1 THE FUNDAMENTALS OF TESTING

Peter Morgan

If you were buying a new car, you would not expect to take delivery from the showroom with a scratch down the side of the vehicle. The car should have the right number of wheels (including a 'spare' for emergencies), a steering wheel, an engine and all the other essential components, and it should come with appropriate documentation, with all pre-sales checks completed and passed satisfactorily. The car you receive should be the car described in the sales literature; it should have the correct engine size, the correct colour scheme and whatever extras you have ordered, and performance in areas such as fuel consumption and maximum speed should match published figures. In short, a level of expectation is set by brochures, by your experience of sitting in the driving seat and probably by a test drive. If your expectations are not met, you will feel justifiably aggrieved.

This kind of expectation seems not to apply to new software installations; examples of software being delivered not working as expected, or not working at all, are common. Why is this? There is no single cause that can be rectified to solve the problem, but one important contributing factor is the inadequacy of the testing to which software applications are exposed.

Software testing is neither complex nor difficult to implement, yet it is a discipline that is seldom applied with anything approaching the necessary rigour to provide confidence in delivered software. Software testing is costly in human effort or in the technology that can multiply the effect of human effort, yet it is seldom implemented at a level that will provide any assurance that software will operate effectively, efficiently or even correctly.

This book explores the fundamentals of this important but neglected discipline to provide a basis on which a practical and cost-effective software testing regime can be constructed.

INTRODUCTION

In this opening chapter we have three very important objectives to achieve. First, we will introduce you to the fundamental ideas that underpin the discipline of software testing, and this will involve the use and explanation of some new terminology. Second, we will establish the

structure that we have used throughout the book to help you to use the book as a learning and revision aid. Third, we will use this chapter to point forward to the content of later chapters.

The key ideas of software testing are applicable irrespective of the software involved and any particular development methodology (waterfall, Agile etc.). Software development methodologies are discussed in detail in **Chapter 2.**

We begin by defining what we expect you to get from reading this chapter. The learning objectives below are based on those defined in the Software Foundation Certificate syllabus, so you need to ensure that you have achieved all of these objectives before attempting the examination.

Learning objectives

The learning objectives for this chapter are listed below. You can confirm that you have achieved these by using the self-assessment questions immediately after the learning objectives listed below, the 'Check of understanding' boxes distributed throughout the text and the example examination questions provided at the end of the chapter. The chapter summary will remind you of the key ideas.

The sections are allocated a K number to represent the level of understanding required for that section; where an individual topic has a lower K number than the section as a whole, this is indicated for that topic; for an explanation of the K numbers, see the **Introduction**.

What is testing? (K2)

- FL-1.1.1 Identify typical objectives of testing. (K1)
- FL-1.1.2 Differentiate testing from debugging.

Why is testing necessary? (K2)

- FL-1.2.1 Give examples of why testing is necessary.
- FL-1.2.2 Describe the relationship between testing and quality assurance and give examples of how testing contributes to higher quality.
- FL-1.2.3 Distinguish between error, defect, and failure.
- FL-1.2.4 Distinguish between the root cause of a defect and its effects.

Seven testing principles (K2)

- FL-1.3.1 Explain the seven testing principles.

Test process (K2)

- FL-1.4.1 Explain the impact of context on the test process.

- FL-1.4.2 Describe the test activities and respective tasks within the test process.
- FL-1.4.3 Differentiate the work products that support the test process.
- FL-1.4.4 Explain the value of maintaining traceability between the test basis and test work products.

The psychology of testing (K2)

- FL-1.5.1 Identify the psychological factors that influence the success of testing. (K1)
- FL-1.5.2 Explain the difference between the mindset required for test activities and the mindset required for development activities.

Self-assessment questions

The following questions have been designed to enable you to check your current level of understanding for the topics in this chapter. The answers are at the end of the chapter.

Question SA1 (K2)
Which of the following correctly describes the interdependence between an error, a defect and a failure?

a. An error causes a failure that can lead to a defect.
b. A defect causes a failure that can lead to an error.
c. An error causes a defect that can lead to a failure.
d. A failure causes an error that can lead to a defect.

Question SA2 (K2)
An online site where goods can be bought and sold has been implemented. Which one of the following could be the root cause of a defect?

a. Customers complain that the time taken to move to the ‘payments’ screen is too long.
b. The lead business analyst was not familiar with all possible permutations of customer actions.
c. Multiple customers can buy the **one** item that is for sale.
d. There is no project-wide defect tracking system in use.

Question SA3 (K2)
Which of the following illustrates the principle of defect clustering?

a. Testing everything in most cases is not possible
b. Even if no defects are found, testing cannot show correctness.

c. It is incorrect to assume that finding and fixing a large number of defects will ensure the success of a system.

d. A small subset of the code will usually contain most of the defects discovered during the testing phases.

WHY SOFTWARE FAILS

Examples of software failure are depressingly common. Here are some you may recognise:

- After successful test flights and air worthiness accreditation, problems arose in the manufacture of the Airbus A380 aircraft. Assembly of the large subparts into the completed aircraft revealed enormous cabling and wiring problems. The wiring of large subparts could not be joined together. It has been estimated that the direct or indirect costs of rectification were $6.1 billion. (Note: this problem was quickly fixed and the aircraft entered into service within 18 months of the cabling difficulties being identified.)
- When the UK Government introduced online filing of tax returns, a user could sometimes see the amount that a previous user earned. This was regardless of the physical location of the two applicants.
- In November 2005, information on the UK's top 10 wanted criminals was displayed on a website. The publication of this information was described in newspapers and on morning radio and television and, as a result, many people attempted to access the site. The performance of the website proved inadequate under this load and it had to be taken offline. The publicity created performance peaks beyond the capacity of the website.
- A new smartphone mapping application (app) was introduced in September 2012. Among many other problems, a museum was incorrectly located in the middle of a river, and Sweden's second city, Gothenburg, seemed to have disappeared from at least one map.
- Security breaches at the US military resulted in the payment details of many personnel (perhaps even 'all') being compromised, including names, addresses, email addresses and bank details.

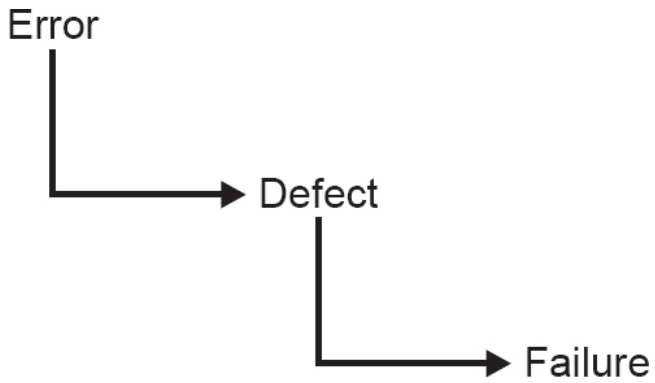
Perhaps these examples are not the same as the notorious final payment demands for 'zero pounds and zero pence' of some utilities customers in the 1970s. However, what is it that still makes them so startling? Is it a sense that something fairly obvious was missed? Is it the feeling that, expensive and important as they were, the systems were allowed to enter service before they were ready? Do you think these systems were adequately tested? Obviously, they were not but in this book we want to explore why this was the case and why these kinds of failure continue to plague us.

To understand what is going on we need to start at the beginning, with the people who design systems. Do they make mistakes? Of course they do. People make mistakes because they are fallible, but there are also many pressures that make mistakes more likely. Pressures such as deadlines, complexity of systems and organisations, and changing technology all bear down on designers of systems and increase the likelihood of defects in specifications, in designs and in

software code. Errors are where major system failures usually begin. An error is best thought of as ‘invisible’, or better perhaps as intangible – you cannot touch it. It is an incorrect thought, a wrong assumption, or a thing that is forgotten, or not considered. Only when something is written down can it become ‘a fault’ or a defect. So, an incorrect choice can lead to a document with a defect in it, which, if it is used to specify a component, can result in the component being faulty and this may exhibit incorrect behaviour. If this faulty component is built into a system, the system may fail. While failure is not always guaranteed, it is likely that errors in the thought processes as specifications are produced will lead to faulty components and faulty components will cause system failure.

An error (or mistake) leads to a defect (or fault), which can cause an observed failure (Figure 1.1).

Figure 1.1 Effect of an error



There are other reasons why systems fail. Environmental conditions such as the presence of radiation, magnetism, electronic fields or pollution can affect the operation of hardware and firmware and lead to system failure.

It is worth pointing out at this stage that not every apparent failure is a real failure – something appears to be a failure in the software, but the observed behaviour is correct. Perhaps the tester that created the test misunderstood what should happen in the precise circumstances. When an apparent failure in test is actually the application or system performing correctly, this is termed a false positive. Conversely, a false negative is where there is a real failure, but this is not identified as such and the test is seen as correct.

If we want to avoid failure, we must either avoid errors and faults or find them and rectify them. Testing can contribute to both avoidance and rectification, as we will see when we have looked at the testing process in a little more detail. One thing is clear: if we wish to identify errors through testing we need to begin testing as soon as we begin making errors – right at the beginning of the development process – and we need to continue testing until we are confident that there will be no serious system failures – right at the end of the development process.

