) is a good all-round specification-based black-box technique. It can be applied at any level of testing and is often a good technique to use first. It is a common sense approach to testing, so much so that most testers practise it informally even though they may not realize it. However, while it is better to use the technique informally than not at all, it is much better to use the technique in a formal way to attain the full benefits that it can deliver. This technique will be found in most testing books, including [Myers, 1979] and [Copeland, 2003].

The idea behind the technique is to divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same (i.e. the system should handle them equivalently), hence 'equivalence partitioning'. **Equivalence partitions** are also known as equivalence classes - the two terms mean exactly the same thing.

The equivalence-partitioning technique then requires that we need test only one condition from each partition. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software. If one condition in a partition works, we assume all of the conditions in that partition will work, and so there is little point in testing any of these others. Conversely, if one of the conditions in a partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition. Of course these are simplifying assumptions that may not always be right but if we write them down, at least it gives other people the chance to challenge the assumptions we have made and hopefully help to identify better partitions. If you have time, you may want to try more than one value from a partition, especially if you want to confirm a selection of typical user inputs.

For example, a savings account in a bank earns a different rate of interest depending on the balance in the account. In order to test the software that calculates the interest due, we can identify the ranges of balance values that earn the different rates of interest. For example, if a balance in the range $0 up to $100 has a 3% interest rate, a balance over $100 and up to $1000 has a 5% interest rate, and balances of $1000 and over have a 7% interest rate, we would initially identify three valid equivalence partitions and one invalid partition as shown below.

Invalid partition	Valid (for 3% interest)		Valid (for 5%)		Valid (for 7%)
-$0.01	$0.00	$100.00	$100.01	$999.99	$1000.00

Notice that we have identified four partitions here, even though the specification only mentions three. This illustrates a very important task of the tester - not only do we test what is in our specification, but we also think about things that haven't been specified. In this case we have thought of the situation where the balance is less than zero. We haven't (yet) identified an invalid partition on the right, but this would also be a good thing to consider. In order to identify where the 7% partition ends, we would need to know what the maximum balance is for this account (which may not be easy to find out). In our example we have left this open for the time being. Note that non-numeric input is also an invalid partition (e.g. the letter 'a') but we discuss only the numeric partitions for now.

We have made an assumption here about what the smallest difference is between two values. We have assumed two decimal places, i.e. $100.00, but we could have assumed zero decimal places (i.e. $100) or more than two decimal places (e.g. $100.0000) In any case it is a good idea to state your assumptions - then other people can see them and let you know if they are correct or not.

When designing the test cases for this software we would ensure that the three valid equivalence partitions are each covered once, and we would also test the invalid partition at least once. So for example, we might choose to calculate the interest on balances of-$10.00, $50.00, $260.00 and $1348.00. If we hadn't specifically identified these partitions, it is possible that at least one of them could have been missed at the expense of testing another one several times over. Note that we could also apply equivalence partitioning to outputs as well. In this case we have three interest rates: 3%, 5% and 7%, plus the error message for the invalid partition (or partitions). In this example, the output partitions line up exactly with the input partitions.

How would someone test this without thinking about the partitions? A naive tester (let's call him Robbie) might have thought that a good set of tests would be to test every $50. That would give the following tests: $50.00, $100.00, $150.00, $200.00, $250.00, ... say up to $800.00 (then Robbie would have got tired of it and thought that enough tests had been carried out). But look at what Robbie has tested: only two out of four partitions! So if the system does not correctly handle a negative balance or a balance of $1000 or more, he would not have found these defects - so the naive approach is less effective than equivalence partitioning. At the same time, Robbie has four times more tests (16 tests versus our four tests using equivalence partitions), so he is also much less efficient! This is why we say that using techniques such as this makes testing both more effective and more efficient.

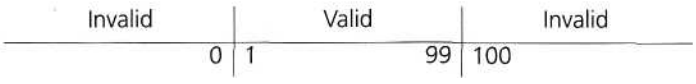
Note that when we say a partition is 'invalid', it doesn't mean that it represents a value that cannot be entered by a user or a value that the user isn't supposed to enter. It just means that it is not one of the expected inputs for this particular field. The software should correctly handle values from the invalid partition, by replying with an error message such as 'Balance must be at least $0.00'.

Note also that the invalid partition may be invalid only in the context of crediting interest payments. An account that is overdrawn will require some different action.

Boundary value analysis

Boundary value analysis (BVA) is based on testing at the boundaries between partitions. If you have ever done 'range checking', you were probably using the boundary value analysis technique, even if you weren't aware of it. Note that we have both valid boundaries (in the valid partitions) and invalid boundaries (in the invalid partitions).

As an example, consider a printer that has an input option of the number of



To apply boundary value analysis, we will take the minimum and maximum (boundary) values from the valid partition (1 and 99 in this case) together with copies to be made, from 1 to 99.

the first or last value respectively in each of the invalid partitions adjacent to the valid partition (0 and 100 in this case). In this example we would have three equivalence partitioning tests (one from each of the three partitions) and four boundary value tests.

Consider the bank system described in the section about equivalence partitioning.

Invalid partition	Valid (for 3% interest)		Valid (for 5%)		Valid (for 7%)
–$0.01	$0.00	$100.00	$100.01	$999.99	$1000.00

Because the boundary values are defined as those values on the edge of a partition, we have identified the following boundary values: -$0.01 (an invalid boundary value because it is at the edge of an invalid partition), $0.00, $100.00, $100.01, $999.99 and $1000.00, all valid boundary values.

So by applying boundary value analysis we will have six tests for boundary values. Compare what our naive tester Robbie had done: he did actually hit one of the boundary values ($100) though it was more by accident than design. So in addition to testing only half of the partitions, Robbie has only tested one-sixth of the boundaries (so he will be less effective at finding any boundary defects). If we consider all of our tests for both equivalence partitioning and boundary value analysis, the techniques give us a total of nine tests, compared to the 16 that Robbie had, so we are still considerably more efficient as well as being over three times more effective (testing four partitions and six boundaries, so 10 conditions in total compared to three).

Note that in the bank interest example, we have valid partitions next to other valid partitions. If we were to consider an invalid boundary for the 3% interest rate, we have -$0.01, but what about the value just above $100.00? The value of $100.01 is not an *invalid* boundary; it is actually a *valid* boundary because it falls into a valid partition. So the partition for 5%, for example, has no invalid boundary values associated with partitions next to it.

A good way to represent the valid and invalid partitions and boundaries is in a table such as Table 4.1:

TABLE 4.1 Equivalence partitions and boundaries

Test conditions	Valid partitions	Invalid partitions	Valid boundaries	Invalid boundaries
Balance in aU$_o$Unt	$0.00 $100.00	< $0.00	$0.00	-$0.01
	$100.01-$999.99	>$Max	$100.00	$Max+0.01
	$1000.00- $Max	non-integer (if balance is an input field)	$100.01	
			$999.99	
			$1000.00	
			$Max	
Interest rates	3%	Any other value	Not applicable	Not applicable
	5%	Non-integer		
	7%	No interest calculated		

By showing the values in the table, we can see that no maximum has been specified for the 7% interest rate. We would now want to know what the maximum value is for an account balance, so that we can test that boundary. This is called an 'open boundary', because one of the sides of the partition is left open, i.e. not defined. But that doesn't mean we can ignore it - we should still try to test it, but how?

Open boundaries are more difficult to test, but there are ways to approach them. Actually the best solution to the problem is to find out what the boundary should be specified as! One approach is to go back to the specification to see if a maximum has been stated somewhere else for a balance amount. If so, then we know what our boundary value is. Another approach might be to investigate other related areas of the system. For example, the field that holds the account balance figure may be only six figures plus two decimal figures. This would give a maximum account balance of $999 999.99 so we could use that as our maximum boundary value. If we really cannot find anything about what this boundary should be, then we probably need to use an intuitive or experience-based approach to probe various large values trying to make it fail.

