CHAPTER 1
Fundamentals of testing

In this chapter, we will introduce you to the fundamentals of testing: why testing is needed; its limitations, objectives and purpose; the principles behind testing; the process that testers follow; and some of the psychological factors that testers must consider in their work. By reading this chapter you'll gain an understanding of the fundamentals of testing and be able to describe those fundamentals.

1.1 WHY IS TESTING NECESSARY?

1 **Describe, with examples, the way in which a defect in software can cause harm to a person, to the environment or to a company. (K2)**

2 **Distinguish between the root cause of a defect and its effects. (K2)**

3 **Give reasons why testing is necessary by giving examples. (K2)**

4 **Describe why testing is part of quality assurance and give examples of how testing contributes to higher quality. (K2)**

5 **Recall the terms 'mistake', 'defect', 'fault', 'failure' and the correspon ding terms 'error' and 'bug'. (Kl)**

6 **Explain the fundamental principles in testing. (K2)**

1.1.1 Introduction

In this section, we're going to kick off the book with a discussion on why testing matters. We'll describe and illustrate how software defects or bugs can cause problems for people, the environment or a company. We'll draw important distinctions between defects, their root causes and their effects. We'll explain why testing is necessary to find these defects, how testing promotes quality, and how testing fits into quality assurance. In this section, we will also introduce some fundamental principles of testing.

As we go through this section, watch for the Syllabus terms **bug, defect, error, failure, fault, mistake, quality, risk, software, testing** and **exhaustive testing.** You'll find these terms defined in the glossary.

You may be asking 'what is testing?' and we'll look more closely at the definition of testing in Section 1.2. For the moment, let's adopt a simple everyday-life usage: 'when we are testing something we are checking whether it is OK'. We'll need to refine that when we define software testing later on. Let's start by considering why testing is needed. Testing is necessary because we all make mistakes. Some of those mistakes are unimportant, but some of them are expensive or dangerous. We need to check everything and anything we produce because things can always go wrong - humans make mistakes all the time - it is what we do best!

Because we should assume our work contains mistakes, we all need to check our own work. However, some mistakes come from bad assumptions and blind spots, so we might make the same mistakes when we check our own work as we made when we did it. So we may not notice the flaws in what we have done. Ideally, we should get someone else to check our work - another person is more likely to spot the flaws.

In this book, we'll explore the implications of these two simple paragraphs again and again. Does it matter if there are mistakes in what we do? Does it matter if we don't find some of those flaws? We know that in ordinary life, some of our mistakes do not matter, and some are very important. It is the same with software systems. We need to know whether a particular error is likely to cause problems. To help us think about this, we need to consider the context within which we use different software systems.

1.1.2 Software systems context

Testing Principle - Testing is context dependent

Testing is done differently in different contexts. For example, safety-critical software is tested differently from an e-commerce site.

These days, almost everyone is aware of **software** systems. We encounter them in our homes, at work, while shopping, and because of mass-communication systems. More and more, they are part of our lives. We use software in day-to-day business applications such as banking and in consumer products such as cars and washing machines. However, most people have had an experience with software that did not work as expected: an error on a bill, a delay when waiting for a credit card to process and a website that did not load correctly are common examples of problems that may happen because of software problems. Not all software systems carry the same level of **risk** and not all problems have the same impact when they occur. A risk is something that has not happened yet and it may never happen; it is a potential problem. We are concerned about these potential problems because, if one of them did happen, we'd feel a negative impact. When we discuss risks, we need to consider how likely it is that the problem would occur and the impact if it happens. For example, whenever we cross the road, there is some risk that we'll be injured by a car. The likelihood depends on factors such as how much traffic is on the road, whether there is a safe crossing place, how well we can see, and how fast we can cross. The impact depends on how fast the car is going, whether we are wearing protective gear, our age and our health. The risk for a particular person can be worked out and therefore the best road-crossing strategy.

Some of the problems we encounter when using software are quite trivial, but others can be costly and damaging - with loss of money, time or business reputation - and even may result in injury or death. For example, suppose a user interface has typographical defects. Does this matter? It may be trivial, but it could have a significant effect, depending on the website and the defect:

- If my personal family-tree website has my maternal grandmother's maiden name spelt wrong, my mother gets annoyed and I have to put up with some family teasing, but I can fix it easily and only the family see it (probably).
- If the company website has some spelling mistakes in the text, potential cus tomers may be put off the company as it looks unprofessional.
- If a software program miscalculates pesticide application quantities, the effect could be very significant: suppose a decimal point is wrongly placed so that the application rate is 10 times too large. The farmer or gardener uses more pesticide than needed, which raises his costs, has environmental impacts on wildlife and water supplies and has health and safety impact for the farmer, gardener, family and workforce, livestock and pets. There may also be consequent loss of trust in and business for the company and possi ble legal costs and fines for causing the environmental and health problems.

1.1.3 Causes of software defects

Why is it that software systems sometimes don't work correctly? We know that people make mistakes - we are fallible.

If someone makes an **error** or mistake in using the software, this may lead directly to a problem - the software is used incorrectly and so does not behave as we expected. However, people also design and build the software and they can make mistakes during the design and build. These mistakes mean that there are flaws in the software itself. These are called **defects** or sometimes bugs or faults. Remember, the software is not just the code; check the definition of software again to remind yourself.

When the software code has been built, it is executed and then any defects may cause the system to fail to do what it should do (or do something it shouldn't), causing a **failure.** Not all defects result in failures; some stay dormant in the code and we may never notice them.

Do our mistakes matter?

Let's think about the consequences of mistakes. We agree that any human being, programmers and testers included, can make an error. These errors may produce defects in the software code or system, or in a document. If a defect in code is executed, the system may experience a failure. So the mistakes we make matter partly because they have consequences for the products for which we are responsible.

Our mistakes are also important because software systems and projects are complicated. Many interim and final products are built during a project, and people will almost certainly make mistakes and errors in all the activities of the build. Some of these are found and removed by the authors of the work, but it is difficult for people to find their own mistakes while building a product. Defects in software, systems or documents may result in failures, but not all

defects do cause failures. We could argue that if a mistake does not lead to a defect or a defect does not lead to a failure, then it is not of any importance - we may not even know we've made an error.

Our fallibility is compounded when we lack experience, don't have the right information, misunderstand, or if we are careless, tired or under time pressure. All these factors affect our ability to make sensible decisions - our brains either don't have the information or cannot process it quickly enough.

Additionally, we are more likely to make errors when dealing with perplexing technical or business problems, complex business processes, code or infrastructure, changing technologies, or many system interactions. This is because our brains can only deal with a reasonable amount of complexity or change - when asked to deal with more our brains may not process the information we have correctly.

It is not just defects that give rise to failure. Failures can also be caused by environmental conditions as well: for example, a radiation burst, a strong magnetic field, electronic fields, or pollution could cause faults in hardware or firmware. Those faults might prevent or change the execution of software. Failures may also arise because of human error in interacting with the software, perhaps a wrong input value being entered or an output being misinterpreted. Finally, failures may also be caused by someone deliberately trying to cause a failure in a system - malicious damage.

When we think about what might go wrong we have to consider defects and failures arising from:

- errors in the specification, design and implementation of the software and system;
- errors in use of the system;
- environmental conditions;
- intentional damage;
- potential consequences of earlier errors, intentional damage, defects and failures.

When do defects arise?

In Figure 1.1 we can see how defects may arise in four requirements for a product.

We can see that requirement 1 is implemented correctly - we understood the customer's requirement, designed correctly to meet that requirement, built correctly to meet the design, and so deliver that requirement with the right attributes: functionally, it does what it is supposed to do and it also has the right non-functional attributes, so it is fast enough, easy to understand and so on.

With the other requirements, errors have been made at different stages. Requirement 2 is fine until the software is coded, when we make some mistakes and introduce defects. Probably, these are easily spotted and corrected during testing, because we can see the product does not meet its design specification.