Before we move on, let us just remind ourselves of the importance of what we are considering. Incorrect software can harm:

- people (e.g. by causing an aircraft crash in which people die, or by causing a hospital life support system to fail);
- companies (e.g. by causing incorrect billing, which results in the company losing money);
- the environment (e.g. by releasing chemicals or radiation into the atmosphere).

Software failures can sometimes cause all three of these at once. The scenario of a train carrying nuclear waste being involved in a crash has been explored to help build public confidence in the safety of transporting nuclear waste by train. A failure of the train's on-board systems, or of the signalling system that controls the train's movements, could lead to catastrophic results. This may not be likely (we hope it is not) but it is a possibility that could be linked with software failure. Software failures, then, can lead to:

- loss of money;
- loss of time;
- loss of business reputation;
- injury;
- death.

KEEPING SOFTWARE UNDER CONTROL

With all of the examples we have seen so far, what common themes can we identify? There may be several themes that we could draw out of the examples, but one theme is clear: either insufficient testing or the wrong type of testing was done. More and better software testing seems a reasonable aim, but that aim is not quite as simple to achieve as we might expect.

Exhaustive testing of complex systems is not possible

The launch of the smartphone app occurred at the same time as a new phone hardware platform – the new app was only available on the new hardware for what many would recognise as the market leader at that time. The product launch received extensive worldwide coverage, with the mapping app given a prominent place in launch publicity. In a matter of hours there were tens of thousands of users, and numbers quickly escalated, with many people wanting to see their location in the new app and see how this compared with (for example) Google Maps. Each phone user was an expert in his/her location – after all they lived there, and 'test cases' were generated showing that problems existed.

If every possible test had been run, problems would have been detected and rectified prior to the product launch. However, if every test had been run, the testing would still be running now, and the product launch would never have taken place; this illustrates one of the general principles of software testing, which are explained below. With large and complex systems, it will never be possible to test everything exhaustively; in fact, it is impossible to test even moderately complex systems exhaustively.

For the mapping app, it would be unhelpful to say that not enough testing was done; for this particular project, and for many others of similar complexity, that would certainly always be the case. Here the problem was that the right sort of testing was not done because the problems had not been detected.

Testing and risk

Risk is inherent in all software development. The system may not work or the project to build it may not be completed on time, for example. These uncertainties become more significant as the system complexity and the implications of failure increase. Intuitively, we would expect to test an automatic flight control system more than we would test a video game system. Why? Because the risk is greater. There is a greater probability of failure in the more complex system and the impact of failure is also greater. What we test, and how much we test it, must be related in some way to the risk. Greater risk implies more and better testing.

There is much more on risk and risk management in **Chapter 5**.

Testing and quality

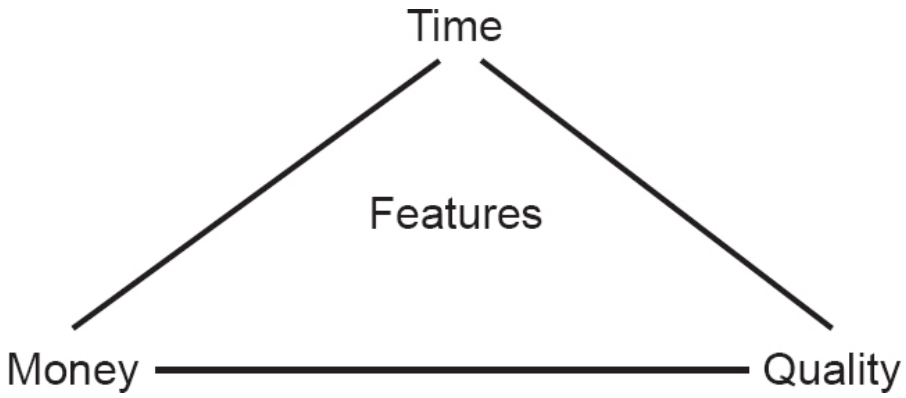
Quality is notoriously hard to define. If a system meets its users' requirements, that constitutes a good starting point. In the examples we looked at earlier, the online tax returns system had an obvious functional weakness in allowing one user to view another user's details. While the user community for such a system is potentially large and disparate, it is hard to imagine any user that would find that situation anything other than unacceptable. In the top 10 criminals example, the problem was slightly different. There was no failure of functionality in this case; the system was simply swamped by requests for access. This is an example of a non-functional failure, in that the system was not able to deliver its services to its users because it was not designed to handle the peak load that materialised after radio and TV coverage.

The problems with the Airbus A380 aircraft is an interesting story, because although completed subparts could not be brought together to build an entire aircraft, each of the subparts was 'correctly manufactured'. Aircraft are increasingly sophisticated, and the A380 aircraft has approximately 530 km of cables, 100,000 wires and 40,300 connectors. Software is used both to design the aircraft and in the manufacture. However, the large subparts were made in two different counties, with different versions of the software in each manufacturing base. So, when Airbus was bringing together two halves of the aircraft, the different software meant that the wiring on one did not match the wiring on the other. The cables could not meet up without being changed. Testing may have taken place, but it did not test something as straightforward as the versions of the design software and whether they were compatible (which in this case they were plainly not). Each large subpart was built according to its own version of the CATIA (Computer Aided Three-Dimensional Interactive Application) software. The result did not give an aircraft that could fly.

Of course, the software development process, like any other, must balance competing demands for resources. If we need to deliver a system faster (i.e. in less time), for example, it will usually cost

more. The items at the corners (or vertices) of the triangle of resources in Figure 1.2 are time, money and quality. These three items affect one another, and also influence the features that are or are not included in the delivered software.

Figure 1.2 Resources triangle



One role for testing is to ensure that key functional and non-functional requirements are examined before the system enters service and any defects are reported to the development team for rectification. Testing cannot directly remove defects, nor can it directly enhance quality. By reporting defects, it makes their removal possible and so contributes to the enhanced quality of the system. In addition, the systematic coverage of a software product in testing allows at least some aspects of the quality of the software to be measured. Testing is one component in the overall quality assurance activity that seeks to ensure that systems enter service without defects that can lead to serious failures.

Deciding when ‘enough is enough’

How much testing is enough, and how do we decide when to stop testing?

We have so far decided that we cannot test everything, even if we would wish to. We also know that every system is subject to risk of one kind or another and that there is a level of quality that is acceptable for a given system. These are the factors we will use to decide how much testing to do.

The most important aspect of achieving an acceptable result from a finite and limited amount of testing is prioritisation. Do the most important tests first so that at any time you can be certain that the tests that have been done are more important than the ones still to be done. Even if the testing activity is cut in half, it will still be true that the most important testing has been done. The most important tests will be those that test the most important aspects of the system: they will test the most important functions as defined by the users or sponsors of the system, and the most important non-functional behaviour, and they will address the most significant risks.

The next most important aspect is setting criteria that will give you an objective test of whether it is safe to stop testing, so that time and all the other pressures do not confuse the outcome. These criteria, usually known as completion criteria, set the standards for the testing activity by defining areas such as how much of the software is to be tested (this is covered in more detail in **Chapter 4**

) and what levels of defects can be tolerated in a delivered product (which is covered in more detail in **Chapter 5**).

Priorities and completion criteria provide a basis for planning (which will be covered in **Chapter 2** and **Chapter 5**) but the triangle of resources in Figure 1.2 still applies. In the end, the desired level of quality and risk may have to be compromised, but our approach ensures that we can still determine how much testing is required to achieve the agreed levels and we can still be certain that any reduction in the time or effort available for testing will not affect the balance – the most important tests will still be those that have already been done whenever we stop.

CHECK OF UNDERSTANDING

1. Describe the interaction between errors, defects and failures.
2. Software failures can cause losses. Give three consequences of software failures.
3. What are the vertices of the ‘triangle of resources’?

WHAT TESTING IS AND WHAT TESTING DOES

So far, we have worked with an intuitive idea of what testing is. We have recognised that it is an activity used to reduce risk and improve quality by finding defects, which is all true. There are indeed many different definitions of ‘testing’ when applied to software. Here is one that we have found useful – but don’t worry: you are not expected to remember it!

> Testing is the systematic and methodical examination of a work product using a variety of techniques, with the express intention of attempting to show that it does not fulfil its desired or intended purpose. This is undertaken in an environment that represents most nearly that which will be used in live operation.

Like other definitions of testing, it is not perfect. But it does point the way to material that will be covered in later chapters of this book, including:

- systematic – it has to be planned (**Chapter 5**);
- methodical – it is a process (throughout, but initially later in this chapter);
- work product – it is not just code that is examined (**Chapter 3**);
- variety of techniques (**Chapter 4**).

Any definition of testing has limitations! The definition depends upon the aim(s) or goals of that testing, or indeed the testing of that application. For not all testing has the same aim, and the aim of testing can vary through the Software Development Life Cycle.