We could also try to find out about the lower open boundary - what is the lowest negative balance? Although we have omitted this from our example, setting it out in the table shows that we have omitted it, so helps us be more thorough if we wanted to be.

Representing the partitions and boundaries in a table such as this also makes it easier to see whether or not you have tested each one (if that is your objective).

Extending equivalence partitioning and boundary value analysis So far, by using EP and BVA we have identified conditions that could be tested, i.e. partitions and boundary values. The techniques are used to identify test conditions, which could be at a fairly high level (e.g. 'low-interest account') or at a detailed level (e.g. 'value of $100.00'). We have been looking at applying these techniques to ranges of numbers. However, we can also apply the techniques to other things.

For example, if you are booking a flight, you have a choice of Economy/Coach, Premium Economy, Business or First Class tickets. Each of these is an equivalence partition in its own right and should be tested, but it doesn't make sense to talk about boundaries for this type of partition, which is a collection of valid things. The invalid partition would be an attempt to type in any other type of flight class (e.g. 'Staff'). If this field is implemented using a drop-down list, then it should not be possible to type anything else in, but it is still a good test to try at least once in some drop-down field. When you are analyzing the test basis (e.g. a requirements specification), equivalence partitioning can help to identify where a drop-down list would be appropriate.

When trying to identify a defect, you might try several values in a partition. If this results in different behavior where you expected it to be the same, then there may be two (or more) partitions where you initially thought there was only one.

We can apply equivalence partitioning and boundary value analysis to all levels of testing. The examples here were at a fairly detailed level probably most appropriate in component testing or in the detailed testing of a single screen.

At a system level, for example, we may have three basic configurations which our users can choose from when setting up their systems, with a number of options for each configuration. The basic configurations could be system administrator, manager and customer liaison. These represent three equivalence partitions that could be tested. We could have serious problems if we forget to test the configuration for the system administrator, for example.

We can also apply equivalence partitioning and boundary value analysis more than once to the same specification item. For example, if an internal telephone system for a company with 200 telephones has 3-digit extension numbers from 100 to 699, we can identify the following partitions and boundaries:

- digits (characters 0 to 9) with the invalid partition containing non-digits
- number of digits, 3 (so invalid boundary values of 2 digits and 4 digits)
- range of extension numbers, 100 to 699 (so invalid boundary values of 099 and 700)
- extensions that are in use and those that are not (two valid partitions, no boundaries)
- the lowest and highest extension numbers that are in use could also be used as boundary values

One test case could test more than one of these partitions/boundaries. For example, Extension 409 which is in use would test four valid partitions: digits, the number of digits, the valid range, and the 'in use' partition. It also tests the boundary values for digits, 0 and 9.

How many test cases would we need to test all of these partitions and boundaries, both valid and invalid? We would need a non-digit, a 2-digit and 4-digit number, the values of 99, 100, 699 and 700, one extension that is not in use, and possibly the lowest and highest extensions in use. This is ten or eleven test cases - the exact number would depend on what we could combine in one test case.

Compare this with the one-digit number example in Section 1.1.6. Here we found that we needed 68 tests just to test a one-digit field, if we were to test it thoroughly. Using equivalence partitioning and boundary value analysis helps us to identify tests that are most likely to find defects, and to use fewer test cases to find them. This is because the contents of a partition are representative of all of the possible values. Rather than test all ten individual digits, we test one in the middle (e.g. 4) and the two edges (0 and 9). Instead of testing every possible non-digit character, one can represent all of them. So we have four tests (instead of 68) for a one-digit field.

As we mentioned earlier, we can also apply these techniques to output partitions. Consider the following extension to our bank interest rate example. Suppose that a customer with more than one account can have an extra 1% interest on this account if they have at least $1000 in it. Now we have two possible output values (7% interest and 8% interest) for the same account balance, so we have identified another test condition (8% interest rate). (We may also have identified that same output condition by looking at customers with more than one account, which is a partition of types of customer.)

Equivalence partitioning can be applied to different types of input as well. Our examples have concentrated on inputs that would be typed in by a (human) user when using the system. However, systems receive input data from other

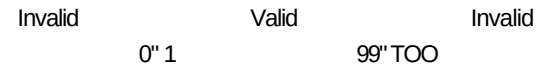
sources as well, such as from other systems via some interface - this is also a good place to look for partitions (and boundaries). For example, the value of an interface parameter may fall into valid and invalid equivalence partitions. This type of defect is often difficult to find in testing once the interfaces have been joined together, so is particularly useful to apply in integration testing (either component integration or system integration).

Boundary value analysis can be applied to the whole of a string of characters (e.g. a name or address). The number of characters in the string is a partition, e.g. between 1 and 30 characters is the valid partition with valid boundaries of 1 and 30. The invalid boundaries would be 0 characters (null, just hit the Return key) and 31 characters. Both of these should produce an error message.

Partitions can also be identified when setting up test data. If there are different types of record, your testing will be more representative if you include a data record of each type. The size of a record is also a partition with boundaries, so we could include maximum and minimum size records in the test database.

If you have some inside knowledge about how the data is physically organized, you may be able to identify some hidden boundaries. For example, if an overflow storage block is used when more than 255 characters are entered into a field, the boundary value tests would include 255 and 256 characters in that field. This may be verging on white-box testing, since we have some knowledge of how the data is structured, but it doesn't matter how we classify things as long as our testing is effective at finding defects. Don't get hung up on a fine distinction - just do whatever testing makes sense, based on what you know. An old Chinese proverb says, 'It doesn't matter whether the cat is white or black; all that matters is that the cat catches mice'.

With boundary value analysis, we think of the boundary as a dividing line between two things. Hence we have a value on each side of the boundary (but the boundary itself is not a value).



Looking at the values for our printer example, 0 is in an invalid partition, 1 and 99 are in the valid partition and 100 is in the other invalid partition. So the boundary is between the values of 0 and 1, and between the values of 99 and 100. There is a school of thought that regards an actual value as a boundary value. By tradition, these are the values in the valid partition (i.e. the values specified). This approach then requires three values for every boundary two, so you would have 0,1 and 2 for the left boundary, and 98, 99 and 100 for the right boundary in this example. The boundary values are said to be 'on and either side of the boundary' and the value that is 'on' the boundary is generally taken to be in the valid partition.

Note that Beizer talks about domain testing, a generalization of equivalence partitioning, with three-value boundaries. He makes a distinction between open and closed boundaries, where a closed boundary is one where the point is included in the domain. So the convention is for the valid partition to have closed boundaries. You may be pleased to know that you don't have to know this for the exam! British Standard 7925-2 Standard for Software Component Testing also defines a three-value approach to boundary value analysis.

So which approach is best? If you use the two-value approach together with equivalence partitioning, you are equally effective and slightly more efficient than the three-value approach. (We won't go into the details here but this can be demonstrated.) In this book we will use the two-value approach. In the exam, you may have a question based on either the two-value or the three-value approach, but it should be clear what the correct choice is in either case.

Designing test cases

Having identified the conditions that you wish to test, in this case by using equivalence partitioning and boundary value analysis, the next step is to design the test cases. The more test conditions that can be covered in a single test case, the fewer test cases will be needed in order to cover all the conditions. This is usually the best approach to take for positive tests and for tests that you are reasonably confident will pass. However if a test fails, then we need to find out why it failed - which test condition was handled incorrectly? We need to get a good balance between covering too many and too few test conditions in our tests.