The defects introduced in requirement 3 are harder to deal with; we built exactly what we were told to but unfortunately the designer made some mistakes so there are defects in the design. Unless we check against the requirements definition, we will not spot those defects during testing. When we do notice them they will be hard to fix because design changes will be required.

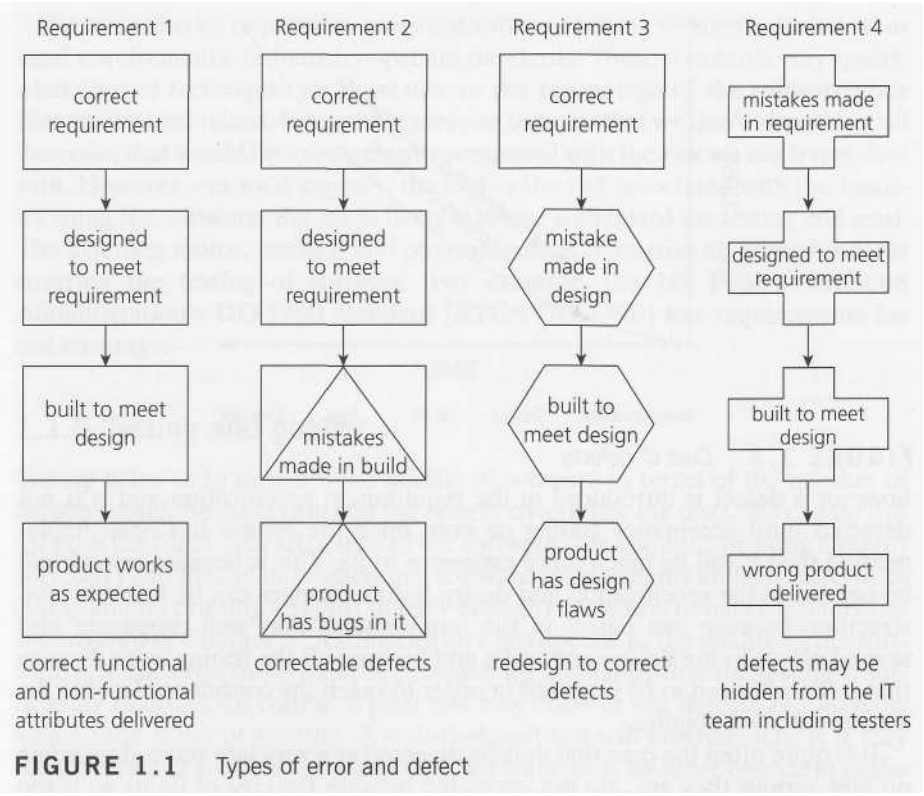

FIGURE 1.1 Types of error and defect

The defects in requirement 4 were introduced during the definition of the requirements; the product has been designed and built to meet that flawed requirements definition. If we test the product meets its requirements and design, it will pass its tests but may be rejected by the user or customer. Defects reported by the customer in acceptance test or live use can be very costly. Unfortunately, requirements and design defects are not rare; assessments of thousands of projects have shown that defects introduced during requirements and design make up close to half of the total number of defects [Jones].

What is the cost of defects?

As well as considering the impact of failures arising from defects we have not found, we need to consider the impact of when we find those defects. The cost of finding and fixing defects rises considerably across the life cycle; think of the old English proverb 'a stitch in time saves nine'. This means that if you mend a tear in your sleeve now while it is small, it's easy to mend, but if you leave it, it will get worse and need more stitches to mend it.

If we relate the scenarios mentioned previously to Figure 1.2, we see that, if an error is made and the consequent defect is detected in the requirements at the specification stage, then it is relatively cheap to find and fix. The observation of increasing defect-removal costs in software traces back to [Boehm]. You'll find evidence for the economics of testing and other quality assurance activities in [Gilb], [Black 2001] or [Black 2004]. The specification can be corrected and re-issued. Similarly if an error is made and the consequent defect detected in the design at the design stage then the design can be corrected and re-issued with relatively little expense. The same applies for construction. If

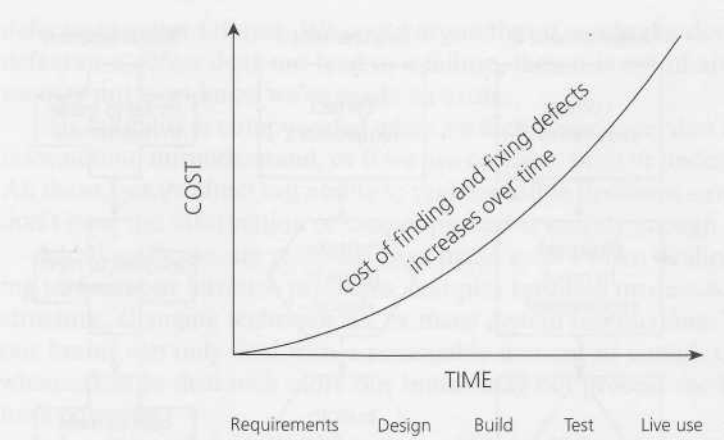

FIGURE 1.2 Cost of defects

however a defect is introduced in the requirement specification and it is not detected until acceptance testing or even once the system has been implemented then it will be much more expensive to fix. This is because rework will be needed in the specification and design before changes can be made in construction; because one defect in the requirements may well propagate into several places in the design and code; and because all the testing work done-to that point will need to be repeated in order to reach the confidence level in the software that we require.

It is quite often the case that defects detected at a very late stage, depending on how serious they are, are not corrected because the cost of doing so is too expensive. Also, if the software is delivered and meets an agreed specification, it sometimes still won't be accepted if the specification was wrong. The project team may have delivered exactly what they were asked to deliver, but it is not what the users wanted. This can lead to users being unhappy with the system that is finally delivered. In some cases, where the defect is too serious, the system may have to be de-installed completely.

1.1.4 Role of testing in software development, maintenance and operations

We have seen that human errors can cause a defect or fault to be introduced at any stage within the software development life cycle and, depending upon the consequences of the mistake, the results can be trivial or catastrophic. Rigorous testing is necessary during development and maintenance to identify defects, in order to reduce failures in the operational environment and increase the quality of the operational system. This includes looking for places in the user interface where a user might make a mistake in input of data or in the interpretation of the output, and looking for potential weak points for intentional and malicious attack. Executing tests helps us move towards improved quality of product and service, but that is just one of the verification and validation methods applied to products. Processes are also checked, for example by audit. A variety of methods may be used to check work, some of which are done by the author of the work and some by others to get an independent view.

We may also be required to carry out software testing to meet contractual or legal requirements, or industry-specific standards. These standards may specify what type of techniques we must use, or the percentage of the software code that must be exercised. It may be a surprise to learn that we don't always test all the code; that would be too expensive compared with the risk we are trying deal with. However - as we'd expect - the higher the risk associated with the industry using the software, the more likely it is that a standard for testing will exist. The avionics, motor, medical and pharmaceutical industries all have standards covering the testing of software. For example, the US Federal Aviation Administration's DO-178B standard [RTCA/DO-178B] has requirements for test coverage.

1.1.5 Testing and quality

Testing helps us to measure the **quality** of software in terms of the number of defects found, the tests run, and the system covered by the tests. We can do this for both the functional attributes of the software (for example, printing a report correctly) and for the non-functional software requirements and characteristics (for example, printing a report quickly enough). Non-functional characteristics are covered in Chapter 2. Testing can give confidence in the quality of the software if it finds few or no defects, provided we are happy that the testing is sufficiently rigorous. Of course, a poor test may uncover few defects and leave us with a false sense of security. A well-designed test will uncover defects if they are present and so, if such a test passes, we will rightly be more confident in the software and be able to assert that the overall level of risk of using the system has been reduced. When testing does find defects, the quality of the software system increases when those defects are fixed, provided the fixes are carried out properly.

What is quality?