So, what does testing aim to do? Here are some of the principle reasons:

- to examine work products (note it is ‘work product’; as well as code, this can include requirements, user stories and the overall design, amongst other items);
- to check if all the requirements have been satisfied;
- to see whether the item under test is complete, and works as the users and other stakeholders expect;
- to instil confidence in the quality of the item under test;
- to prevent defects – testing can not only ‘catch’ defects, but sometimes stop them happening in the first place;
- to find failures and defects and prevent these reaching the production version of the software. Often this is the **first** item that people will list;
- to give sufficient information to enable decision makers to make decisions, perhaps about whether the software product is suitable for release;
- to reduce the level of risk of inadequate software quality (e.g. previously undetected failures occurring in operation);
- to comply with contractual, legal or regulatory requirements or standards. Some standards require, for example, 80 per cent decision coverage to be shown by testing.

Testing can therefore be a multifaceted activity. However, we need to understand a little more about how software testing works in practice before we can think about how to implement effective testing.

Before that though, we will sum up what testing is, by looking at a few of the key terms or definitions that are relevant to this opening chapter. Remember, the keywords that are given in the syllabus for this section of material can be used in examination questions, where a clear understanding of the meaning of the terms is required. Testing relies upon an understanding of what the system, application or utility is meant to achieve. This is usually, but not always, written down, perhaps in a requirements document, in a description of the house style of web applications in the company, or in the interface definitions that are required for external systems. This body of knowledge is termed the **test basis**; testing has the test basis as a key starting point for what to test and how to test. Within the test basis, there are descriptions of what should happen under certain circumstances – ‘if the user-name and password combination is incorrect, display an error message, and request that the operator tries again, and if the details are incorrect on the third attempt, lock the user account’. Such descriptions are called **test conditions** – what will happen as a consequence of previous choices or actions. Test conditions give the circumstances, but not usually the values required to run a test. The user-name or password may indeed be incorrect, but the specific values to be used are given in a **test case**. A test case is derived from one or more test conditions and describes in some detail what is required to enter into the application or system, and most importantly, what the expected outcome is, so that we know whether the test is successful or not (has ‘passed’ or ‘failed’). In many instances, there are preconditions that must be in place before one or more particular test cases can be run and actions that are to be done when the test case or cases have been run. This is called a **test procedure**. Here are the definitions of these four terms, taken from the ISTQB glossary

Test basis:	The body of knowledge used as the basis for test analysis and design.
Test condition:	An aspect of the test basis that is relevant in order to achieve specific test objectives.
Test case:	A set of preconditions, inputs, actions (where applicable), expected results and postconditions, developed based on test conditions.
Test procedure:	A sequence of test cases in execution order, and any associated actions that may be required to set up the initial pre-conditions and any wrap up activities post execution.

Testing and debugging

Testing and debugging are different kinds of activity, both of which are very important. Debugging is the process that developers go through to identify the cause of bugs, or defects in code, and undertake corrections. Ideally, some check of the correction is made, but this may not extend to checking that other areas of the system have not been inadvertently affected by the correction. Testing, on the other hand, is a systematic exploration of a component or system with the main aim of finding and reporting defects. Testing does not include correction of defects – these are passed on to the developer to correct. Testing does, however, ensure that changes and corrections are checked for their effect on other parts of the component or system.

Usually, developers will have undertaken some initial testing (and taken appropriate corrective action as necessary – debugging) to raise the level of quality of the component or system to a level that is worth testing; that is, a level that is sufficiently robust to enable rigorous testing to be performed. Debugging does not give confidence that the component or system meets its requirements completely. Testing makes a rigorous examination of the behaviour of a component or system and reports all defects found for the development team to correct. Testing then repeats enough tests to ensure that defect corrections have been effective. So, both are needed to achieve a quality result.

While it is generally true that testers test, and developers undertake debugging action, this is not always so clear-cut. In some software development methodologies (typically Agile development, but not restricted to this), testers are routinely involved in both component testing and debugging activities.

Defects, effects and root causes

We talked earlier about the differences and interconnections between an error, a defect and a failure. There is a similar relationship between the root cause of a defect, a description of the defect and the effect (or manifestation) of the defect. Indeed, some organisations routinely undertake root cause analysis of defects, with the aim of preventing similar problems happening in the future. It is important to realise that similar or even identical defects can have completely different root causes.

The defect is what is wrong (an incorrect calculation of interest rates), the effect is how this appears (an angry customer who is charged too much), and the root cause is why the defect came

about (an incorrect understanding by the business lead of how long-term interest rates are to be calculated). So, in considering defects it is important to distinguish between the root cause (the 'why') and the effects (the 'what happened') of a defect.

Testing and quality assurance

Testing is part of Quality Control, which itself is part of Quality Management. However, as we shall see, Quality Control encompasses MORE than testing. Quality Management comprises both Quality Control and Quality Assurance. Quality Assurance is about making sure that processes are undertaken correctly. If processes are carried out correctly, there is a greater likelihood that the end product will be better. Quality Control is about seeing whether the desired level of quality is being achieved (and if not, doing something about it). So testing is part of Quality Control (checking the quality of something – in this case, the software under test), but it does not necessarily have a part to play in courses of action if the quality does not match the desired level. And of course, checking quality can be undertaken in ways other than testing (e.g. questionnaires can be used).

Static testing and dynamic testing

Static testing is the term used for testing when the code is not exercised. This may sound strange but remember that failures often begin with a human error, namely a wrong way of thinking or an incorrect assumption (an error) when producing a document such as a specification (which will then have a defect in it). We need to test as early as possible because errors are much cheaper to fix than defects or failures (as you will see). We discussed earlier that errors are intangible, but the earlier we find something that is incorrect, the easier (and cheaper) it is to fix. That is why testing should start as early as possible (another basic principle explained in more detail later in this chapter). Static testing involves techniques such as reviews, which can be effective in preventing defects in the resulting software; for example, by removing ambiguities, omissions and faults from specification documents. This a topic in its own right and is covered in detail in **Chapter 3**. Dynamic testing is the kind that exercises the program under test with some test data, so we speak of test execution in this context. The discipline of software testing encompasses both static and dynamic testing.

Testing as a process

We have already seen that there is much more to testing than test execution. Before test execution, there is some preparatory work to do to design the tests and set them up; after test execution, there is some work needed to record the results and check whether the tests are complete. Even more important than this is deciding what we are trying to achieve with the testing and setting clear objectives for each test. A test designed to give confidence that a program functions according to its specification, for example, will be quite different from one designed to find as many defects as possible. We define a test process to ensure that we do not miss critical steps and that we do things in the right order. We will return to this important topic later, when we explain a generalised test process in detail.

Testing as a set of techniques

The final challenge is to ensure that the testing we do is effective testing. It might seem paradoxical, but a good test is one that finds a defect if there is one present. A test that finds no defect has consumed resources but added no value; a test that finds a defect has created an opportunity to improve the quality of the product. How do we design tests that find defects? We actually do two things to maximise the effectiveness of the tests. First, we use well-proven test design techniques, and a selection of the most important of these is explained in detail in **Chapter 4**. The techniques are all based on certain testing principles that have been discovered and documented over the years, and these principles are the second mechanism we use to ensure that tests are effective. Even when we cannot apply rigorous test design for some reason (such as time pressures), we can still apply the general principles to guide our testing. We turn to these next.

CHECK OF UNDERSTANDING

1. Describe static testing and dynamic testing.
2. What is debugging?
3. What other elements apart from ‘test execution’ are included in ‘testing’?

GENERAL TESTING PRINCIPLES

Testing is a very complex activity, and the software problems described earlier highlight that it can be difficult to do well. We now describe some general testing principles that help testers, principles that have been developed over the years from a variety of sources. These are not all obvious, but their purpose is to guide testers and prevent the types of problems described previously. Testers use these principles with the test techniques described in **Chapter 4**.

Testing shows the presence, not absence, of defects

Running a test through a software system can only show that one or more defects exist. Testing cannot show that the software is error free. Consider whether the top 10 wanted criminals website was error free. There were no functional defects, yet the website failed. In this case the problem was non-functional and the absence of defects was not adequate as a criterion for release of the website into operation.

In **Chapter 2** we will discuss retesting, when a previously failed test is rerun to show that under the same conditions, the reported problem no longer exists. In this type of situation, testing can show that one particular problem no longer exists.

Although there may be other objectives, usually the main purpose of testing is to find defects. Therefore, tests should be designed to find as many defects as possible.

Exhaustive testing is impossible

If testing finds problems, then surely you would expect more testing to find additional problems, until eventually we would have found them all. We discussed exhaustive testing earlier when looking at the smartphone mapping app and concluded that for large complex systems, exhaustive testing is not possible. However, could it be possible to test small pieces of software exhaustively and only incorporate exhaustively tested code into large systems?

> Exhaustive testing – a test approach in which all possible data combinations are used. This includes implicit data combinations present in the state of the software/data at the start of testing.

Consider a small piece of software where one can enter a password, specified to contain up to three characters, with no consecutive repeating entries. Using only Western alphabetic capital letters and completing all three characters, there are 26 × 26 × 26 input permutations (not all of which will be valid). However, with a standard keyboard there are not 26 × 26 × 26 permutations, but a much higher number: 256 × 256 × 256, or 2^{24}. Even then, the number of possibilities is higher. What happens if three characters are entered, and the ‘delete last character’ right arrow key removes the last two? Are special key combinations accepted, or do they cause system actions (Ctrl + P, for example)? What about entering a character, and waiting 20 minutes before entering the other two characters? It may be the same combination of keystrokes, but the circumstances are different. We can also include the situation where the 20-minute break occurs over the change-of-day interval. It is not possible to say whether there are any defects until all possible input combinations have been tried.