Let's look at how one test case can cover one or more test conditions. Using the bank balance example, our first test could be of a new customer with a balance of $500. This would cover a balance in the partition from $100.01 to $999.99 and an output partition of a 5% interest rate. We would also be covering other partitions that we haven't discussed yet. for example a valid customer, a new customer, a customer with only one account, etc. All of the partitions covered in this test are valid partitions.

When we come to test invalid partitions, the safest option is probably to try to cover only one invalid test condition per test case. This is because programs may stop processing input as soon as they encounter the first problem. So if you have an invalid customer name, invalid address, and invalid balance, you may get an error message saying 'invalid input' and you don''t know whether the test has detected only one invalid input or all of them. (This is also why specific error messages are much better than general ones!)

However, if it is known that the software under test is required to process all input regardless of its validity, then it is sensible to continue as before and design test cases that cover as many invalid conditions in one go as possible. For example, if every invalid field in a form has some red text above or below the field saying that this field is invalid and why, then you know that each field has been checked, so you have tested all of the error processing in one test case. In either case, there should be separate test cases covering valid and invalid conditions.

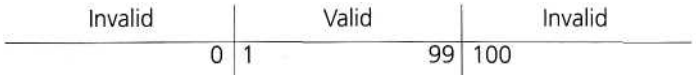
To cover the boundary test cases, it may be possible to combine all of the minimum valid boundaries for a group of fields into one test case and also the maximum boundary values. The invalid boundaries could be tested together if the validation is done on every field; otherwise they should be tested separately, as with the invalid partitions.

Why do both equivalence partitioning and boundary value analysis?

Technically, because every boundary is in some partition, if you did only boundary value analysis you would also have tested every equivalence partition. However, this approach may cause problems if that value fails - was it only the boundary value that failed or did the whole partition fail? Also by testing only

boundaries we would probably not give the users much confidence as we are using extreme values rather than normal values. The boundaries may be more difficult (and therefore more costly) to set up as well.

For example, in the printer copies example described earlier we identified the following boundary values:



Suppose we test only the valid boundary values 1 and 99 and nothing in between. If both tests pass, this seems to indicate that all the values in between should also work. However, suppose that one page prints correctly, but 99 pages do not. Now we don't know whether any set of more than one page works, so the first thing we would do would be to test for say 10 pages, i.e. a value from the equivalence partition.

We recommend that you test the partitions separately from boundaries - this means choosing partition values that are NOT boundary values.

However, if you use the three-value boundary value approach, then you would have valid boundary values of 1, 2, 98 and 99, so having a separate equivalence value in addition to the extra two boundary values would not give much additional benefit. But notice that one equivalence value, e.g. 10, replaces both of the extra two boundary values (2 and 98). This is why equivalence partitioning with two-value boundary value analysis is more efficient than three-value boundary value analysis.

Which partitions and boundaries you decide to exercise (you don't need to test them all), and which ones you decide to test first, depends on your test objectives. If your goal is the most thorough approach, then follow the procedure of testing valid partitions first, then invalid partitions, then valid boundaries and finally invalid boundaries. However if you are under time pressure and cannot test everything (and who isn't?), then your test objectives will help you decide what to test. If you are after user confidence of typical transactions with a minimum number of tests, you may do valid partitions only. If you want to find as many defects as possible as quickly as possible, you may start with boundary values, both valid and invalid. If you want confidence that the system will handle bad inputs correctly, you may do mainly invalid partitions and boundaries. Your previous experience of types of defects found can help you find similar defects; for example if there are typically a number of boundary defects, then you would start by testing boundaries.

Equivalence partitioning and boundary value analysis are described in most testing books, including [Myers, 1979] and [Copeland, 2003]. Examples of types of equivalence classes to look out for are given in [Kaner *et al.*, 1993] Equivalence partitioning and boundary value analysis are described in BS7925-2, including designing tests and measuring coverage.

4.3.2 Decision table testing

Why use decision tables?.

The techniques of equivalence partitioning and boundary value analysis are often applied to specific situations or inputs. However, if different combinations

of inputs result in different actions being taken, this can be more difficult to show using equivalence partitioning and boundary value analysis, which tend to be more focused on the user interface. The other two specification-based techniques, decision tables and state transition testing are more focused on business logic or business rules.

A **decision table** is a good way to deal with combinations of things (e.g. inputs). This technique is sometimes also referred to as a 'cause-effect' table. The reason for this is that there is an associated logic diagramming technique called 'cause-effect graphing' which was sometimes used to help derive the decision table (Myers describes this as a combinatorial logic network [Myers, 1979]). However, most people find it more useful just to use the table described in [Copeland, 2003].

If you begin using decision tables to explore what the business rules are that should be tested, you may find that the analysts and developers find the tables very helpful and want to begin using them too. Do encourage this, as it will make your job easier in the future. Decision tables provide a systematic way of stating complex business rules, which is useful for developers as well as for testers. Decision tables can be used in test design whether or not they are used in specifications, as they help testers explore the effects of combinations of different inputs and other software states that must correctly implement business rules. Helping the developers do a better job can also lead to better relationships with them.

Testing combinations can be a challenge, as the number of combinations can often be huge. Testing all combinations may be impractical if not impossible. We have to be satisfied with testing just a small subset of combinations but making the choice of which combinations to test and which to leave out is not trivial. If you do not have a systematic way of selecting combinations, an arbitrary subset will be used and this may well result in an ineffective test effort.

Decision tables aid the systematic selection of effective test cases and can have the beneficial side-effect of finding problems and ambiguities in the specification. It is a technique that works well in conjunction with equivalence partitioning. The combination of conditions explored may be combinations of equivalence partitions.

In addition to decision tables, there are other techniques that deal with testing combinations of things: pairwise testing and orthogonal arrays. These are described in [Copeland, 2003]. Another source of techniques is [Pol *et al.*, 2001]. Decision tables and cause-effect graphing are described in [BS7925-2], including designing tests and measuring coverage.

Using decision tables for test design

The first task is to identify a suitable function or subsystem that has a behavior which reacts according to a combination of inputs or events. The behavior of interest must not be too extensive (i.e. should not contain too many inputs) otherwise the number of combinations will become cumbersome and difficult to manage. It is better to deal with large numbers of conditions by dividing them into subsets and dealing with the subsets one at a time.

Once you have identified the aspects that need to be combined, then you put them into a table listing all the combinations of True and False for each of the aspects. Take an example of a loan application, where you can enter the amount

of the monthly repayment or the number of years you want to take to pay it back (the term of the loan). If you enter both, the system will make a compromise between the two if they conflict. The two conditions are the loan amount and the term, so we put them in a table (see Table 4.2).

TABLE 4.2 Empty decision table

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been entered				
Term of loan has been entered				

Next we will identify all of the combinations of True and False (see Table 4.3). With two conditions, each of which can be True or False, we will have four combinations (two to the power of the number of things to be combined). Note that if we have three things to combine, we will have eight combinations, with four things, there are 16, etc. This is why it is good to tackle small sets of combinations at a time. In order to keep track of which combinations we have, we will alternate True and False on the bottom row, put two Trues and then two Falses on the row above the bottom row, etc., so the top row will have all Trues and then all Falses (and this principle applies to all such tables).

TABLE 4.3 Decision table with input combinations

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has teen *entered*	T	T	F	F
Term of loan has been entered	T	F	T	F

The next step (at least for this example) is to identify the correct outcome for each combination (see Table 4.4). In this example, we can enter one or both of the two fields. Each combination is sometimes referred to as a rule.

TABLE 4.4 Decision table with combinations and outcomes

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been *entered*	T	T	F	F
Term of loan has been	T	F	T	F
entered				
Actions/Outcomes				
Process loan amount	Y	Y		
Process term	Y		Y	

At this point, we may realize that we hadn't thought about what happens if the customer doesn't enter anything in either of the two fields. The table has highlighted a combination that was not mentioned in the specification for this example. We could assume that this combination should result in an error message, so we need to add another action (see Table 4.5). This highlights the strength of this technique to discover omissions and ambiguities in specifications. It is not unusual for some combinations to be omitted from specifications; therefore this is also a valuable technique to use when reviewing the test basis.