Projects aim to deliver software to specification. For the project to deliver what the customer needs requires a correct specification. Additionally, the delivered system must meet the specification. This is known as validation ('is this the right specification?') and verification ('is the system correct to specification?'). Of course, as well as wanting the right software system built correctly, the customer wants the project to be within budget and timescale - it should arrive when they need it and not cost too much.

The ISTQB glossary definition covers not just the specified requirements but also user and customer needs and expectations. It is important that the project team, the customers and any other project stakeholders set and agree expectations. We need to understand what the customers understand by quality and what their expectations are. What we as software developers and testers may see as quality - that the software meets its defined specification, is technically excellent and has few bugs in it - may not provide a quality solution for our customers. Furthermore, if our customers find they have spent more money than they wanted or that the software doesn't help them carry out their tasks, they won't be impressed by the technical excellence of the solution. If the customer wants a cheap car for a 'run-about' and has a small budget then an expensive

sports car or a military tank are not quality solutions, however well built they are.

To help you compare different people's expectations, Table 1.1 summarizes and explains quality viewpoints and expectations using 'producing and buying tomatoes' as an analogy for 'producing and buying software'. You'll see as you look through the table that the approach to testing would be quite different depending on which viewpoint we favor [Trienekens], [Evans].

In addition to understanding what quality feels and looks like to customers, users, and other stakeholders, it helps to have some quality attributes to measure quality against, particularly to aid the first, product-based, viewpoint in the table. These attributes or characteristics can serve as a framework or checklists for areas to consider coverage. One such set of quality attributes can

TABLE 1.1 Viewpoints of expectations and quality

Viewpoint	Software	Tomatoes
Quality is measured by looking at the attributes of the product.	We will measure the attributes of the software, e.g. its reliability in terms of mean time between failures (MBTF), and release when they reach a specified level e.g. MTBF of 12 hours.	The tomatoes are the right size and shape for packing for the supermarket. The tomatoes have a good taste and color,
Quality is fitness for use. Quality can have subjective aspects and not just quantitative aspects.	We will ask the users whether they can carry out their tasks; if they are satisfied that they can we will release the software.	The tomatoes are right for our recipe,
Quality is based on good manufacturing processes, and meeting defined requirements. It is measured by testing, inspection, and analysis of faults and failures.	We will use a recognized software development process. We will only release the software if there are fewer than five outstanding high-priority defects once the planned tests are complete.	The tomatoes are organically farmed. The tomatoes have no blemishes and no pest damage,
Expectation of value for money. affordability, and a value-based trade-off between time, effort and cost aspects. We can afford to buy this software and we expect a return on investment.	We have time-boxed the testing to two weeks to stay in the project budget.	The tomatoes have a good shelf life. The tomatoes are cheap or good value for money,
Transcendent feelings - this is about the feelings of an individual or group of individuals towards a product or a supplier.	We like this software! It is fun and it's the latest thing! So what if it has a few problems? We want to use it anyway... We really enjoy working with this software team. So, there were a few problems - they sorted them out really quickly - we trust them.	We get our tomatoes from a small local farm and we get on so well with the growers,

be found in the ISO 9126 standard. This hierarchy of characteristics and sub-characteristics of quality is discussed in Chapter 2.

What is root cause analysis?

When we detect failures, we might try to track them back to their root cause, the real reason that they happened. There are several ways of carrying out root cause analysis, often involving a group brainstorming ideas and discussing them, so you may see different techniques in different organizations. If you are interested in using root cause analysis in your work, you'll find simple techniques described in [Evans], [TQMI] and [Robson]. For example, suppose an organization has a problem with printing repeatedly failing. Some IT maintenance folk get together to examine the problem and they start by brainstorming all the possible causes of the failures. Then they group them into categories they have chosen, and see if there are common underlying or root causes. Some of the obvious causes they discover might be:

- Printer runs out of supplies (ink or paper).
- Printer driver software fails.
- Printer room is too hot for the printer and it seizes up.

These are the immediate causes. If we look at one of them - 'Printer runs out of supplies (ink or paper)' - it may happen because:

- No-one is responsible for checking the paper and ink levels in the printer; possible root cause: no process for checking printer ink/paper levels before use.
- Some staff don't know how to change the ink cartridges; possible root cause: staff not trained or given instructions in looking after the printers.
- There is no supply of replacement cartridges or paper; possible root cause: no process for stock control and ordering.

If your testing is confined to software, you might look at these and say, 'These are not software problems, so they don't concern us!' So, as software testers we might confine ourselves to reporting the printer driver failure. However, our remit as testers may be beyond the software; we might have a remit to look at a whole system including hardware and firmware. Additionally, even if our remit is software, we might want to consider how software might help people prevent or resolve problems; we may look beyond this view. The software could provide a user interface which helps the user anticipate when paper or ink is getting low. It could provide simple step-by-step instructions to help the users change the cartridges or replenish paper. It could provide a high temperature warning so that the environment can be managed. As testers, we want not just to think and report on defects but, with the rest of the project team, think about any potential causes of failures.

We use testing to help us find faults and (potential) failures during software development, maintenance and operations. We do this to help reduce the risk of failures occurring in an operational environment - in other words once the system is being used - and to contribute to the quality of the software system. However, whilst we need to think about and report on a wide variety of defects and failures, not all get fixed. Programmers and others may correct defects

before we release the system for operational use, but it may be more sensible to work around the failure. Fixing a defect has some chance of introducing another defect or of being done incorrectly or incompletely. This is especially true if we are fixing a defect under pressure. For this reason, projects will take a view sometimes that they will defer fixing a fault. This does not mean that the tester who has found the problems has wasted time. It is useful to know that there is a problem and we can help the system users work around and avoid it.

The more rigorous our testing, the more defects we'll find. But you'll see in Chapters 3 and 4, when we look at techniques for testing, that rigorous testing does not necessarily mean more testing; what we want to do is testing that finds defects - a small number of well-placed, targeted tests may be more rigorous than a large number of badly focused tests.

We saw earlier that one strategy for dealing with errors, faults and failures is to try to prevent them, and we looked at identifying the causes of defects and failures. When we start a new project, it is worth learning from the problems encountered in previous projects or in the production software. Understanding the root causes of defects is an important aspect of quality assurance activities, and testing contributes by helping us to identify defects as early as possible before the software is in use. As testers, we are also interested in looking at defects found in other projects, so that we can improve our processes. Process improvements should prevent those defects recurring and, as a consequence, improve the quality of future systems. Organizations should consider testing as part of a larger quality assurance strategy, which includes other activities (e.g., development standards, training and root cause analysis).

1.1.6 How much testing is enough?

Testing Principle - Exhaustive testing is impossible

Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Instead of exhaustive testing, we use risks and priorities to focus testing efforts.

We've seen that testing helps us find defects and improve software quality. How much testing should we do? We have a choice: test everything, test nothing or test some of the software. Now, your immediate response to that may well be to say, 'Everything must be tested'. We don't want to use software that has not been completely tested, do we? This implies that we must exercise every aspect of a software system during testing. What we need to consider is whether we must, or even can, test completely.

Let's look at how much testing we'd need to do to be able to test exhaustively. How many tests would you need to do to completely test a one-digit numeric field? The immediate question is, 'What you mean by test completely?' There are 10 possible valid numeric values but as well as the valid values we need to ensure that all the invalid values are rejected. There are 26 uppercase alpha characters, 26 lower case, at least 6 special and punctuation characters as well as a blank value. So there would be at least 68 tests for this example of a one-digit field.

This problem just gets worse as we look at more realistic examples. In practice, systems have more than one input field with the fields being of varying sizes. These tests would be alongside others such as running the tests in differ-

ent environments. If we take an example where one screen has 15 input fields, each having 5 possible values, then to test all of the valid input value combinations you would need 30 517 578 125 (5^{15}) tests! It is unlikely that the project timescales would allow for this number of tests.

Testing our one-digit field with values 2, 3 and 4 makes our tests more thorough, but it does not give us more information than if we had just tested with the value 3.