Even in this small example, there are many, many possible data combinations to attempt. The number of possible combinations using a smartphone might be significantly less, but it is still large enough to be impractical to use all of them.

Unless the application under test (AUT) has an extremely simple logical structure and limited input, it is not possible to test all possible combinations of data input and circumstances. For this reason, risk and priorities are used to concentrate on the most important aspects to test. Both ‘risk’ and ‘priorities’ are covered later in more detail. Their use is important to ensure that the most important parts are tested.

Early testing saves time and money

When discussing why software fails, we briefly mentioned the idea of early testing. This principle is important because, as a proposed deployment date approaches, time pressure can increase dramatically. There is a real danger that testing will be squeezed, and this is bad news if the only testing we are doing is after all the development has been completed. The earlier the testing activity is started, the longer the elapsed time available. Testers do not have to wait until software is available to test.

Work products are created throughout the Software Development Life Cycle (SDLC), and we talk about these different work products later in this chapter. As soon as these are ready, we can test

them. In **Chapter 2**, we will see that requirement documents are the basis for acceptance testing, so the creation of acceptance tests can begin as soon as requirement documents are available. As we create these tests, they will highlight the contents of the requirements. Are individual requirements testable? Can we find ambiguous or missing requirements?

Many problems in software systems can be traced back to missing or incorrect requirements. We will look at this in more detail when we discuss reviews in **Chapter 3**. The use of reviews can break the 'error–defect–failure' cycle. In early testing, we are trying to find errors and defects before they are passed to the next stage of the development process. Early testing techniques are attempting to show that what is produced as a system specification, for example, accurately reflects that which is in the requirement documents. Ed Kit discusses identifying and eliminating defects at the part of the SDLC in which they are introduced.[1] If an error/defect is introduced in the coding activity, it is preferable to detect and correct it at this stage. If a problem is not corrected at the stage in which it is introduced, this leads to what Kit calls 'errors of migration'. The result is rework. We need to rework not just the part where the mistake was made, but each subsequent part where the error has been replicated. A defect found at acceptance testing where the original mistake was in the requirements will require several work products to be reworked, and subsequently to be retested.

Studies have been done on the cost impacts of errors at the different development stages. While it is difficult to put figures on the relative costs of finding defects at different levels in the SDLC, Table 1.1 does concentrate the mind!

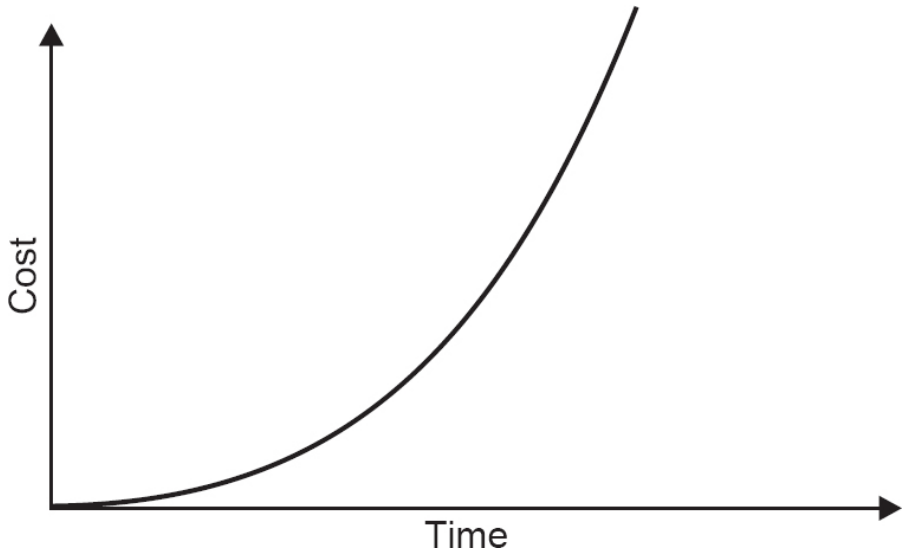
Table 1.1 Comparative cost to correct errors

Stage error is found	Comparative cost
Requirements	$1
Coding	$10
Program testing	$100
System testing	$1,000
User acceptance testing	$10,000
Live running	$100,000

This is known as the cost escalation model.

What is undoubtedly true is that the graph of the relative cost of early and late identification /correction of defects rises very steeply (Figure 1.3).

Figure 1.3 Effect of identification time on cost of errors



The earlier a problem (defect) is found, the less it costs to fix.

The objectives of various stages of testing can be different. For example, in the review processes, we may focus on whether the documents are consistent and no defects have been introduced when the documents were produced. Other stages of testing can have other objectives. The important point is that testing has defined objectives.

One of the drivers behind the push to Agile development methodologies is to enable testing to be incorporated throughout the software build process. This is nothing more than the ‘early testing’ principle.

Defects cluster together

Problems do occur in software. It is a fact. Once testing has identified (most of) the defects in a particular application, it is at first surprising that the spread of defects is not uniform. In a large application, it is often a small number of modules that exhibit the majority of the problems. This can be for a variety of reasons, some of which are:

- system complexity;
- volatile code;
- the effects of change on change;
- development staff experience;
- development staff inexperience.

This is the application of the Pareto principle to software testing: approximately 80 per cent of the problems are found in about 20 per cent of the modules. It is useful if testing activity reflects this spread of defects, and targets areas of the application under test where a high proportion of defects can be found. However, it must be remembered that testing should not concentrate exclusively on these parts. There may be fewer defects in the remaining code, but testers still need to search diligently for them.

Be aware of the pesticide paradox

Running the same set of tests continually will not continue to find new defects. Developers will soon know that the test team always tests the boundaries of conditions, for example, so they will learn to test these conditions themselves before the software is delivered. This does not make defects elsewhere in the code less likely, so continuing to use the same test set will result in decreasing the effectiveness of the tests. Using other techniques will find different defects.

For example, a small change to software could be specifically tested and an additional set of tests performed, aimed at showing that no additional problems have been introduced (this is known as regression testing). However, the software may fail in production because the regression tests are no longer relevant to the requirements of the system or the test objectives. Any regression test set needs to change to reflect business needs, and what are now seen as the most important risks. Regression testing will be covered in more detail in **Chapter 2**.

Testing is context dependent

Different testing is necessary in different circumstances. A website where information can merely be viewed will be tested in a different way to an ecommerce site, where goods can be bought using credit/debit cards. We need to test an air traffic control system with more rigour than an application for calculating the length of a mortgage.

Risk can be a large factor in determining the type of testing that is needed. The higher the possibility of losses, the more we need to invest in testing the software before it is implemented. A fuller discussion of risk is given in **Chapter 5**.

For an ecommerce site, we should concentrate on security aspects. Is it possible to bypass the use of passwords? Can ‘payment’ be made with an invalid credit card, by entering excessive data into the card number field? Security testing is an example of a specialist area, not appropriate for all applications. Such types of testing may require specialist staff and software tools. Test tools are covered in more detail in **Chapter 6**.

Absence-of-errors is a fallacy

Software with no known errors is not necessarily ready to be shipped. Does the application under test match up to the users’ expectations of it? The fact that no defects are outstanding is not a good reason to ship the software.

Before dynamic testing has begun, there are no defects reported against the code delivered. Does this mean that software that has not been tested (but has no outstanding defects against it) can be shipped? We think not.

CHECK OF UNDERSTANDING

1. Why is ‘zero defects’ an insufficient guide to software quality?
2. Give three reasons why defect clustering may exist.

3. Briefly justify the idea of early testing.

TEST PROCESS

We previously determined that testing is a process, discussed above. It would be easy to think that testing is thus always the same. However, this is not true.

Test process in context

Variations between organisations, and indeed projects within the same organisation, can have an influence on how testing is carried out, and on the specific test process that is used. Specific matters, or contextual factors, that will affect the test process can be many and varied, so please do not assume that the factors listed in the syllabus are the **only** factors. Many of these are covered in more detail in later chapters of this book. We give those in the syllabus here, and remember, you could be examined on these:

- the Software Development Life Cycle and project methodologies that are in use;
- test levels and test types being considered;
- product and project risks (if there is potential loss of life, you will generally ‘test more’);
- the business domain (online games are possibly tested in different ways to a billing system in a utility company);
- operational constraints, which could include the following, but also other matters:
  - budgets and other resources (including staffing levels);
  - timescale, and whether there are time-to-market constraints;
  - complexity (however this is measured);
  - any contractual or regulation requirements; for example both motor manufacturing and the pharmaceutical industries have special industry-wide regulations.
- any policies and practices that are specific to the organisation;
- required standards, both internal and external.