TABLE 4.5 Decision table with additional outcomes

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been *entered*	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount	Y	Y		
Process term	Y		Y	
Error message				Y

Suppose we change our example slightly, so that the customer is not allowed to enter both repayment and term. Now our table will change, because there should also be an error message if both are entered, so it will look like Table 4.6.

TABLE 4.6 Decision table with changed outcomes

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount		Y		
Process term			Y	
Error message	Y			Y

You might notice now that there is only one 'Yes' in each column, i.e. our actions are mutually exclusive - only one action occurs for each combination of conditions. We could represent this in a different way by listing the actions in the cell of one row, as shown in Table 4.7. Note that if more than one action results from any of the combinations, then it would be better to show them as separate rows rather than combining them into one row.

TABLE 4.7 Decision table with outcomes in one row

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Result	Error message	Process loan amount	Process term	Error message

The final step of this technique is to write test cases to exercise each of the four rules in our table.

In this example we started by identifying the input conditions and then identifying the outcomes. However in practice it might work the other way around - we can see that there are a number of different outcomes, and have to work back to understand what combination of input conditions actually drive those outcomes. The technique works just as well doing it in this way, and may well be an iterative approach as you discover more about the rules that drive the system.

Credit card worked example

Let's look at another example. If you are a new customer opening a credit card account, you will get a 15% discount on all your purchases today. If you are an existing customer and you hold a loyalty card, you get a 10% discount. If you have a coupon, you can get 20% off today (but it can't be used with the 'new customer' discount). Discount amounts are added, if applicable. This is shown in Table 4.8.

TABLE 4.8 Decision table for credit card example

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**	**Rule 5**	**Rule 6**	**Rule 7**	**Rule 8**
New customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Actions								
Discount (%)	X	X	20	15	30	10	20	0

In Table 4.8, the conditions and actions are listed in the left hand column. All the other columns in the decision table each represent a separate rule, one for each combination of conditions. We may choose to test each rule/combination and if there are only a few this will usually be the case. However, if the number of rules/combinations is large we are more likely to sample them by selecting a rich subset for testing.

Note that we have put X for the discount for two of the columns (Rules 1 and 2) - this means that this combination should not occur. You cannot be both a new customer and already hold a loyalty card! There should be an error message stating this, but even if we don't know what that message should be, it will still make a good test.

We have made an assumption in Rule 3. Since the coupon has a greater discount than the new customer discount, we assume that the customer will choose 20% rather than 15%. We cannot add them, since the coupon cannot be used with the 'new customer' discount. The 20% action is an assumption on our part, and we should check that this assumption (and any other assumptions that we make) is correct, by asking the person who wrote the specification or the users.

For Rule 5, however, we can add the discounts, since both the coupon and the loyalty card discount should apply (at least that's our assumption).

Rules 4, 6 and 7 have only one type of discount and Rule 8 has no discount. so 0%.

If we are applying this technique thoroughly, we would have one test for each column or rule of our decision table. The advantage of doing this is that we may test a combination of things that otherwise we might not have tested and that could find a defect.

However, if we have a lot of combinations, it may not be possible or sensible to test every combination. If we are time-constrained, we may not have time to test all combinations. Don't just assume that all combinations need to be tested; it is better to prioritize and test the most important combinations. Having the full table enables us to see which combinations we decided to test and which not to test this time.

There may also be many different actions as a result of the combinations of conditions. In the example above we just had one: the discount to be applied. The decision table shows which actions apply to each combination of conditions.

In the example above all the conditions are binary, i.e. they have only two possible values: True or False (or, if you prefer Yes or No). Often it is the case that conditions are more complex, having potentially many possible values. Where this is the case the number of combinations is likely to be very large, so the combinations may only be sampled rather than exercising all of them.

4.3.3 State transition testing

State transition testing is used where some aspect of the system can be described in what is called a 'finite state machine'. This simply means that the system can be in a (finite) number of different states, and the transitions from one state to another are determined by the rules of the 'machine'. This is the model on which the system and the tests are based. Any system where you get a different output for the same input, depending on what has happened before, is a finite state system. A finite state system is often shown as a **state diagram** (see Figure 4.2).

For example, if you request to withdraw $100 from a bank ATM, you may be given cash. Later you may make exactly the same request but be refused the money (because your balance is insufficient). This later refusal is because the state of your bank account has changed from having sufficient funds to cover

the withdrawal to having insufficient funds. The transaction that caused your account to change its state was probably the earlier withdrawal. A state diagram can represent a model from the point of view of the system, the account or the customer.

Another example is a word processor. If a document is open, you are able to close it. If no document is open, then 'Close' is not available. After you choose 'Close' once, you cannot choose it again for the same document unless you open that document. A document thus has two states: open and closed.

A state transition model has four basic parts:

- the states that the software may occupy (open/closed or funded/insufficient funds);
- the transitions from one state to another (not all transitions are allowed);
- the events that cause a transition (closing a file or withdrawing money);
- the actions that result from a transition (an error message or being given your cash).

Note that in any given state, one event can cause only one action, but that the same event - from a different state - may cause a different action and a different end state.

We will look first at test cases that execute valid state transitions.

Figure 4.2 shows an example of entering a Personal Identity Number (PIN) to a bank account. The states are shown as circles, the transitions as lines with arrows and the events as the text near the transitions. (We have not shown the actions explicitly on this diagram, but they would be a message to the customer saying things such as 'Please enter your PIN'.)

The state diagram shows seven states but only four possible events (Card inserted, Enter PIN, PIN OK and PIN not OK). We have not specified all of the possible transitions here - there would also be a time-out from 'wait for PIN' and from the three tries which would go back to the start state after the time had elapsed and would probably eject the card. There would also be a transition from the 'eat card' state back to the start state. We have not specified all the possible events either - there would be a 'cancel' option from 'wait for PIN'

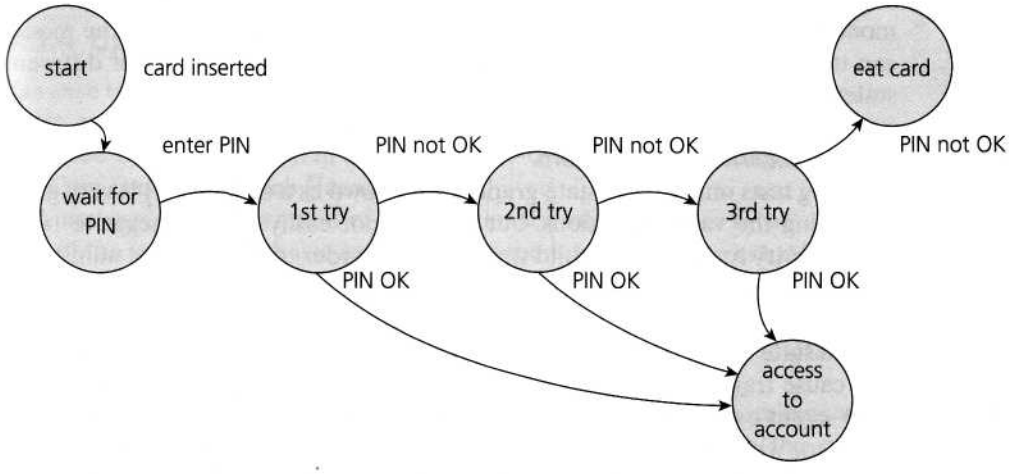


FIGURE 4.2 State diagram for PIN entry

and from the three tries, which would also go back to the start state and eject the card. The 'access account' state would be the beginning of another state diagram showing the valid transactions that could now be performed on the account.