Pressures on a project include time and budget as well as pressure to deliver a technical solution that meets the customers' needs. Customers and project managers will want to spend an amount on testing that provides a return on investment for them. This return on investment includes preventing failures after release that are costly. Testing completely - even if that is what customers and project managers ask for - is simply not what they can afford.

Instead we need a test approach which provides the right amount of testing for this project, these customers (and other stakeholders) and this software. We do this by aligning the testing we do with the risks for the customers, the stakeholders, the project and the software. Assessing and managing risk is one of the most important activities in any project, and is a key activity and reason for testing. Deciding how much testing is enough should take account of the level of risk, including technical and business risks related to the product and project constraints such as time and budget.

We carry out a risk assessment to decide how much testing to do. We can then vary the testing effort based on the level of risk in different areas. Additionally, testing should provide sufficient information to stakeholders to make informed decisions about the release of the software or system we're testing, for the next development step or handover to customers. The effort put into the quality assurance and testing activities needs to be tai lored to the risks and costs associated with the project. Because of the limits in the budget, the time, and in testing we need to decide how we will focus our testing, based on the risks. We'll look at risk assessment in Chapter 5.

1.2 WHAT IS TESTING?

Syllabus learning objectives for 1.2 What is testing?

1 **Recall the common objectives of testing. (Kl)**
2 **Describe the purpose of testing in software development, maintenance and operations as a means to find defects, provide confidence and infor mation, and prevent defects. (K2)**

In this section, we will review the common objectives of testing. We'll explain how testing helps us to find defects, provide confidence and information, and prevent defects. We will also introduce additional fundamental principles of testing.

As you read this section, you'll encounter the terms **code, debugging, development of software, requirement, review, test basis, test case, testing** and **test objective.**

1.2.1 The driving test - an analogy for software testing

We have spent some time describing why we need to test, but we have not discussed what testing is. What do we mean by the word testing? We use the words test and testing in everyday life and earlier we said testing could be described as 'checking the software is OK'. That is not a detailed enough definition to help us understand software testing. Let's use an analogy to help us: driving tests. In a driving test, the examiner critically assesses the candidate's driving, noting every mistake, large or small, made by the driver under test. The examiner takes the driver through a route which tests many possible driving activities, such as road junctions of different types, control and maneuvering of the car, ability to stop safely in an emergency, and awareness of the road, other road users and hazards. Some of the activities *must* be tested. For example, in the UK, an emergency stop test is always carried out, with the examiner simulating the moment of emergency by hitting the dashboard at which point the driver must stop the car quickly, safely and without loss of control. At the end of the test, the examiner makes a judgment about the driver's performance. Has the driver passed the test or failed? The examiner bases the judgment on the number and severity of the failures identified, and also whether the driver has been able to meet the driving requirements. A single severe fault is enough to fail the whole test, but a small number of minor faults might still mean the test is passed. Many minor faults would reduce the confidence of the examiner in the quality —of the driving to the point where the driver cannot pass. The format of the driving test and the conduct of the examiner are worth considering:

- The test is planned and prepared for. In advance of the test, the examiner has planned a series of routes which cover the key driving activities to allow a thorough assessment of the driver's performance. The drivers under test do not know what route they will be asked to take in advance, although they know the requirements of the test.
- The test has known goals - assessing whether the driver is sufficiently safe to be allowed to drive by themselves without an instructor, without endanger ing themselves or others. There are clear pass/fail criteria, based on the number and severity of faults, but the confidence of the examiner in the driving is also taken into account.
- The test is therefore carried out to show that the driver satisfies the require ments for driving and to demonstrate that they are fit to drive. The examiner looks for faults in the driving. The time for the test is limited, so it is not a complete test of the driver's abilities, but it is representative and allows the examiner to make a risk-based decision about the driver. All the drivers are tested in an equivalent way and the examiner is neutral and objective. The examiner will log factual observations which enable a risk assessment to be made about the driving. Based on this, a driver who passes will be given a form enabling him to apply for a full driving license. A driver who fails will get a report with a list of faults and areas to improve before retaking the test.

- As well as observing the driver actually driving, the examiner will ask questions or the driver will take a written exam to check their under standing of the rules of the road, road signs, and what to do in various traffic situations.

1.2.2 Defining software testing

With that analogy in mind, let's look at the ISTQB definition of software **testing.**

Let's break the definition down into parts; the definition has some key phrases to remember. The definition starts with a description of testing as a process and then lists some objectives of the test process. First, let's look at testing as a process:

- *Process* - Testing is a process rather than a single activity - there are a series of activities involved.
- *All life cycle activities* - Chapter 2 looks at testing as a process that takes place throughout the **software development** life cycle. We saw earlier that the later in the life cycle we find bugs, the more expensive they are to fix. If we can find and fix requirements defects at the requirements stage, that must make commercial sense. We'll build the right software, correctly and at a lower cost overall. So, the thought process of design ing tests early in the life cycle can help to prevent defects from being introduced into **code.** We sometimes refer to this as 'verifying the **test basis** via the test design'. The test basis includes documents such as the **requirements** and design specifications. You'll see how to do this in Chapter 4.
- *Both static and dynamic* - We'll see in Chapter 3 that as well as tests where the software code is executed to demonstrate the results of running tests (often called dynamic testing) we can also test and find defects without exe cuting code. This is called static testing. This testing includes **reviewing** of documents (including source code) and static analysis. This is a useful and cost effective way of testing.
- *Planning* - Activities take place before and after test execution. We need to manage the testing; for example, we plan what we want to do; we control the test activities; we report on testing progress and the status of the software under test; and we finalize or close testing when a phase completes. Chapter 5 covers these test management activities.
- *Preparation* - We need to choose what testing we'll do, by selecting test con ditions and designing **test cases.** Chapter 4 covers the test design activities.
- *Evaluation* - As well as executing the tests, we must check the results and evaluate the software under test and the completion criteria, which help us decide whether we have finished testing and whether the software product has passed the tests.
- *Software products and related work products* - We don't just test code. We test the requirements and design specifications, and we test related documents such as operation, user and training material. Static and dynamic testing are both needed to cover the range of products we need to test.

The second part of the definition covers the some of the objectives for testing - the reasons why we do it:

- Determine that (software products) satisfy specified requirements - Some of the testing we do is focused on checking products against the specification for the product; for example we review the design to see if it meets require ments, and then we might execute the code to check that it meets the design. If the product meets its specification, we can provide that information to help stakeholders judge the quality of the product and decide whether it is ready for use.
- Demonstrate that (software products) are fit for purpose - This is slightly different to the point above; after all the specified requirements might be wrong or incomplete. 'Fit for purpose' looks at whether the software does enough to help the users to carry out their tasks; we look at whether the soft ware does what the user might reasonably expect. For example, we might look at who might purchase or use the software, and check that it does do what they expect; this might lead us to add a review of the marketing mate rial to our static tests, to check that expectations of the software are properly set. One way of judging the quality of a product is by how fit it is for its purpose.
- Detect defects - We most often think of software testing as a means of detecting faults or defects that in operational use will cause failures. Finding the defects helps us understand the risks associated with putting the software into operational use, and fixing the defects improves the quality of the prod ucts. However, identifying defects has another benefit. With root cause analysis, they also help us improve the development processes and make fewer mistakes in future work.

This is a suitable definition of testing for any level of testing, from compo-nent testing through to acceptance testing, provided that we remember to take the varying objectives of these different levels of testing into account. (In Chapter 2 we'll cover the different test levels, their objectives, and how they fit into the software development life cycle.)

We can clearly see now why the common perception of testing (that it only consists of running tests, i.e. executing the software) is not complete. This is one of the testing activities, but not all of the testing process.

1.2.3 Software test and driving test compared

We can see that the software test is very like a driving test in many ways, although of course it is not a perfect analogy! The driving examiner becomes the software tester. The driver being examined becomes the system or software under test, and you'll see as we go through this book that the same approach broadly holds.