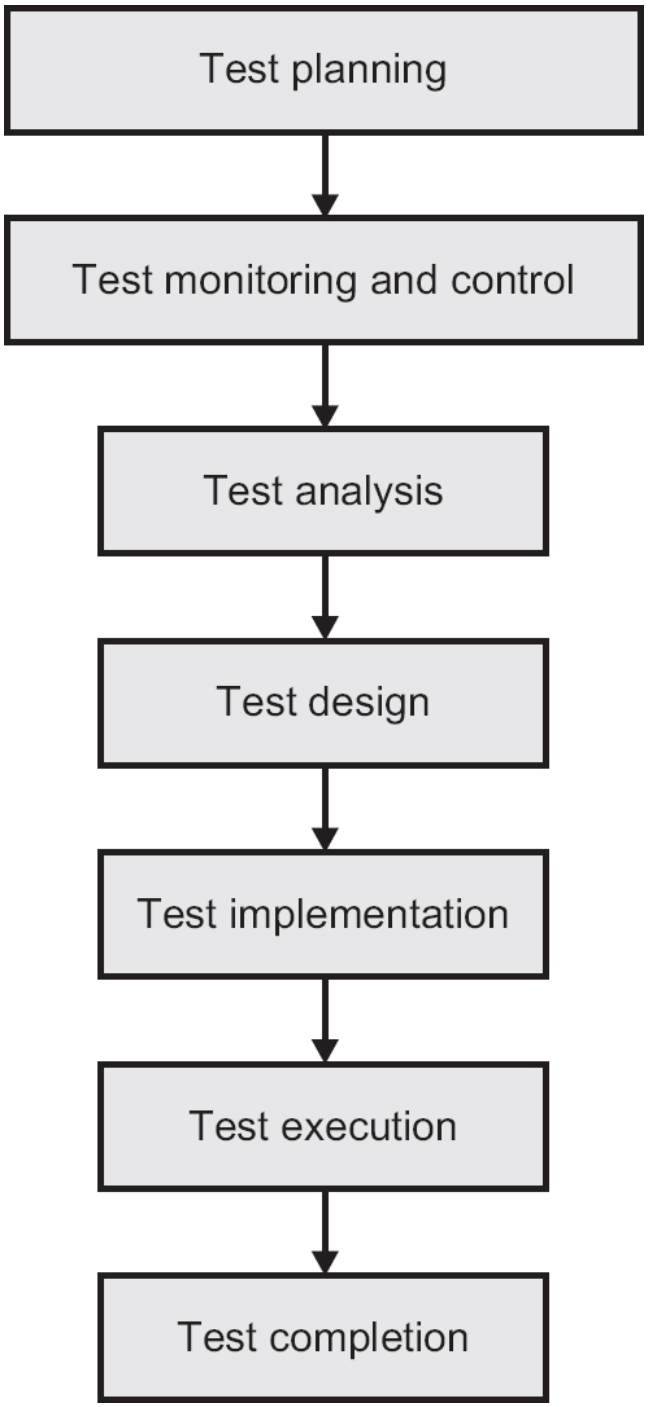
Test activities and tasks

The most visible part of testing is running one or more tests: test execution. We also have to prepare for running tests, analyse the tests that have been run and see whether testing is complete. Both planning and analysing are very necessary activities that enhance and amplify the benefits of the test execution itself. It is no good testing without deciding how, when and what to test. Planning is also required for the less formal test approaches such as exploratory testing, covered in more detail in **Chapter 4**. We are describing a generalised test process – as was said earlier, this will not be the same on every project within an organisation, nor between different organisations. Not all of the activity groups that are described will be recognised as separate entities in some projects, or within certain organisations.

So, there are the following activity groups in the test process that we will describe in more detail (Figure 1.4):

1. test planning;
2. test monitoring and control;
3. test analysis;
4. test design;
5. test implementation;
6. test execution;
7. test completion.

Figure 1.4 A generalised test process

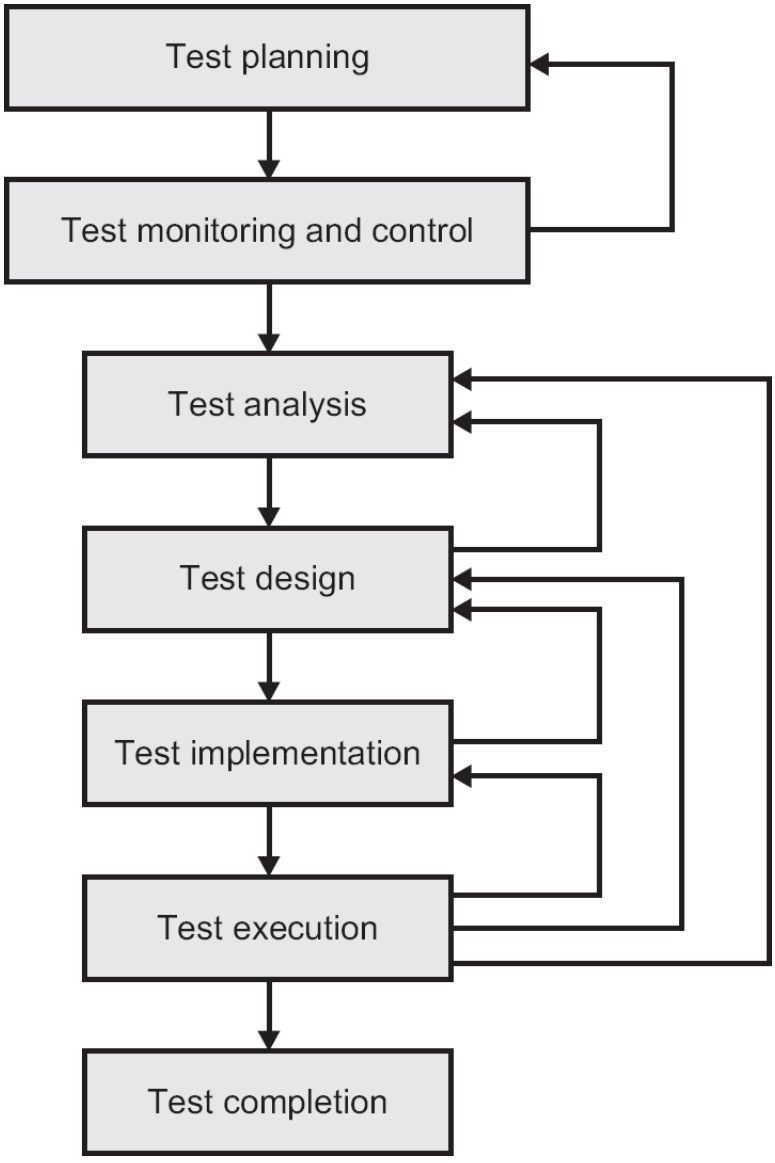


Although the main activity groups are in a broad sequence, they are not undertaken in a rigid way. The first two groups are overarching, and the other groups are both planned and monitored, and what has been undertaken may have a bearing upon other actions, some of which have been completed. An earlier activity group may need to be revisited. A defect found in test execution can sometimes be resolved by adding functionality that was originally not present (either missing, or the new functionality is needed to make the other part correct). The new features themselves have to be tested, so even though implementation and execution are in progress, the ‘earlier’ activity groups of test analysis and test design have to be performed for the new features (Figure 1.5). The activity groups may overlap or be performed concurrently. It could be that test design is being undertaken for one test level at the same time as test execution is underway for another test level on the same project (test execution for system testing and test design for acceptance testing, for example). Please note, not all possible interactions between later and earlier groups have been

included, and all of the latter five stages have interaction between **both** test planning and test monitoring and control

---

Figure 1.5 Iteration of activities



We sometimes need to do two or more of the main activities in parallel. Time pressure can mean that we begin test execution before all tests have been designed.

Test planning

Planning is determining what is going to be tested, and how this will be achieved. It is where we draw a map; how activities will be done; and who will do them. Test planning is also where we define the test completion criteria. Completion criteria are how we know when testing is finished. It is where the objectives of testing are set, and where the approach for meeting test objectives (within any context-based limitations) are set. A test plan is created and schedules are drawn up to

enable any internal or external deadlines to be met. Plans and schedules may be amended, depending upon how other activities are progressing – amendments take place as a result of monitoring and control activities. Planning is described in more detail in **Chapter 5**.

Test monitoring and control

Test monitoring and control go together. Monitoring is concerned with seeing if what has been achieved is what was expected to be done at this point in time, whereas control is taking any necessary action to meet the original or revised objectives as given in the test plan. Monitoring and control are supported by referring to the exit criteria or completion criteria for different stages of testing – this is sometimes referred to as the definition of 'done' in some project life cycles. Looking at exit criteria for test execution for a stage of testing may include:

- checking test results to see if the required test coverage has been achieved;
- determining the component or system quality, by looking at both test results and test logs;
- seeing if more tests are required (for example, if there are not enough tests to reach the level of risk coverage that is required).

Progress against the (original or revised) test plan is communicated to the necessary people (project sponsor, user stakeholders, the development team) in test progress reports. These will usually detail any actions being put in place to enable some milestones to be met earlier than would otherwise be the case or help to inform any decisions that stakeholders need to make. Test monitoring and control are covered more fully in **Chapter 5**.