However this state diagram, even though it is incomplete, still gives us information on which to design some useful tests and to explain the state transition technique.

In deriving test cases, we may start with a typical scenario. A sensible first test case here would be the normal situation, where the correct PIN is entered the first time. To be more thorough, we may want to make sure that we cover every state (i.e. at least one test goes through each state) or we may want to cover every transition. A second test (to visit every state) would be to enter an incorrect PIN each time, so that the system eats the card. We still haven't tested every transition yet. In order to do that, we would want a test where the PIN was incorrect the first time but OK the second time, and another test where the PIN was correct on the third try. These tests are probably less important than the first two.

Note that a transition does not need to change to a different state (although all of the transitions shown above do go to a different state). So there could be a transition from 'access account' which just goes back to 'access account' for an action such as 'request balance'.

Test conditions can be derived from the state graph in various ways. Each state can be noted as a test condition, as can each transition. In the Syllabus, we need to be able to identify the coverage of a set of tests in terms of transitions.

Going beyond the level expected in the Syllabus, we can also consider transition pairs and triples and so on. Coverage of all individual transitions is also known as 0-switch coverage, coverage of transition pairs is l-switch coverage, coverage of transition triples is 2-switch coverage, etc. Deriving test cases from the state transition model is a black-box approach. Measuring how much you have tested (covered) is getting close to a white-box perspective. However, state transition testing is regarded as a black-box technique.

One of the advantages of the state transition technique is that the model can be as detailed or as abstract as you need it to be. Where a part of the system is more important (that is, requires more testing) a greater depth of detail can be modeled. Where the system is less important (requires less testing), the model can use a single state to signify what would otherwise be a series of different states.

Testing for invalid transitions

Deriving tests only from a state graph (also known as a state chart) is very good for seeing the valid transitions, but we may not easily see the negative tests, where we try to generate invalid transitions. In order to see the total number of combinations of states and transitions, both valid and invalid, a **state table** is useful.

The state table lists all the states down one side of the table and all the events that cause transitions along the top (or vice versa). Each cell then represents a state-event pair. The content of each cell indicates which state the system will move to, when the corresponding event occurs while in the associated state. This will include possible erroneous events - events that are not expected to happen in certain states. These are negative test conditions.

TABLE 4.9 State table for the PIN example

	Insert card	Valid PIN	Invalid PIN
S1) Start state	S2	–	–
S2) Wait for PIN	–	S6	S3
S3) 1st try invalid	–	S6	S4
S4) 2nd try invalid	–	S6	S5
S5) 3rd try invalid	–	–	S7
S6) Access account	–	?	?
S7) Eat card	S1 (for new card)	–	–

Table 4.9 lists the states in the first column and the possible inputs across the top row. So, for example, if the system is in State 1, inserting a card will take it to State 2. If we are in State 2, and a valid PIN is entered, we go to State 6 to access the account. In State 2 if we enter an invalid PIN, we go to State 3. We have put a dash in the cells that should be impossible, i.e. they represent invalid transitions from that state.

We have put a question mark for two cells, where we enter either a valid or invalid PIN when we are accessing the account. Perhaps the system will take our PIN number as the amount of cash to withdraw? It might be a good test! Most of the other invalid cells would be physically impossible in this example. Invalid (negative) tests will attempt to generate invalid transitions, transitions that shouldn't be possible (but often make good tests when it turns out they are possible).

A more extensive description of state machines is found in [Marick, 1994]. State transition testing is also described in [Craig, 2002], [Copeland, 2003], [Beizer, 1990] and [Broekman, 2003]. State transition testing is described in BS7925-2, including designing tests and coverage measures.

4.3.4 Use case testing

Use case testing is a technique that helps us identify test cases that exercise the whole system on a transaction by transaction basis from start to finish. They are described by Ivar Jacobson in his book *Object-Oriented Software Engineering: A Use Case Driven Approach* [Jacobson, 1992].

A use case is a description of a particular use of the system by an actor (a user of the system). Each use case describes the interactions the actor has with the system in order to achieve a specific task (or, at least, produce something of value to the user). Actors are generally people but they may also be other systems. Use cases are a sequence of steps that describe the interactions between the actor and the system.

Use cases are defined in terms of the actor, not the system, describing what the actor does and what the actor sees rather than what inputs the system expects and what the system'outputs. They often use the language and terms of the business rather than technical terms, especially when the actor is a business

user. They serve as the foundation for developing test cases mostly at the system and acceptance testing levels.

Use cases can uncover integration defects, that is, defects caused by the incorrect interaction between different components. Used in this way, the actor may be something that the system interfaces to such as a communication link or sub-system.

Use cases describe the process flows through a system based on its most likely use. This makes the test cases derived from use cases particularly good for finding defects in the real-world use of the system (i.e. the defects that the users are most likely to come across when first using the system). Each use case usually has a mainstream (or most likely) scenario and sometimes additional alternative branches (covering, for example, special cases or exceptional conditions). Each use case must specify any preconditions that need to be met for the use case to work. Use cases must also specify postconditions that are observable results and a description of the final state of the system after the use case has been executed successfully.

The PIN example that we used for state transition testing could also be defined in terms of use cases, as shown in Figure 4.3. We show a success scenario and the extensions (which represent the ways in which the scenario could fail to be a success).

For use case testing, we would have a test of the success scenario and one tesi for each extension. In this example, we may give extension 4b a higher priority than 4a from a security point of view.

System requirements can also be specified as a set of use cases. This approach can make it easier to involve the users in the requirements gathering and definition process.

	Step	Description
Main Success Scenario **A: Actor** **S: System**	1	A: Inserts card
	2	S: Validates card and asks for PIN
	3	A: Enters PIN
	4	S: Validates PIN
	5	S: Allows access to account
Extensions	2a	Card not valid S: Display message and reject card
	4a	PIN not valid S: Display message and ask for re-try (twice)
	4b	PIN invalid 3 times S: Eat card and exit

FIGURE 4.3 Partial use case for PIN entry

4.4 STRUCTURE-BASED OR WHITE-BOX TECHNIQUES

1 **Describe the concept and importance of code coverage. (K2)**
2 **Explain the concepts of statement and decision coverage and under**
stand that these concepts can also be used at other test levels than com
ponent testing (e.g. on business procedures at system level). (K2)
3 **Write test cases from given control flows using the following test design**
techniques: (K3)
a statement
coverage; b
decision coverage.
4 **Assess statement and decision coverage for completeness. (K3)**

In this section we will look in detail at the concept of coverage and how it can be used to measure some aspects of the thoroughness of testing. In order to see how coverage actually works, we will use some code-level examples (although coverage also applies to other levels such as business procedures). In particular, we will show how to measure coverage of statements and decisions, and how to write test cases to extend coverage if it is not 100%. The same principles apply to coverage of system-level coverage items, for example menu items.

In this section, look for the definitions of the glossary terms: **code coverage, decision coverage, statement coverage, structural testing, structure-based testing** and **white-box testing**

4.4.1 Using structure-based techniques to measure coverage and design tests

Structure-based techniques serve two purposes: test coverage measurement and structural test case design. They are often used first to assess the amount of testing performed by tests derived from specification-based techniques, i.e. to assess coverage. They are then used to design additional tests with the aim of increasing the test coverage.

Structure-based test design techniques are a good way of generating additional test cases that are different from existing tests. They can help ensure more breadth of testing, in the sense that test cases that achieve 100% **coverage** in any measure will be exercising all parts of the software from the point of view of the items being covered.

What is test coverage?