- *Planning and preparation* - Both the examiner and the tester need a plan of action and need to prepare for the test, which is not exhaustive, but is repre sentative and allows risk-based decisions about the outcome.
- *Static and dynamic* - Both dynamic (driving the car or executing the soft ware) and static (questions to the driver or a review of the software) tests are useful.

- *Evaluation* - The examiner and the tester must make an objective evaluation,
  log the test outcome and report factual observations about the test.
- *Determine that they satisfy specified requirements* - The examiner and tester
  both check against requirements to carry out particular tasks successfully.
- *Demonstrate that they are fit for purpose* - The examiner and the tester are not
  evaluating for perfection but for meeting sufficient of the attributes required
  to pass the test.
- *Detect defects* - The examiner and tester both look for and log faults.

Let's think a little more about planning. Because time is limited, in order to make a representative route that would provide a sufficiently good test, both software testers and driving examiners decide in advance on the route they will take. It is not possible to carry out the driving test and make decisions about where to ask the driver to go next on the spur of moment. If the examiner did that, they might run out of time and have to return to the test center without having observed all the necessary maneuvers. The driver will still want a pass/fail report. In the same way, if we embark on testing a software system without a plan of action, we are very likely to run out of time before we know whether we have done enough testing. We'll see that good testers always have a plan of action. In some cases, we use a lightweight outline providing the goals and general direction of the test, allowing the testers to vary the test during execution. In other cases, we use detailed scripts showing the steps in the test route and documenting exactly what the tester should expect to happen as each step. Whichever approach the tester takes, there will be some plan of action. Similarly, just as the driving examiner makes a log and report, a good tester will objectively document defects found and the outcome of the test.

So, test activities exist before and after test execution, and we explain those activities in this book. As a tester or test manager, you will be involved in planning and control of the testing, choosing test conditions, designing test cases based on those test conditions, executing them and checking results, evaluating whether enough testing has been done by Examining completion (or exit) criteria, reporting on the testing process and system under test, and presenting test completion (or summary) reports.

1.2.4 When can we meet our test objectives?

Testing Principle - Early testing

Testing activities should start as early as possible in the software or system development life cycle and should be focused on defined objectives.

We can use both dynamic testing and static testing as a means for achieving similar **test objectives.** Both provide information to improve both the system to be tested, and the development and testing processes. We mentioned above that testing can have different goals and objectives, which often include:

- finding defects;
- gaining confidence in and providing information about the level

of quality;
- preventing defects.

Many types of review and testing activities-take place at different stages in the life cycle, as we'll see in Chapter 2. These have different objectives. Early testing - such as early test design and review activities - finds defects early on when they are cheap to find and fix. Once the code is written, programmers and testers often run a set of tests so that they can identify and fix defects in the software. In this 'development testing' (which includes component, integration and system testing), the main objective may be to cause as many failures as possible so that defects in the software are identified and can be fixed. Following that testing, the users of the software may carry out acceptance testing to confirm that the system works as expected and to gain confidence that it has met the requirements.

Fixing the defects may not always be the test objective or the desired outcome. Sometimes we simply want to gather information and measure the software. This can take the form of attribute measures such as mean time between failures to assess reliability, or an assessment of the defect density in the software to assess and understand the risk of releasing it.

When maintaining software by enhancing it or fixing bugs, we are changing software that is already being used. In that case an objective of testing may be to ensure that we have not made errors and introduced defects when we changed the software. This is called regression testing - testing to ensure nothing has changed that should not have changed.

We may continue to test the system once it is in operational use. In this case, the main objective may be to assess system characteristics such as reliability or availability.

Testing Principle - Defect clustering

A small number of modules contain most of the defects discovered during pre-release testing or show the most operational failures.

1.2.5 Focusing on defects can help us plan our tests

Reviewing defects and failures in order to improve processes allows us to improve our testing and our requirements, design and development processes. One phenomenon that many testers have observed is that defects tend to cluster. This can happen because an area of the code is particularly complex and tricky, or because changing software and other products tends to cause knock-on defects. Testers will often use this information when making their risk assessment for planning the tests, and will focus on known 'hot spots'.

A main focus of reviews and other static tests is to carry out testing as early as possible, finding and fixing defects more cheaply and preventing defects from appearing at later stages of this project. These activities help us find out about defects earlier and identify potential clusters. Additionally, an important outcome of all testing is information that assists in risk assessment; these reviews will contribute to the planning for the tests executed later in the software development life cycle. We might also carry out root cause analysis to prevent defects and failures happening again and perhaps to identify the cause of clusters and potential future clusters.

1.2.6 The defect clusters change over time

Testing Principle - Pesticide paradox

If the same tests are repeated over and over again, eventually the same set of test cases will no longer find any new bugs. To overcome this 'pesticide paradox', the test cases need to be regularly reviewed and revised, and new and different tests need to be written to exercise different parts of the software or system to potentially find more defects.

Over time, as we improve our whole software development life cycle and the early static testing, we may well find that dynamic test levels find fewer defects. A typical test improvement initiative will initially find more defects as the testing improves and then, as the defect prevention kicks in, we see the defect numbers dropping, as shown in Figure 1.3. The first part of the improvement enables us to reduce failures in operation; later the improvements are making us more efficient and effective in producing the software with fewer defects in it.

As the 'hot spots' for bugs get cleaned up we need to move our focus elsewhere, to the next set of risks. Over time, our focus may change from finding coding bugs, to looking at the requirements and design documents for defects, and to looking for process improvements so that we prevent defects in the product. Referring to Figure 1.3, by releases 9 and 10, we would expect that the overall cost and effort associated with reviews and testing is much lower than in releases 4 or 7.

1.2.7 Debugging removes defects

When a test finds a defect that must be fixed, a programmer must do some work to locate the defect in the code and make the fix. In this process, called **debugging,** a programmer will examine the code for the immediate cause of the problem, repair the code and check that the code now executes as expected. The fix is often then tested separately (e.g. by an independent tester) to confirm the fix. Notice that testing and debugging are different activities. Developers may test their own fixes, in which case the very tight cycle of identifying faults,

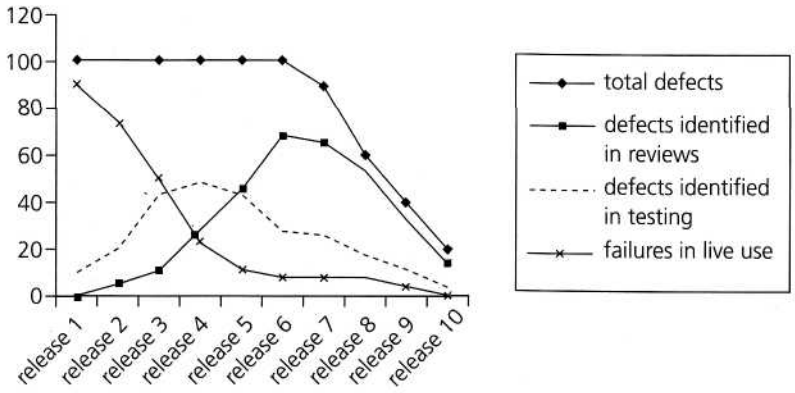

FIGURE 1.3 Changes in defect numbers during process improvement

debugging, and retesting is often loosely referred to as debugging. However, often following the debugging cycle the fixed code is tested independently both to retest the fix itself and to apply regression testing to the surrounding unchanged software.

1.2.8 Is the software defect free?

Testing Principle - Testing shows presence of defects

Testing can show that defects are present, but cannot prove that there are no defects. Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, it is not a proof of correctness.

This principle arises from the theory of the process of scientific experimentation and has been adopted by testers; you'll see the idea in many testing books. While you are not expected to read the scientific theory [Popper] the analogy used in science is useful; however many white swans we see, we cannot say 'All swans are white'. However, as soon as we see one black swan we can say 'Not all swans are white'. In the same way, however many tests we execute without finding a bug, we have not shown 'There are no bugs'. As soon as we find a bug, we have shown 'This code is not bug-free'.