Test analysis

Test analysis examines the test basis (which is usually written but not always confined to a single document), to identify what to test. Testable parts are identified, and test conditions drawn up. Major activities in the test analysis activity group are as follows

- examining the test basis relevant for the testing to be carried out at this time:
  - requirements documents, functional requirements, business requirements, system requirements or other items (use cases, user stories) that detail both functional **and** non-functional component or system behaviour;
  - design or implementation detail (system or software architecture, entity-relationship diagrams, interface specifications) that define the component or system structure;
  - implementation information for the component or system, including code, database metadata and queries;
  - risk analysis reports, which may consider both functional and non-functional aspects, and the structure of the software to be tested.
- looking at the test basis looking for defects of various types:
  - ambiguities;
  - things that have been missed out;

    - inconsistencies;
    - inaccuracies;
    - contradictions;
    - unnecessary or superfluous statements.
- identifying features and feature sets to be tested;
- identifying and prioritising test conditions for each feature or set of features (based on the test basis). Prioritisation is based on functional, non-functional and structural factors, other matters (both business and technical) and the levels of risk;
- capturing traceability in both directions ('bi-directional traceability') between the test basis and test conditions.

It can be both appropriate and very helpful to use some of the test techniques that are detailed in **Chapter 4** in the test analysis activity. Test techniques (black box, white box and experience-based techniques) can assist to define more precise and accurate test conditions and include important but less obvious test conditions.

Sometimes, test conditions produced as part of test analysis are used as test objectives in test charters. Test charters can be important in exploratory testing, discussed as part of experience-based test techniques in **Chapter 4**.

The test analysis activity can identify defects in the test basis. This can be especially important when there is no additional review process in place, or where the test process is closely aligned to the review process. Test design activity by its very nature looks at the test basis (including requirements), so it is appropriate to see whether requirements are consistent, clearly expressed and not missing anything. The work of test analysis can also rightly ask whether formal requirements documents have included customer, user and other stakeholder needs. Some development methodologies involve creating test conditions and test cases prior to any coding taking place. Examples of such methodologies include behaviour driven development (BDD) and acceptance test driven development (ATDD).

Test design

Test analysis answers the question 'What to test?', while test design answers the question 'How to test?' It is during the test design activity that test conditions are used to create test cases. This may also use test techniques, and the very process of creating test cases can identify defects. As we discussed earlier, a test case not only describes what to do to see if the test condition is correct but also what the expected result should be. Creating two or more test cases for a test condition can identify actions where the expected result is not clear – in the example we used earlier, are there different error messages if the user name is incorrect and the password is incorrect? If the requirements, specification or other documents are not clear, there is potential for misunderstandings!

Activities in the test design activity group can be summarised as follows:

- using test conditions to create test cases, and prioritising both test cases and sets of test cases;
- identifying any test data to be used with the test cases;
- designing the test environment if necessary, and identifying any other items that are needed for testing to take place (infrastructure and any tools required);
- detailing bi-directional traceability between test basis, test conditions, test cases and test procedures – building upon the traceability that was detailed above under test analysis.

Test implementation

We saw that test design asks ‘How to test?’ Test implementation asks ‘Do we have everything in place to run the tests?’ It is the link between test design and running the test, or test execution. We have the tests ready; how can we run them? Test implementation is not always a separate activity; it can be combined with test design, or, in exploratory testing and some other kinds of experienced-based testing, test design and test implementation may take place and be documented as part of the running of tests. In exploratory testing, tests are designed, implemented and executed simultaneously.

As we get ready for test execution, the test implementation main activities are as follows:

- creating and prioritising test procedures and possibly creating automated test scripts;
- creating test suites from the test procedures and from automated test scripts if there are any;
- ordering test suites in a test execution schedule, so that it makes efficient use of resources (it is often the case that a test execution schedule will **create** an order, **amend** the order and then **delete** it – all done to run efficiently);
- building the test environment with anything extra in place (test harnesses, simulators, dummy third-party interfaces, service virtualisation and any other infrastructure items). The test environment has then to be checked, to ensure that all is ready to start testing;
- preparing test data and checking that it has been loaded into the test environment;
- checking and updating the bi-directional traceability between the test basis, test conditions, test cases, test procedures and now to include test suites.

Test execution

Test execution involves running tests, and is where (what are often seen as) the most visible test activities are undertaken. Test suites are run as detailed in the test execution schedule (both of which were created as part of test implementation). Execution includes the following activities:

- Recording the identification and version of what is being tested: test items or test object, test tool(s) and any testware that are in use. This information is important if any defects are to be raised.
- Running tests, either manually or using test execution tools.

- Comparing actual and expected results (this may be done by any test execution tool(s) being used).
- Looking at instances where the actual result and the expected result differ to determine the possible cause. This may be a defect in the software under test, but could also be that the expected results were incorrect, or the test data was not quite correct.
- Reporting defects based on the failures that were found in the testing.
- Recording the result of each test (pass, fail, blocked).
- Repeating tests where the software has been corrected, the test data has been changed or as part of regression testing).
- Confirming or amending the bi-directional traceability between the test basis, the test conditions, test cases, test procedures and expected results.

Test completion

Testing at this stage has finished, and test completion activities collect data so that lessons can be learned, testware reused in future projects and so on. These activities may occur at the time that the software system is released, but could also be at other significant project milestones – the project is completed (or even cancelled), a test level is completed or an Agile project iteration is finished (where test completion activities may be part of the iteration retrospective meeting). The key here is to make sure that information is not lost (including the experiences of those involved). Activities in the test completion group are summarised below:

- Checking that defect reports are all closed as necessary. This may result in the raising of change requests or creating product backlog items for any that remain unresolved when test execution activities have ended.
- Creating a test summary report, to communicate the results of the testing activities. This is usually for the benefit of the project stakeholders.
- Closing down and saving (‘archiving’) the test environment, any test data, the test infrastructure and other testware. These may need to be reused at a later date.
- Handing over the testware to those who will maintain the software in the future or any other projects or other stakeholders to whom it could be beneficial.
- Analysing lessons learned from testing activities that have been completed, which will hopefully drive changes in future iterations, releases and projects.
- Using the information that has been gathered to make the testing process better – to improve test process maturity.

Agile methodologies and a generalised test process

Thus far we have concentrated on ‘traditional’ methodologies when discussing the test process. We now want to focus on Agile methodologies.

The use of Agile methodologies and the relationship with a test process is not a syllabus topic for the Foundation Certificate, but in the following discussion our understanding of both a generalised test process and Agile methodologies will grow.

We use the term ‘Agile methodologies’ because there is not a single variant or ‘flavour’ of Agile. Any distinctions between these are unimportant here, but the principles are important. Agile Software Development Life Cycles aim to have frequent deliveries of software, where each iteration or sprint will build on any previously made available. The aim is to have software that could be implemented into the production environment at the end of each sprint – although a delivery into production might not happen as frequently as this.

From this brief introduction to Agile, it follows that an Agile project has a beginning and an end, but the processes that go between these stages can be repeated many times. Agile projects can be successfully progressing over the course of many months, or even years, with a continuous stream of (same length) iterations producing production-quality software, typically every two, three or four weeks. In terms of the test process, there is a part of the test planning stage that takes place at the start of the project, and the test completion activities take place at the end of the project. However, some test planning, and all of the middle five stages of the test process we have described above are present in each and every sprint.

Some test planning activities take place at the beginning of the project. This typically includes resourcing activities, some outline planning on the length of the project, and an initial attempt at ordering the features to be implemented (although this could change significantly as the project progresses, sprint by sprint). Infrastructure planning, together with the identification and provision of any specific testing tools is usually undertaken at this stage, together with a clear understanding within the whole team of what is ‘done’ (i.e. a definition of ‘done’).

At the start of each sprint, planning activity takes place, to determine items to include in the sprint. This is a whole-team activity based on reaching a consensus of how long each potential deliverable will take. As sprint follows sprint, so the accuracy of estimation increases. The sprint is a fixed length, so throughout the duration of the development items could be added or removed to ensure that, at the conclusion, there is tested code ‘on the shelf’ and available for implementation as required.

Other activities of the generalised test process we described are undertaken in each sprint. A daily stand-up meeting should result in a short feedback loop, to enable any adjustments to take place and items that prevent progress to be resolved. The whole time for development and testing is limited, so the preparation for testing and the testing itself have to be undertaken in parallel. Towards the end of the sprint, it is possible that most or all of the team are involved in testing, with a focus of delivering all that was agreed at the time of the sprint planning meeting.

Automated testing tools are frequently used in Agile projects. Tests can be very low level, and a tester on such a project can provide very useful input to developers in defining tests to be used to identify specific conditions. The earlier in a sprint that this is done, the more advantages can be gained within that sprint.

The proposed deliverables are sometimes used in conjunction with the definition of 'done' to enable a burn-down chart to be drawn up. This enables a track to be kept of progress for all to see – in itself usually a motivating factor. As sprint follows sprint, the subject of regression testing previously delivered, working software becomes more important. Part of the test analysis and test design will involve selecting regression tests that are appropriate for the current sprint. Tests that previously worked might now (correctly) not pass because the new sprint has changed the intention of previously delivered software. The conclusion of the sprint is often a sprint review meeting, which can include a demonstration to user representatives and/or the project sponsor.