Test coverage measures in some specific way the amount of testing performed by a set of tests (derived in some other way, e.g. using specification-based techniques). Wherever we can

count things and can tell whether or not each of those things has been tested by some test, then we can measure coverage. The basic coverage measure is

$$\text{Coverage} = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}} \times 100\%$$

where the 'coverage item' is whatever we have been able to count and see whether a test has exercised or used this item.

There is danger in using a coverage measure. 100% coverage does *not* mean 100% tested! Coverage techniques measure only one dimension of a multi-dimensional concept. Two different test cases may achieve exactly the same coverage but the input data of one may find an error that the input data of the other doesn't.

One drawback of code coverage measurement is that it measures coverage of what *has* been written, i.e. the code itself; it cannot say anything about the software that has *not* been written. If a specified function has not been implemented, specification-based testing techniques will reveal this. If a function was omitted from the specification, then experience-based techniques may find it. But structure-based techniques can only look at a structure which is already there.

Types of coverage

Test coverage can be measured based on a number of different structural elements in a system or component. Coverage can be measured at component-testing level, integration-testing level or at system- or acceptance-testing levels. For example, at system or acceptance level, the coverage items may be requirements, menu options, screens, or typical business transactions. Other coverage measures include things such as database structural elements (records, fields and sub-fields) and files. It is worth checking for any new tools, as the test tool market develops quite rapidly.

At integration level, we could measure coverage of interfaces or specific interactions that have been tested. The call coverage of module, object or procedure calls can also be measured (and is supported by tools to some extent).

We can measure coverage for each of the specification-based techniques as well:

- EP: percentage of equivalence partitions exercised (we could measure valid and invalid partition coverage separately if this makes sense);
- BVA: percentage of boundaries exercised (we could also separate valid and invalid boundaries if we wished);
- Decision tables: percentage of business rules or decision table columns tested;
- State transition testing: there are a number of possible coverage measures:
  - Percentage of states visited
  - Percentage of (valid) transitions exercised (this is known as Chow's 0-switch coverage)
  - Percentage of pairs of valid transitions exercised ('transition pairs' or Chow's 1-switch coverage) - and longer series of transitions, such as tran sition triples, quadruples, etc.
  - Percentage of invalid transitions exercised (from the state table)

The coverage measures for specification-based techniques would apply at whichever test level the technique has been used (e.g. system or component level).

When coverage is discussed by business analysts, system testers or users, it most likely refers to the percentage of requirements that have been tested by a set of tests. This may be measured by a tool such as a requirements management tool or a test management tool.

However, when coverage is discussed by programmers, it most likely refers to the coverage of code, where the structural elements can be identified using a tool, since there is good tool support for measuring **code coverage.** We will cover statement and decision coverage shortly.

Statements and decision outcomes are both structures that can be measured in code and there is good tool support for these coverage measures. Code coverage is normally done in component and component integration testing - if it is done at all. If someone claims to have achieved code coverage, it is important to establish exactly what elements of the code have been covered, as statement coverage (often what is meant) is significantly weaker than decision coverage or some of the other code-coverage measures.

How to measure coverage

For most practical purposes, coverage measurement is something that requires tool support. However, knowledge of the steps typically taken to measure coverage is useful in understanding the relative merits of each technique. Our example assumes an intrusive coverage measurement tool that alters the code by inserting **instrumentation:**

1 Decide on the structural element to be used, i.e. the coverage items to be counted.

2 Count the structural elements or items.

3 Instrument the code.

4 Run the tests for which coverage measurement is required.

5 Using the output from the instrumentation, determine the percentage of ele ments or items exercised.

Instrumenting the code (step 3) implies inserting code alongside each structural element in order to record when that structural element has been exercised. Determining the actual coverage measure (step 5) is then a matter of analyzing the recorded information.

Coverage measurement of code is best done using tools (as described in Chapter 6) and there are a number of such tools on the market. These tools can help to increase quality and productivity of testing. They increase quality by ensuring that more structural aspects are tested, so defects on those structural paths can be found. They increase productivity and efficiency by highlighting tests that may be redundant, i.e. testing the same structure as other tests (although this is not necessarily a bad thing, since we may find a defect testing the same structure with different data).

In common with all structure-based testing techniques, code coverage techniques are best used on areas of software code where more thorough testing is required. Safety-critical code; code that is vital to the correct operation of a system, and complex pieces of code are all examples of where structure-based

techniques are particularly worth applying. For example, DO178-B [RTCA] requires structural coverage for certain types of system to be used by the military. Structural coverage techniques should always be used in addition to specification-based and experience-based testing techniques rather than as an alternative to them.

Structure-based test case design

If you are aiming for a given level of coverage (say 95%) but you have not reached your target (e.g. you only have 87% so far), then additional test cases can be designed with the aim of exercising some or all of the structural elements not yet reached. This is structure-based test design. These new tests are then run through the instrumented code and a new coverage measure is calculated. This is repeated until the required coverage measure is achieved (or until you decide that your goal was too ambitious!). Ideally all the tests ought to be run again on the un-instrumented code.

We will look at some examples of structure-based coverage and test design for statement and decision testing below.

4.4.2 Statement coverage and statement testing

$$\text{Statement coverage} = \frac{\text{Number of statements exercised}}{\text{Total number of statements}} \times 100\%$$

Statement coverage is calculated by:

Studies and experience in the industry have indicated that what is considered reasonably thorough black-box testing may actually achieve only 60% to 75% statement coverage. Typical ad hoc testing is likely to be around 30% - this leaves 70% of the statements untested.

Different coverage tools may work in slightly different ways, so they may give different coverage figures for the same set of tests on the same code, although at 100% coverage they should be the same.

We will illustrate the principles of coverage on code. In order to simplify our examples, we will use a basic pseudo-code - this is not any specific programming language, but should be readable and understandable to you, even if you have not done any programming yourself.

For example, consider code sample 4.1.

```
READ A
READB
IFA>BTHENC = 0
ENDIF
```

Code sample 4.1

To achieve 100% statement coverage of this code segment just one test case is required, one which ensures that variable A contains a value that is greater

than the value of variable B, for example, A = 12 and B = 10. Note that here we are doing structural test *design* first, since we are choosing our input values in order ensure statement coverage.

Let's look at an example where we measure coverage first. In order to simplify the example, we will regard each line as a statement. (Different tools and methods may count different things as statements, but the basic principle is the same however they are counted.) A statement may be on a single line, or it may be spread over several lines. One line may contain more than one statement, just one statement, or only part of a statement. Some statements can contain other statements inside them. In code sample 4.2, we have two read statements, one assignment statement, and then one IF statement on three lines, but the IF statement contains another statement (print) as part of it.

```
1  READ A
2  READ B
3  C =A + 2*B
4  IF C> 50 THEN
5        PRINT large C
6  ENDIF
```

Code sample 4.2

Although it isn't completely correct, we have numbered each line and will regard each line as a statement. (Some tools may group statements that would always be executed together in a basic block which is regarded as a single statement.) However, we will just use numbered lines to illustrate the principles of coverage of statements (lines). Let's analyze the coverage of a set of tests on our six-statement program:

TEST SET 1 Test 1_1: A
= 2, B = 3 Test 1_2: A =
0, B = 25 Test 1_3: A =
47, B = 1

Which statements have we covered?

- In Test 1_1, the value of C will be 8, so we will cover the statements on lines 1 to 4 and line 6.
- In Test 1_2, the value of C will be 50, so we will cover exactly the same state ments as Test 1_1.
- In Test 1_3, the value of C will be 49, so again we will cover the same state ments.

Since we have covered five out of six statements, we have 83% statement coverage (with three tests). What test would we need in order to cover statement 5, the one statement that we haven't exercised yet? How about this one:

Test 1_4: A = 20, B = 25

This time the value of C is 70, so we will print 'Large C and we will have exercised all six of the statements, so now statement coverage = 100%. Notice that we measured coverage first, and then designed a test to cover the statement that we had not yet covered.