1.2.9 If we don't find defects does that mean the users will accept the software?

Testing Principle - Absence of errors fallacy

Finding and fixing defects does not help if the system built is unusable and does not fulfill the users' needs and expectations.

There is another important principle we must consider; the customers for software - the people and organizations who buy and use it to aid in their day-to-day tasks - are not interested in defects or numbers of defects, except when they are directly affected by the instability of the software. The people using software are more interested in the software supporting them in completing tasks efficiently and effectively. The software must meet their needs. It is for this reason that the requirements and design defects we discussed earlier are so important, and why reviews and inspections (see Chapter 3) are such a fundamental part of the entire test activity.

1.3 TESTING PRINCIPLES

1 Explain the fundamental principles in testing. (K2)

In Sections 1.1 and 1.2, we have introduced a number of testing principles and brief explanations. These are listed in Table 1.2, for you to read over to remind yourself about them. These principles have been suggested over the past 40 years and offer general guidelines common for all testing.

TABLE 1.2 Testing principles

Principle 1:	**Testing shows presence of defects**	Testing can show that defects are present, but cannot prove that there are no defects. Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, it is not a proof of correctness.
Principle 2:	**Exhaustive testing is impossible**	Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Instead of exhaustive testing, we use risks and priorities to focus testing efforts.
Principle 3:	**Early testing**	Testing activities should start as early as possible in the software or system development life cycle and should be focused on defined objectives.
Principle 4:	**Defect clustering**	A small number of modules contain most of the defects discovered during pre-release testing or show the most operational failures.
Principle 5:	**Pesticide paradox**	If the same tests are repeated over and over again, eventually the same set of test cases will no longer find any new bugs. To overcome this 'pesticide paradox', the test cases need to be regularly reviewed and revised, and new and different tests need to be written to exercise different parts of the software or system to potentially find more defects.
Principle 6:	**Testing is context dependent**	Testing is done differently in different contexts. For example, safety-critical software is tested differently from an e-commerce site.
Principle 7:	**Absence-of-errors fallacy**	Finding and fixing defects does not help if the system built is unusable and does not fulfill the users' needs and expectations.

1.4 FUNDAMENTAL TEST PROCESS

1 Recall the fundamental test activities from planning to test closure activities and the main tasks of each test activity. (Kl)

1.4.1 Introduction

In this section, we will describe the fundamental test process and activities. These start with test planning and continue through to test closure. For each part of the test process, we'll discuss the main tasks of each test activity.

In this section, you'll also encounter the glossary terms **confirmation testing, exit criteria, incident, regression testing, test basis, test condition, test coverage, test data, test execution, test log, test plan, test strategy, test summary report** and **testware.**

As we have seen, although executing tests is important, we also need a plan of action and a report on the outcome of testing. Project and test plans should include time to be spent on planning the tests, designing test cases, preparing for execution and evaluating status. The idea of a fundamental test process for all levels of test has developed over the years. Whatever the level of testing, we see the same type of main activities happening, although there may be a different amount of formality at the different levels, for example, component tests might be carried out less formally than system tests in most organizations with a less documented test process. The decision about the level of formality of the processes will depend on the system and software context and the level of risk associated with the software. So we can divide the activities within the fundamental test process into the following basic steps:

- planning and control;
- analysis and design;
- implementation and execution;
- evaluating exit criteria and reporting;
- test closure activities.

These activities are logically sequential, but, in a particular project, may overlap, take place concurrently and even be repeated. This process is particularly used for dynamic testing, but the main headings of the process can be applied to reviews as well. For example, we need to plan and prepare for reviews, carry out the reviews, and evaluate the outcomes of the reviews. For some reviews, such as inspections, we will have exit criteria and will go through closure activities. However, the detail and naming of the activities will be different for static testing. We'll discuss static testing in Chapter 3.

1.4.2 Test planning and control

During **test planning,** we make sure we understand the goals and objectives of the customers, stakeholders, and the project, and the risks which testing is intended to address. This will give us what is sometimes called the mission of testing or the test assignment. Based on this understanding, we set the goals and objectives for the testing itself, and derive an approach and plan for the tests, including specification of test activities. To help us we may have organization or program **test policies** and a **test strategy.** Test policy gives rules for testing, e.g. 'we always review the design documents'; test strategy is the overall high-level approach, e.g. 'system testing is carried out by an independent team reporting to the program quality manager. It will be risk-based and proceeds from a product (quality) risk analysis' (see Chapter 5). If policy and strategy are defined already they drive our planning but if not we should ask for them to be stated and defined. Test planning has the following major tasks, given approximately in order, which help us build a test plan:

- Determine the scope and risks and identify the objectives of testing: we con sider what software, components, systems or other products are in scope for testing; the business, product, project and technical risks which need to be addressed; and whether we are testing primarily to uncover defects, to show that the software meets requirements, to demonstrate that the system is fit for purpose or to measure the qualities and attributes of the software.
- Determine the **test approach** (techniques, test items, **coverage,** identifying and interfacing with the teams involved in testing, testware): we consider how we will carry out the testing, the techniques to use, what needs testing and how extensively (i.e. what extent of coverage). We'll look at who needs to get involved and when (this could include developers, users, IT infrastruc ture teams); we'll decide what we are going to produce as part of the testing (e.g. testware such as test procedures and test data). This will be related to the requirements of the test strategy.
- Implement the test policy and/or the test strategy: we mentioned that there may be an organization or program policy and strategy for testing. If this is the case, during our planning we must ensure that what we plan to do adheres to the policy and strategy or we must have agreed with stakeholders, and documented, a good reason for diverging from it.
- Determine the required test resources (e.g. people, test environment, PCs): from the planning we have already done we can now go into detail; we decide on our team make-up and we also set up all the supporting hardware and software we require for the test environment.
- Schedule test analysis and design tasks, test implementation, execution and evaluation: we will need a schedule of all the tasks and activities, so that we can track them and make sure we can complete the testing on time.
- Determine the **exit criteria:** we need to set criteria such as coverage criteria (for example, the percentage of statements in the software that must be executed during testing) that will help us track whether we are completing the test activ ities correctly. They will show us which tasks and checks we must complete for a particular level of testing before we can say that testing is finished.

Management of any activity does not stop with planning it. We need to control and measure progress against the plan. So, **test control** is an ongoing activity. We need to compare actual progress against the planned progress, and report to the project manager and customer on the current status of testing, including any changes or deviations from the plan. We'll need to take actions where necessary to meet the objectives of the project. Such actions may entail changing our original plan, which often happens. When different groups perform different review and test activities within the project, the planning and control needs to happen within each of those groups but also across the groups to coordinate between them, allowing smooth hand-offs between each stage of testing. Test planning takes into account the feedback from **monitoring** and control activities which take place through out the project. Test control has the following major tasks:

- Measure and analyze the results of reviews and testing: We need to know how many reviews and tests we have done. We need to track how many tests have passed and how many failed, along with the number, type and importance of the defects reported.
- Monitor and document progress, test coverage and exit criteria: It is important that we inform the project team how much testing has been done, what the results are, and what conclusions and risk assessment we have made. We must make the test outcome visible and useful to the whole team.
- Provide information on testing: We should expect to make regular and exceptional reports to the project manager, project sponsor, customer and other key stakeholders to help them make informed decisions about project status. We should also use the information we have to analyze the testing itself.
- Initiate corrective actions: For example, tighten exit criteria for defects fixed, ask for more effort to be put into debugging or prioritize defects for fixing test blockers.
- Make decisions: Based on the measures and information gathered during testing and any changes to business and project risks or our increased under standing of technical and product risks, we'll make decisions or enable others to make decisions: to continue testing, to stop testing, to release the software or to retain it for further work for example.