For development using Agile methodologies, the final stage of our test process – 'test completion activities' – is scheduled after the end of the last sprint. This should not be done before, because testing collateral that was used in the last-but-three sprint might no longer be appropriate at the conclusion of the final sprint. As we discussed earlier, even the regression tests might have changed, or a more suitable set of regression tests identified.

Further information about Agile methodologies is given in **Chapter 2**.

Test work products

Each of the activity groups we have discussed has one or more work products that are typically produced as part of that set of activities. However, just as there are major variations in the way particular organisations implement the test process, so there can be variations in both the work products that are produced, and in some cases even the names of those work products. This section follows the test process that we have outlined above, and work products are given for each of the seven activity groups that we described in a generalised test process.

Many of the work products that are described here can be captured and managed using test management and defect management tools (these two tool types are both described in more detail in **Chapter 6**). Work products used in test process activity groups can often be easily attributed to the appropriate activity group or when considering the preceding or following activity group, so even though 'Test work products' can be a topic in the examination, this does not mean that you have to learn the lists that are given! An example of an examination-type question relating to this area of study is given at the end of this chapter.

There are some work products that are typically created in two or more test process activity groups. Examples of this are defect reports (most usually in test analysis, test design, test implementation and test execution) and test reports (test progress reports and test summary reports in test monitoring and control, test completion reports in both test monitoring and control, and test completion).

Test planning work products

Test planning activities usually produce plans and schedules. There can be several test plans for a project: component testing test plan, integration testing test plan and so on. Test plans contain information about the test basis and the exit criteria (or definition of 'done'), so that we know in advance when we can say that testing is complete. Other work products will be related to the test basis by traceability information.

Test monitoring and control work products

Work products produced from the test monitoring and control activities include various types of test reports. Some of these are periodic (a test progress report every three weeks, perhaps), while others are at specific completed milestones for the project (test summary reports). Not all reports are for the same target audience, so any reports need to be audience specific in the level of detail. Reports for senior stakeholders may just give the number of tests that have passed, have failed and cannot yet be run, for example, whereas reports for the wider development team may include sub-system-specific information about the tests that have passed and failed.

Test monitoring and control work products also need to highlight project management concerns including task completion, the use of resources and the amount of effort that has been expended. Where necessary, there may be choices highlighted with possible actions that can be taken to still achieve the original timescales, even though at the present time we are behind schedule.

Test analysis work products

The following are work products that the test analysis group of activities is expected to produce:

- Defined and prioritised test conditions (ideally each with bi-directional traceability to the specific part(s) of the test basis that is covered).
- For exploratory testing, test charters may be created.
- Test analysis may result in the discovery and reporting of defects in the test basis.

Test design work products

The key work products that come from test design activities are test cases and groups of test cases that relate to the test conditions created by test analysis. It is extremely useful if these are bi-directionally traceable to the test conditions they relate to. Test cases can be high-level test cases (without specific values, or concrete values to be used for input values) or low-level test cases (with detailed input values, and **specific expected results**). High-level test cases can be reused in different test cycles, although they are not as easy to use by individuals who are not familiar with the software being tested.

Other work products that are produced or amended as part of test design include amendments to the test conditions created in test analysis, the design and/or identification of any test data that is

needed, the definition and design of the test environment, and the identification of any infrastructure or tools that may be required. However, although these last mentioned are done, the amount that is documented can vary.

Test implementation work products

Test implementation activities have some work products that are more easily identifiable than others. The first three below are very recognisable, but the others can be also created at this stage:

- Test procedures, and the sequencing of these.
- Test suites, being comprised of two or more test procedures.
- A test execution schedule.
- In some cases, test implementation activities create work products using or used by tools:
  - service virtualisation;
  - automated test scripts.
- The creation and verification of test data.
- Creation and verification of the test environment.
- Further refinement of the test conditions that were produced as part of test analysis.

When we have specific test data, this can be used to turn high-level test cases into low-level test cases, and to these are added expected results. In some instances, expected results can be derived from the test data using a test oracle – you feed the data in and get the expected results as the answer.

Once test implementation is complete, it can be possible to see the level of coverage by written test cases for parts or all of the test basis. This is because at each stage, we have built-in, bi-directional traceability. It may be possible to see how many test procedures have been written that cover a specific requirement, or area of functionality.

Test execution work products

The three key work products created by test execution activities are given below:

- The status of individual tests (passed, failed, skipped, ready to run, blocked, etc.).
- Defect reports as a result of test execution.
- Details of what was involved in the testing (test item(s), test objects, testware and test tools). This includes the version identifier of each of these items – so that if necessary, the exact test can be replicated at a future time.

When testing is complete, if there is bi-directional traceability between the test basis, test conditions, test cases, test procedures and test suites, it is possible to work backwards and say

which requirements have failed tests against them, where the impact of defects touches the business areas and so on. This will enable checking that the pre-determined coverage level has been met, or not, and help reporting in ways that business stakeholders understand.

Test completion work products

When testing is complete, the following can be created from the test completion activities:

- test summary reports, which could be a ‘test completion report’;
- a list of improvements for future work (the next iteration, or next project);
- change requests or product backlog items;
- finalised testware (and sometimes, the test environment) for future usage.

Traceability between the test basis and test work products

Throughout the descriptions of the test process groups of activities, and the work products that are created by these activity groups, there has been a sub-story of bi-directional traceability. Primarily, this enables the evaluation of test coverage: how does this one failed test reflect back to requirements or business goals? There are several great advantages to having full traceability:

- assessing the impact of changes (if one requirement changes, how many test conditions, test cases, test procedures and test suites may be affected);
- making testing auditable;
- enabling IT governance criteria to be met;
- improving the understandability of test progress reports and test summary reports to reflect passed, failed and blocked tests back to requirements or other aspects of the test basis;
- relating testing to stakeholders in terms that they can understand (it is not ‘three failed tests’ but ‘three tests failed, which means that creating a customer is not possible’);
- enabling the assessment of product quality, process capability and project progress against the goals of the business.

CHECK OF UNDERSTANDING

1. What are the activity groups in the generalised test process described (in the correct sequence)?
2. Give advantages of maintaining traceability between the test basis and test work products (including test cases).
3. When should the expected outcome of a test be defined?
4. Give three work products that are created during the test execution group of activities.

THE PSYCHOLOGY OF TESTING

A variety of different people may be involved in the total testing effort, and they may be drawn from a broad set of backgrounds. Some will be developers, some professional testers and some will be specialists, such as those with performance testing skills, while others may be users drafted in to assist with acceptance testing. Whoever is involved in testing needs at least some understanding of the skills and techniques of testing to make an effective contribution to the overall testing effort.

Testing can be more effective if it is not undertaken by the individual(s) who wrote the code, for the simple reason that the creator of anything (whether it is software or a work of art) has a special relationship with the created object. The nature of that relationship is such that flaws in the created object are rendered invisible to the creator. For that reason, it is important that someone other than the creator should test the object. Of course, we do want the developer who builds a component or system to debug it, and even to attempt to test it, but we accept that testing done by that individual cannot be assumed to be complete. Developers can test their own code, but it requires a mindset change, from that of a developer (to prove it works) to that of a tester (trying to show that it does not work).

Testers and developers think in different ways. However, although we know that testers should be involved from the beginning, it is not always good to get testers involved in code execution at an early stage; there are advantages and disadvantages. Getting developers to test their own code has advantages (as soon as problems are discovered, they can be fixed, without the need for extensive error logs), but also difficulties (it is hard to find your own mistakes – the so-called ‘confirmation bias’). People and projects have objectives, and we all modify actions to blend in with the goals. If a developer has a goal of producing acceptable software by certain dates, then any testing is aimed towards that goal.

If a defect is found in software, the software author may see this as criticism. Testers need to use tact and diplomacy when raising defect reports. Defect reports need to be raised against the software, not against the individual who made the mistake. The mistake may be in the code written, or in one of the documents on which the code is based (requirement documents or system specification). When we raise defects in a constructive way, bad feeling can be avoided.

We all need to focus on good communication, and work on team building. Testers and developers are not opposed but working together, with the joint target of better-quality systems. Communication needs to be objective, and expressed in impersonal ways:

- The aim is to work together rather than be confrontational. Keep the focus on delivering a quality product.
- Results should be presented in a non-personal way. The work product may be wrong, so say this in a non-personal way.
- Attempt to understand how others feel; it is possible to discuss problems and still leave all parties feeling positive.

- At the end of discussions, confirm that you have both understood and been understood. ‘So, am I right in saying that you will aim to deliver with all the agreed priority fixes on Friday this week by 12.00?’

As testers and developers, one of our goals is better-quality systems delivered in a timely manner. Good communication between testers and the development teams is one way that this goal can be reached.

CHECK OF UNDERSTANDING

1. Describe ways in which testers and developers think differently.
2. Contrast the advantages and disadvantages of developers testing their own code.
3. Suggest three ways that confrontation can be avoided.

CODE OF ETHICS

It should be noted that this section is **not** examinable but is retained (being in earlier versions of the syllabus, but not the current one), as it is a useful topic for the tester and aspiring tester to be aware of.