Note that Test 1_4 on its own is more effective- (towards our goal of achieving 100% statement coverage) than the first three tests together. Just taking Test 1_4 on its own is also more efficient than the set of four tests, since it has used only one test instead of four. Being more effective and more efficient is the mark of a good test technique.

4.4.3 Decision coverage and decision testing

A decision is an IF statement, a loop control statement (e.g. DO-WHILE or REPEAT-UNTIL), or a CASE statement, where there are two or more possible exits or outcomes from the statement. With an IF statement, the exit can either be TRUE or FALSE, depending on the value of the logical condition that comes after IF. With a loop control statement, the outcome is either to perform the code within the loop or not - again a True or False exit. Decision coverage is calculated by:

$$\text{Decision coverage} = \frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} \times 100\%$$

What feels like reasonably thorough functional testing may achieve only 40% to 60% decision coverage. Typical ad hoc testing may cover only 20% of the decisions, leaving 80% of the possible outcomes untested. Even if your testing seems reasonably thorough from a functional or specification-based perspective, you may have only covered two-thirds or three-quarters of the decisions. Decision coverage is stronger than statement coverage. It 'subsumes' statement coverage - this means that 100% decision coverage always guarantees 100% statement coverage. Any stronger coverage measure may require more test cases to achieve 100% coverage. For example, consider code sample 4.1 again.

We saw earlier that just one test case was required to achieve 100% statement coverage. However, decision coverage requires each decision to have had both a True and False outcome. Therefore, to achieve 100% decision coverage, a second test case is necessary where A is less than or equal to B. This will ensure that the decision statement 'IF A > B' has a False outcome. So one test is sufficient for 100% statement coverage, but two tests are needed for 100% decision coverage. Note that 100% decision coverage guarantees 100% statement coverage, but *not* the other way around!

```
1  READ A
2  READ B
3  C = A - 2 * B
4  IF C <0 THEN
5        PRINT "C negative"
6  ENDIF
```

Code sample 4.3

Let's suppose that we already have the following test, which gives us 100% statement coverage for code sample 4.3.

TEST SET 2
Test 2_1: A = 20, B = 15

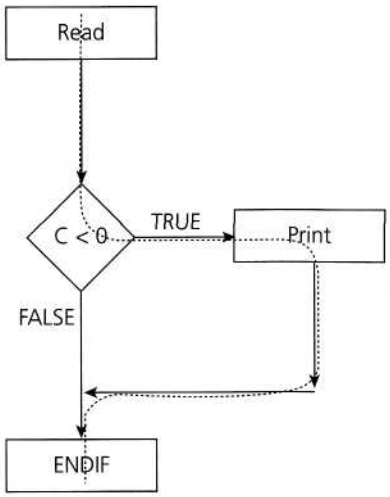


FIGURE 4.4 Control flow diagram for code sample 4.3

Which decision outcomes have we exercised with our test? The value of C is -10, so the condition 'C < 0' is True, so we will print 'C negative' and we have exercised the True outcome from that decision statement. But we have not exercised the decision outcome of False. What other test would we need to exercise the False outcome and to achieve 100% decision coverage?

Before we answer that question, let's have a look at another way to represent this code. Sometimes the decision structure is easier to see in a control flow diagram (see Figure 4.4).

The dotted line shows where Test 2_1 has gone and clearly shows that we haven't yet had a test that takes the False exit from the IF statement.

Let's modify our existing test set by adding another test:

TEST SET 2
Test 2_1: A = 20, B = 15
Test 2_2: A = 10, B = 2

This now covers both of the decision outcomes, True (with Test 2_1) and False (with Test 2_2). If we were to draw the path taken by Test 2_2, it would be a straight line from the read statement down the False exit and through the ENDIF. Note that we could have chosen other numbers to achieve either the True or False outcomes.

4.4.4 Other structure-based techniques

There are other structure-based techniques that can be used to achieve testing to different degrees of thoroughness. Some techniques are stronger (require more tests to achieve 100% coverage and therefore, have a greater chance of detecting defects) and others are weaker.

For example, branch coverage is closely related to decision coverage and at 100% coverage they give exactly the same results. Decision coverage measures the coverage of conditional branches; branch coverage measures the coverage of both conditional and unconditional branches. The Syllabus uses decision

coverage, as it is the source of the branches. Some coverage measurement tools may talk about branch coverage when they actually mean decision coverage.

Other control-flow code-coverage measures include linear code sequence and jump (LCSAJ) coverage, condition coverage, multiple condition coverage (also known as condition combination coverage) and condition determination coverage (also known as multiple condition decision coverage or modified condition decision coverage, MCDC). This technique requires the coverage of all conditions that can affect or determine the decision outcome.

Another popular, but often misunderstood, code-coverage measure is path coverage. Sometimes any structure-based technique is called 'path testing' [Patton, 2001]. However, strictly speaking, for any code that contains a loop, path coverage is impossible since a path that travels round the loop three times is different from the path that travels round the same loop four times. This is true even if the rest of the paths are identical. So if it is possible to travel round the loop an unlimited number of times then there are an unlimited number of paths through that piece of code. For this reason it is more correct to talk about 'independent path segment coverage' though the shorter term 'path coverage is frequently used.

Structure-based measures and related test design techniques are described in [BS7925-2]. Structure-based techniques are also discussed in [Copeland. 2003] and [Myers, 1979]. A good description of the graph theory behind structural testing can be found in [Jorgensen, 1995] and [Hetzel, 1988] also shows a structural approach. [Pol *et al,* 2001] describes a structure-based approach called an algorithm test.

4.5 EXPERIENCE-BASED TECHNIQUES

1 **Recall reasons for writing test cases based on intuition, experience and knowledge about common defects. (Kl)**

2 **Compare experience-based techniques with specification-based testing techniques. (K2)**

In this section we will look at two experience-based techniques, why and when they are useful, and how they fit with specification-based techniques.

Although it is true that testing should be rigorous, thorough and systematic, this is not all there is to testing. There is a definite role for non-systematic techniques, i.e. tests based on a person's knowledge, experience, imagination and intuition. The reason is that some defects are hard to find using more systematic approaches, so a good 'bug hunter' can be very creative at finding those elusive defects.

In this section, look for the definitions of the glossary terms: **error guessing** and **exploratory testing.**

4.5.1 Error guessing

Error guessing is a technique that should always be used as a complement to other more formal techniques. The success of error guessing is very much dependent on the skill of the tester, as good testers know where the defects are most likely to lurk. Some people seem to be naturally good at testing and others are good testers because they have a lot of experience either as a tester or working with a particular system and so are able to pin-point its weaknesses. This is why an error-guessing approach, used after more formal techniques have been applied to some extent, can be very effective. In using more formal techniques, the tester is likely to gain a better understanding of the system, what it does and how it works. With this better understanding, he or she is likely to be better at guessing ways in which the system may not work properly.

There are no rules for error guessing. The tester is encouraged to think of situations in which the software may not be able to cope. Typical conditions to try include division by zero, blank (or no) input, empty files and the wrong kind of data (e.g. alphabetic characters where numeric are required). If anyone ever says of a system or the environment in which it is to operate 'That could never happen', it might be a good idea to test that condition, as such assumptions about what will and will not happen in the live environment are often the cause of failures. A structured approach to the error-guessing technique is to list possible defects or failures and to design tests that attempt to produce them. These defect and failure lists can be built based on the tester's own experience or that of other people, available defect and failure data, and from common knowledge about why software fails.

4.5.2 Exploratory testing

Exploratory testing is a hands-on approach in which testers are involved in minimum planning and maximum test execution. The planning involves the creation of a test charter, a short declaration of the scope of a short (1 to 2 hour) time-boxed test effort, the objectives and possible approaches to be used.