1.4.3 Test analysis and design

Test analysis and design is the activity where general testing objectives are transformed into tangible test conditions and test designs. During test analysis and design, we take general testing objectives identified during planning and build test designs and test procedures (scripts). You'll see how to do this in Chapter 4. Test analysis and design has the following major tasks, in approximately the following order:

- Review the **test basis** (such as the product risk analysis, requirements, archi tecture, design specifications, and interfaces), examining the specifications for the software we are testing. We use the test basis to help us build our tests. We can start designing certain kinds of tests (called black-box tests)

before the code exists, as we can use the test basis documents to understand what the system should do once built. As we study the test basis, we often identify gaps and ambiguities in the specifications, because we are trying to identify precisely what happens at each point in the system, and this also prevents defects appearing in the code.

- Identify **test conditions** based on analysis of test items, their specifications, and what we know about their behavior and structure. This gives us a high-level list of what we are interested in testing. If we return to our driving example, the examiner might have a list of test conditions including 'behav ior at road junctions', 'use of indicators', 'ability to maneuver the car' and so on. In testing, we use the test techniques to help us define the test condi tions. From this we can start to identify the type of generic test data we might need.
- Design the tests (you'll see how to do this in Chapter 4), using techniques to help select representative tests that relate to particular aspects of the soft ware which carry risks or which are of particular interest, based on the test conditions and going into more detail. For example, the driving examiner might look at a list of test conditions and decide that junctions need to include T-junctions, cross roads and so on. In testing, we'll define the test case and test procedures.
- Evaluate testability of the requirements and system. The requirements may be written in a way that allows a tester to design tests; for example, if the per formance of the software is important, that should be specified in a testable way. If the requirements just say 'the software needs to respond quickly enough' that is not testable, because 'quick enough' may mean different things to different people. A more testable requirement would be 'the soft ware needs to respond in 5 seconds with 20 people logged on'. The testabil ity of the system depends on aspects such as whether it is possible to set up the system in an environment that matches the operational environment and whether all the ways the system can be configured or used can be understood and tested. For example, if we test a website, it may not be possible to iden tify and recreate all the configurations of hardware, operating system, browser, connection, firewall and other factors that the website might encounter.
- Design the test environment set-up and identify any required infrastructure and tools. This includes testing tools (see Chapter 6) and support tools such as spreadsheets, word processors, project planning tools, and non-IT tools and equipment - everything we need to carry out our work.

1.4.4 Test implementation and execution

During test implementation and execution, we take the test conditions and make them into **test cases** and testware and set up the test environment. This means that, having put together a high-level **design** for our tests, we now start to build them. We transform our test conditions into test cases and **procedures,** other testware such as scripts for automation. We also need to set up an environment where we will run the tests and build our **test data.** Setting up environments and data often involves significant time and effort, so you should plan

and monitor this work carefully. Test implementation and execution have the following major tasks, in approximately the following order:

- Implementation:
  - Develop and prioritize our test cases, using the techniques you'll see in Chapter 4, and create test data for those tests. We will also write instructions for carrying out the tests (test procedures). For the driving examiner this might mean changing the test condition 'junc tions' to 'take the route down Mayfield Road to the junction with Summer Road and ask the driver to turn left into Summer Road and then right into Green Road, expecting that the driver checks mirrors, signals and maneuvers correctly, while remaining aware of other road users.' We may need to automate some tests using test harnesses and automated test scripts. We'll talk about automation more in Chapter 6.
  - Create **test suites** from the test cases for efficient **test execution.** A test suite is a logical collection of test cases which naturally work together. Test suites often share data and a common high-level set of objectives. We'll also set up a test execution schedule.
  - Implement and verify the environment. We make sure the test envi ronment has been set up correctly, possibly even running specific tests on it.

- Execution:
  - Execute the test suites and individual test cases, following our test proce dures. We might do this manually or by using test execution tools, accord ing to the planned sequence.
  - Log the outcome of test execution and record the identities and versions of the software under test, test tools and testware. We must know exactly what tests we used against what version of the software; we must report defects against specific versions; and the **test log** we keep provides an audit trail.
  - Compare actual results (what happened when we ran the tests) with expected results (what we anticipated would happen).
  - Where there are differences between actual and expected results, report discrepancies as **incidents.** We analyze them to gather further details about the defect, reporting additional information on the problem, identify the causes of the defect, and differentiate between problems in the software and other products under test and any defects in test data, in test documents, or mistakes in the way we exe cuted the test. We would want to log the latter in order to improve the testing itself.
  - Repeat test activities as a result of action taken for each discrepancy. We need to re-execute tests that previously failed in order to confirm a fix **(confirmation testing** or **re-testing).** We execute corrected tests and suites if there were defects in our tests. We test corrected software again to ensure that the defect was indeed fixed correctly (confirmation test) and that the programmers did not introduce defects in unchanged areas of the software and that fixing a defect did not uncover other defects **(regression testing).**

1.4.5 Evaluating exit criteria and reporting

Evaluating exit criteria is the activity where test execution is assessed against the defined objectives. This should be done for each test level, as for each we need to know whether we have done enough testing. Based on our risk assessment, we'll have set criteria against which we'll measure 'enough'. These criteria vary for each project and are known as exit criteria. They tell us whether we can declare a given testing activity or level complete. We may have a mix of coverage or completion criteria (which tell us about test cases that must be included, e.g. 'the driving test must include an emergency stop' or 'the software test must include a response measurement'), acceptance criteria (which tell us how we know whether the software has passed or failed overall, e.g. 'only pass the driver if they have completed the emergency stop correctly' or 'only pass the software for release if it meets the priority 1 requirements list') and process exit criteria (which tell us whether we have completed all the tasks we need to do, e.g. 'the examiner/tester has not finished until they have written and filed the end of test report'). Exit criteria should be set and evaluated for each test level. Evaluating exit criteria has the following major tasks:

- Check test logs against the exit criteria specified in test planning: We look to see what evidence we have for which tests have been executed and checked, and what defects have been raised, fixed, confirmation tested, or are out standing.
- Assess if more tests are needed or if the exit criteria specified should be changed: We may need to run more tests if we have not run all the tests we designed, or if we realize we have not reached the coverage we expected, or if the risks have increased for the project. We may need to change the exit criteria to lower them, if the business and project risks rise in impor tance and the product or technical risks drop in importance. Note that this is not easy to do and must be agreed with stakeholders. The test manage ment tools and test coverage tools that we'll discuss in Chapter 6 help us with this assessment.
- Write a **test summary report** for stakeholders: It is not enough that the testers know the outcome of the test. All the stakeholders need to know what testing has been done and the outcome of the testing, in order to make informed decisions about the software.

1.4.6 Test closure activities

During test closure activities, we collect data from completed test activities to consolidate experience, including checking and filing testware, and analyzing facts and numbers. We may need to do this when software is delivered. We also might close testing for other reasons, such as when we have gathered the information needed from testing, when the project is cancelled, when a particular milestone is achieved, or when a maintenance release or update is done. Test closure activities include the following major tasks:

- Check which planned deliverables we actually delivered and ensure all incident reports have been resolved through defect repair or deferral. For deferred defects, in other words those that remain open, we may request

a change in a future release. We document the-acceptance or rejection of the software system.

- Finalize and archive **testware,** such as scripts, the test environment, and any other test infrastructure, for later reuse. It is important to reuse whatever we can of testware; we will inevitable carry out maintenance testing, and it saves time and effort if our testware can be pulled out from a library of existing tests. It also allows us to compare the results of testing between software versions.
- Hand over testware to the maintenance organization who will support the software and make any bug fixes or maintenance changes, for use in con firmation testing and regression testing. This group may be a separate group to the people who build and test the software; the maintenance testers are one of the customers of the development testers; they will use the library of tests.
- Evaluate how the testing went and analyze lessons learned for future releases and projects. This might include process improvements for the soft ware development life cycle as a whole and also improvement of the test processes. If you reflect on Figure 1.3 again, we might use the test results to set targets for improving reviews and testing with a goal of reducing the number of defects in live use. We might look at the number of incidents which were test problems, with the goal of improving the way we design, execute and check our tests or the management of the test environments and data. This helps us make our testing more mature and cost-effective for the organization. This is documented in a test summary report or might be part of an overall project evaluation report.