We will now look at how testers should behave as professionals in the workplace, a code of ethics, before we move onto the more detailed coverage of topics in the following chapters. Testers can have access to confidential and/or privileged information, and they are to treat any information with care and attention, and act responsibly when dealing with the owner(s) of this information, employers and the wider public interest. Of course, anyone can test software, so the declaration of this code of ethics applies to those who have achieved software testing certification. The code of ethics applies to the following areas:

- Public – certified software testers shall consider the wider public interest in their actions.
- Client and employer – certified software testers shall act in the best interests of their client and employer (being consistent with the wider public interest).
- Product – certified software testers shall ensure that the deliverables they provide (for any products and systems they work on) meet the highest professional standards possible.
- Judgement – certified software testers shall maintain integrity and independence in their professional judgement.
- Management – certified software test managers and leaders shall subscribe to and promote an ethical approach to the management of software testing.
- Profession – certified software testers shall advance the integrity and reputation of the profession consistent with the public interest.

- Colleagues – certified software testers shall be fair to, and supportive of, their colleagues and promote cooperation with software developers.
- Self – certified software testers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

The code of ethics is far-reaching in its aims, and a quick review of the eight points reveals interaction with specific areas of the syllabus. The implementation of this code of ethics is expanded on in all chapters of this book, and perhaps is the reason for the whole book itself.

SUMMARY

In this chapter, we have looked at key ideas that are used in testing and introduced some terminology. We examined some of the types of software problems that can occur, and why the blanket explanation of 'insufficient testing' is unhelpful. The problems encountered then led us through some questions about the nature of testing, why errors and mistakes are made, and how these can be identified and eliminated. Individual examples enabled us to look at what testing can achieve, and the view that testing does not improve software quality but provides information about that quality.

We have examined both general testing principles and a standard template for testing: a generalised test process. These are useful and can be effective in identifying the types of problems we considered at the start of the chapter. The chapter finished by examining how developers and testers think, and how testers should behave by adhering to a code of ethics.

This chapter is an introduction to testing, and to themes that are developed later in the book. It is a chapter in its own right, but also points to information that will come later. A rereading of this chapter when you have worked through the rest of the book will place all the main topics into context.

Example examination questions with answers

E1. K1 question
What is a test condition?

a. A set of test data written to exercise one or more logic paths through the software under test.
b. An aspect of the test basis appropriate to achieve specific test objectives.
c. The body of knowledge used as the foundation for test analysis and design.
d. The pre-determined goals that will enable a decision to be made about whether testing is complete.

E2. K2 question
Which of the following definition pairs for testing and debugging is correct?

a. Debugging can show failures in the software; testing is investigating the causes of any failures and performing corrections.
b. Debugging is investigating the causes of software failures and undertaking corrective action; testing attempts to uncover problems by executing the software.
c. Debugging is always undertaken by developers; testing can be performed by development or testing personnel.
d. Debugging checks that software fixes have been resolved; testing is looking for any unintended consequences of a software fix.

E3. K1 question
Which of the following are aids to good communication within the development team?

i. Try to understand how the other person feels.
ii. Communicate personal feelings, concentrating on individuals.
iii. Confirm the other person has understood what you have said and vice versa.
iv. Emphasise the common goal of better quality.
v. Each discussion is a battle to be won.

a. i, ii and iii aid good communication.
b. iii, iv and v aid good communication.
c. i, iii and iv aid good communication.
d. ii, iii and iv aid good communication.

E4. K2 question
Which of the following illustrates one of the testing principles?

a. No unresolved defects does not mean the software will be successful.
b. The more you test, the more defects will be found.
c. All software can be tested in the same way using the same test techniques.
d. Defects are usually found evenly distributed throughout the software under test.

E5. K2 question
Which of the following activities are part of the test implementation activity group, and which part of test execution?

i. Developing and prioritising test procedures, creating automated test scripts.
ii. Comparing actual and expected results.
iii. Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures and test results.
iv. Preparing test data and ensuring it is properly loaded in the test environment.

v. Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures and test suites.

a. i, ii and iii are part of test implementation, iv and v are part of test execution.
b. i, iii and v are part of test implementation, ii and iv are part of test execution.
c. i, ii and iv are part of test implementation, iii and v are part of test execution.
d. i iv and v are part of test implementation, ii and iii are part of test execution.

E6. K2 question
Which of the following ways of thinking, broadly apply to developers and which to testers?

i. Curiosity, professional pessimism and a critical eye.
ii. Interested in designing and building solutions.
iii. Can be subject to the confirmation bias.
iv. Has good attention to detail.
v. Contemplates what might go wrong.

a. ii and iii: developers; i, iv and v: testers.
b. i and ii: developers; iii, iv and v: testers.
c. iv and v: developers; i, ii and iii: testers.
d. iii and v: developers; i, ii and iv: testers.

E7. K2 question
Which of the following is a recognised reason for testing to be carried out?

a. Using testers in the review of requirements will verify that the software is fit for purpose.
b. Evidence suggests that between 25% and 45% of the project costs should be used in the testing process.
c. The detection and removal of defects increases the likelihood that the software meets stakeholder needs.
d. Testing ensures that there are no residual defects in the software under test.

E8. K2 question
Problems persist with the online customer interface for a utilities company, after the introduction of smart meters in customer homes. Which *one* of the following is a root cause of a defect rather than an effect of a defect?

a. Consumption usage for some days is shown as zero, while that for other days is abnormally high.

b. Some chosen menu options intermittently display the ‘system busy’ icon, but never the correct information at the time it is selected.

c. Customer consumption data is only displayed for up to three days ago, with year-to-date and month-to-date excluding the last three days.

d. Systems architects did not anticipate the amount of web traffic the introduction of smart meters would generate.

E9. K2 question

Which *two* of the following work products are created during the test implementation activity?

i. Documentation about which test item(s), test object(s), test tools and testware were involved in the testing.

ii. Test execution schedule.

iii. Test cases.

iv. Documentation about the **status** (e.g. ‘pass’, ‘fail’, ‘not run’ etc.) of individual test cases or procedures.

v. Test procedures and their sequencing.

a. iii and v.

b. ii and v.

c. i and iii.

d. ii and iv.

Answers to questions in the chapter

SA1. The correct answer is c.
SA2. The correct answer is b.
SA3. The correct answer is d.

Answers to example examination questions

E1. The correct answer is b.

a. is the definition of a test case.

c. is the description of a test basis.

d. is a broad description of test exit criteria.

E2. The correct answer is b.

a. the two parts are ‘switched’; what is described as ‘debugging’ is in fact ‘testing’, and vice versa.

c. is not true. The syllabus states that in some life cycles, testers may be involved in debugging.

d. indicates that debugging is involved in retesting software after fixes. This is a description of 'retesting', a testing activity. The description given to 'testing' is that of regression testing; testing involves more than this.

E3. The correct answer is c.
Choices ii and v are factors that will **not** help good communication, but will cause mistrust, antagonism and stress in the team. These are ruled out. Option c is the answer that has the other choices. A quick check sees that these are all matters that will encourage openness and trust – and are therefore correct choices.

E4. The correct answer is a.

b. this option has **some** truth to it, but it is not one of the testing principles. However, if there are no defects in the code, no amount of testing will find defects.

c. is not true. It is in direct contradiction to the 'testing is context dependent' principle.

d. is again not true. This is usually found **not** to be the case, being the opposite of the 'defect clustering' principle.

E5. The correct answer is d.
Two of the choices are very similar, choices iii and v. The differences here is **test results** (choice iii) and **test suites** (choice v). This points to choice iii being part of test execution and choice v being part of test implementation. Option d is the only one that has these assigned in this way.

E6. The correct answer is a.
This question is not implying that all developers think in one way and all testers in another, nor that individual developers cannot exhibit 'think as testers' characteristics. The choices provided are fairly straightforward, with the exception of iv (has good attention to detail). This last choice is given as an attribute of the way a tester thinks, but is not exclusive to testers! Choices i and iii are more aligned to testers, whereas choices ii and v relate to developers. This points to option a being the correct answer.

E7. The correct answer is c.
Reviewing **requirements** can never verify that **software** is fit for purpose (option a). Just because evidence **may** suggest that a proportion of project cost should be spent on testing is not a reason to perform testing. This rules out option b. Option d states that testing can find all defects, which means that we can show that there are no remaining defects – contrary to one of the testing principles about testing only finding defects; it cannot show that are **no defects**. This leaves option c, which is the correct answer.

E8. The correct answer is d.
A root cause is **why** something has happened, as opposed to what has happened. Options a, b and c describe unusual events (which may or may not be defects but are certainly irritating for

customers). Option b could be a system overload problem – this could be as a result of a higher than expected amount of web traffic. This root cause is described in option d, the correct answer.

E9. The correct answer is b.
We will consider each of the work products in turn:

i. Documentation about which test item(s), test object(s), test tools and testware were involved in the testing – **test execution**.
ii. Test execution schedule – **test implementation**.
iii. Test cases – **test execution**.
iv. Documentation about the **status** (e.g. ‘pass’, ‘fail’, ‘not run’ etc.) of individual test cases or procedures – **test implementation**.
v. Test procedures and their sequencing – **test execution**.

Option b gives the correct choices.

---

1 Kit, E. (1995) Software Testing in the Real World. Reading, MA: Addison-Wesley.