The test design and test execution activities are performed in parallel typically without formally documenting the test conditions, test cases or test scripts. This does not mean that other, more formal testing techniques will not be used. For example, the tester may decide to use boundary value analysis but will think through and test the most important boundary values without necessarily writing them down. Some notes will be written during the exploratory-testing session, so that a report can be produced afterwards.

Test logging is undertaken as test execution is performed, documenting the key aspects of what is tested, any defects found and any thoughts about possible further testing. A key aspect of exploratory testing is learning: learning by the tester about the software, its use, its strengths and its weaknesses. As its name implies, exploratory testing is about exploring, finding out about the software, what it does, what it doesn't do, what works and what doesn't work. The tester is constantly making decisions about what to test next and where to spend the (limited) time.

This is an approach that is most useful when there are no or poor specifications and when time is severely limited. It can also serve to complement other, more formal testing, helping to establish greater confidence in the software. **In**

this way, exploratory testing can be used as a check on the formal test process by helping to ensure that the most serious defects have been found.

Exploratory testing is described in [Kaner, 2002] and [Copeland, 2003] Other ways of testing in an exploratory way ('attacks') are described b\ [Whittaker, 2002].

4.6 CHOOSING A TEST TECHNIQUE

1 List the factors that influence the selection of the appropriate test design technique for a particular kind of problem, such as the type of system, risk, customer requirements, models for use case modeling, requirements models or tester knowledge. (K2)

In this final section we will look at the factors that go into the decision about which techniques to use when.

Which technique is best? This is the wrong question! Each technique is good for certain things, and not as good for other things. For example, one of the benefits of structure-based techniques is that they can find things in the code that aren't supposed to be there, such as 'Trojan horses' or other malicious code. However, if there are parts of the specification that are missing from the code, only specification-based techniques will find that - structure-based techniques can only test what is there. If there are things missing from the specification and from the code, then only experience-based techniques would find them. Each individual technique is aimed at particular types of defect as well. For example, state transition testing is unlikely to find boundary defects.

The choice of which test techniques to use depends on a number of factors, including the type of system, regulatory standards, customer or contractual requirements, level of risk, type of risk, test objective, documentation available, knowledge of the testers, time and budget, development life cycle, use case models and previous experience of types of defects found.

Some techniques are more applicable to certain situations and test levels: others are applicable to all test levels.

This chapter has covered the most popular and commonly used software testing techniques. There are many others that fall outside the scope of the Syllabus that this book is based on. With so many testing techniques to choose from how are testers to decide which ones to use?

Perhaps the single most important thing to understand is that the best testing technique is no single testing technique. Because each testing technique is good at finding one specific class of defect, using just one technique will help ensure that many (perhaps most but not all) defects of that particular class are found. Unfortunately, it may also help to ensure that many defects of other classes are missed! Using a variety of techniques will therefore help ensure that a variety of defects are found, resulting in more effective testing.

So how can we choose the most appropriate testing techniques to use? The decision will be based on a number of factors, both internal and external.

The internal factors that influence the decision about which technique to use are:

- *Models used* - Since testing techniques are based on models, the models available (i.e. developed and used during the specification, design and implementation of the system) will to some extent govern which testing techniques can be used. For example, if the specification contains a state transition diagram, state transition testing would be a good technique to use.
- *Tester knowledge I experience* - How much testers know about the system and about testing techniques will clearly influence their choice of testing tech niques. This knowledge will in itself be influenced by their experience of testing and of the system under test.
- *Likely defects* - Knowledge of the likely defects will be very helpful in choos ing testing techniques (since each technique is good at finding a particular type of defect). This knowledge could be gained through experience of testing a previous version of the system and previous levels of testing on the current version.
- *Test objective* - If the test objective is simply to gain confidence that the soft ware will cope with typical operational tasks then use cases would be a sen sible approach. If the objective is for very thorough testing then more rigorous and detailed techniques (including structure-based techniques) should be chosen.
- *Documentation* - Whether or not documentation (e.g. a requirements specification) exists and whether or not it is up to date will affect the choice of testing techniques. The content and style of the documentation will also influence the choice of techniques (for example, if decision tables or state graphs have been used then the associated test techniques should be used).
- *Life cycle model* - A sequential life cycle model will lend itself to the use of more formal techniques whereas an iterative life cycle model may be better suited to using an exploratory testing approach.

The external factors that influence the decision about which technique to use are:

- *Risk* - The greater the risk (e.g. safety-critical systems), the greater the need for more thorough and more formal testing. Commercial risk may be influ enced by quality issues (so more thorough testing would be appropriate) or by time-to-market issues (so exploratory testing would be a more appropri ate choice).
- *Customer I contractual requirements* - Sometimes contracts specify par ticular testing techniques to use (most commonly statement or branch coverage).
- *Type of system* - The type of system (e.g. embedded, graphical, financial, etc.) will influence the choice of techniques. For example, a financial application involving many calculations would benefit from boundary value analysis.

- *Regulatory requirements* - Some industries have regulatory standards or guidelines that govern the testing techniques used. For example, the aircraft industry requires the use of equivalence partitioning, boundary value analy sis and state transition testing for high integrity systems together with state ment, decision or modified condition decision coverage depending on the level of software integrity required.
- *Time and budget* - Ultimately how much time there is available will always affect the choice of testing techniques. When more time is available we can afford to select more techniques and when time is severely limited we will be limited to those that we know have a good chance of helping us find just the most important defects.

CHAPTER REVIEW

Let's review what you have learned in this chapter.

From Section 4.1, you should now be able to differentiate between a test condition, a test case and a test procedure, and know that they are documented in a test design specification, a test case specification and a test procedure specification respectively. You should be able to write test cases that include expected results and that show clear traceability to the test basis (e.g. requirements). You should be able to translate test cases into a test procedure specification at the appropriate level of detail for testers and you should be able to write a test execution schedule for a given set of test cases that takes into account priorities as well as technical and logical dependencies. You should know the glossary terms **test cases, test case specification, test condition, test data, test procedure specification, test script** and **traceability.**

From Section 4.2 (categories of test design techniques), you should be able to give reasons why both specification-based (black-box) and structure-based (white-box) approaches are useful, and list the common techniques for each of these approaches. You should be able to explain the characteristics and differences between specification-based, structure-based and experience-based techniques. You should know the glossary terms **black-box test design techniques, experience-based test design techniques, specification-based test design techniques, structure-based test design techniques** and **white-box test design techniques.**

From Section 4.3, you should be able to write test cases from given software models using equivalence partitioning, boundary value analysis, decision tables and state transition diagrams. You should understand the main purpose of each of these four techniques, what level and type of testing could use each technique and how coverage can be measured for each of them. You should also understand the concept and benefits of use case testing. You should know the glossary terms **boundary value analysis, decision table testing, equivalence partitioning, state transition testing** and **use case testing.**

From Section 4.4, you should be able to describe the concept and importance of code coverage. You should be able to explain the concepts of statement and decision coverage and understand that these concepts can also be used at test levels other than component testing (such as business procedures at system test level). You should be able to write test cases from given control flows using statement testing and decision testing, and you should be able to assess statement and decision coverage for completeness. You should know the glossary terms **code coverage, decision coverage, statement coverage, structural testing, structure-based testing** and **white-box testing.**

From Section 4.5, you should be able to remember the reasons for writing test cases based on intuition, experience and knowledge about common defects and you should be able to compare experience-based techniques with specification-based techniques. You should know the glossary terms **error guessing** and **exploratory testing.**

From Section 4.6, you should be able to list the factors that influence the selection of the appropriate test design technique for a particular type of problem, such as the type of system, risk, customer requirements, models for use case modeling, requirements models or testing knowledge.