1.5 THE PSYCHOLOGY OF TESTING

1 **Recall that the success** of **testing is influenced by psychological factors: (Kl)**
   - **clear objectives;**
   - **a balance of self-testing and independent testing;**
   - **recognition of courteous communication and feedback on defects.**
2 **Contrast the mindset of a tester and that of a developer. (K2)**

In this section, we'll discuss the various psychological factors that influence testing and its success. These include clear objectives for testing, the proper roles and balance of self-testing and independent testing, clear, courteous communication and feedback on defects. We'll also contrast the mindset of a tester and of a developer.

You'll find a single Syllabus term in this section, **independent testing,** and the glossary term, **independence.**

1.5.1 Independent testing - who is a tester?

The mindset we want to use while testing and reviewing is different from the one we use while analyzing or developing. By this we mean that, if we are building something we are working positively to solve problems in the design and to realize a product that meets some need. However, when we test or review a product, we are looking for defects in the product and thus are critical of it.

Suppose you were going to cook a meal to enter in a competition for chefs. You select the menu, collect the ingredients, cook the food, set the table, and serve the meal. If you want to win, you do each task as well as you can. Suppose instead you are one of the judges evaluating the competition meals. You examine everything critically, including the menu, the ingredients, the methods used, keeping to time and budget allowances, choice of ingredients, the elegance of the table setting and the serving, and the look and taste of the meal. To differentiate between the competition chefs, you'll praise every good aspect of their performances but you'll also note every fault and error each chef made. So it is with software testing: building the software requires a different mindset from testing the software.

We do not mean that a tester cannot be a programmer, or that a programmer cannot be a tester, although they often are separate roles. In fact, programmers *are* testers - they test the components which they build, and the integration of the components into the system. The good chef will be as critical as the competition judges of his own work, in order to prevent and rectify errors and defects before anyone notices them. So, with the right mindset, programmers can test their own code; indeed programmers do test their own code and find many problems, resolving them before anyone else sees the code. Business analysis and marketing staff should review their own requirements. System architects should review their own designs. However, we all know it is difficult to find our own mistakes. So, business analysts, marketing staff, architects and programmers often rely on others to help test their work. This other person might be a fellow analyst, designer or developer. A person who will use the software may help test it. Business analysts who worked on the requirements and design may perform some tests. Testing specialists - professional testers - are often involved. In fact, testing may involve a succession of people each carrying out a different level of testing. This allows an independent test of the system.

We'll look at the points in the software development life cycle where testing takes place in Chapter 2. You'll see there that several stages of reviews and testing are carried out throughout the life cycle and these may be independent reviews and tests. Early in the life cycle, reviews of requirements and design documents by someone other than the author helps find defects before coding starts and helps us build the right software. Following coding, the software can be tested independently. This degree of **independence** avoids author bias and is often more effective at finding defects and failures.

Several levels of independence can be identified, listed here from the lowest level of independence to the highest:

- tests by the person who wrote the item under test;
- tests by another person within the same team, such as another programmer;
- tests by a person from a different organizational group, such as an independ ent test team;

• tests designed by a person from a different-organization or company, such as outsourced testing or certification by an external body.

We should note, however, that independence is not necessarily the most important factor in good testing. Developers who know how to test and who are, like good chefs, self-critical, have the benefit of familiarity and the pride-of-work that comes with true professionalism. Such developers can efficiently find many defects in their own code. Some software development methodologies insist on developers designing tests before they start coding and executing those tests continuously as they change the code. This approach promotes early testing and early defect detection, which is cost effective. Remember, independent testing may be carried out at any level of testing and the choice of independence level depends on the risk in the particular context.

1.5.2 Why do we sometimes not get on with the rest of the team?

As well as independence, separation of the tester role from the developer role is also done to help focus effort and to provide the benefits of trained and professional testing resources. In many organizations, earlier stages of testing are carried out by the developers and integrators and later stages independently, either by a specialist test group or by the customers. However, this separation can lead to problems as well as advantages. The advantage of independence and focus may be lost if the inter-team relationships deteriorate, as we'll see.

Each organization and each project will have its own goals and objectives. Different stakeholders, such as the customers, the development team and the managers of the organization, will have different viewpoints about quality and have their own objectives. Because people and projects are driven by objectives, the stakeholder with the strongest views or the greatest influence over a group will define, consciously or subconsciously, what those objectives are. People tend to align their plans with these objectives. For example, depending on the objective, a tester might focus either on finding defects or on confirming that software works. But if one stakeholder is less influential during the project but more influential at delivery, there may be a clash of views about whether the testing has met its objectives. One manager may want the confirmation that the software works and that it is 'good enough' if this is seen as a way of delivering as fast as possible. Another manager may want the testing to find as many defects as possible before the software is released, which will take longer to do and will require time for fixing, re-testing and regression testing. If there are not clearly stated objectives and exit criteria for testing which all the stakeholders have agreed, arguments might arise, during the testing or after release, about whether 'enough' testing has been done.

Many of us find it challenging to actually enjoy criticism of our work. We usually believe that we have done our best to produce work (documents, code, tests, whatever) which is correct and complete. So when someone else identifies a defect, a mistake we have made, we might take this personally and get annoyed with the other person, especially if we are under time pressure. This is true of managers, staff, testers and developers. We all make mistakes and we sometimes get annoyed, upset or depressed when someone points them out. So,

when as testers we run a test which (from our viewpoint) is a good test that finds defects and failures in the software, we need to be careful how we react. We are pleased, of course, since we have found a good bug! But how will the requirements analyst, designer, developer, project manager and customer react? The people who build products may react defensively and perceive this reported defect as personal criticism against the product and against the author. The project manager may be annoyed with everyone for holding up the project. The customer may lose confidence in the product because he can see defects. Because testing can be seen as a destructive activity, we need to take care to report on defects and failures as objectively and politely as possible. If others are to see our work as constructive in the management of product risks, we need to be careful when we are reviewing and when we are testing:

- Communicate findings on the product in a neutral, fact-focused way without criticizing the person who created it. For example, write objective and factual incident reports and review findings.
  - Don't gloat - you are not perfect either!
  - Don't blame - any mistakes are probably by the group rather than an individual.
  - Be constructively critical and discuss the defect and how you are going to log it.
- Explain that by knowing about this now we can work round it or fix it so the delivered system is better for the customer.
  - Say what you liked and what worked, as well as what didn't work.
  - Show what the risk is honestly - not everything is high priority.
  - Don't just see the pessimistic side - give praise as well as criticism.
  - Show what risks have been uncovered and the benefits of the review or test.
- Start with collaboration rather than battles. Remind everyone of the common goal of better quality systems.
  - Be polite and helpful, collaborate with your colleagues.
  - Try to understand how the other person feels and why they react as they do.
  - Confirm that the other person has understood what you have said and vice versa.
  - Explain how the test or review helps the author - what's in it for him or her.
  - Offer your work to be reviewed, too.

It's our job as reviewers and testers to provide everyone with clear, objective information and to do this we go bug-hunting, defect-mining and failure-making. People who will make good reviewers and testers have the desire and ability to find problems, and this is true whether testing is their main job or part of their role as a developer. These people build up experience of where errors are likely to be made, and are characterized by their curiosity, professional pessimism, critical eye and attention to detail. However, unless we also have good interpersonal and communication skills, courtesy, understanding of others and a good attitude towards our peers, colleagues, customers, managers and the rest of the team, we will fail as testers because no-one will listen to us.

The tester and test leader need good interpersonal skills to communicate factual information about defects, progress and risks in a constructive way [Perry]. For the author of the software or document, defect information can help them improve their skills, but only if it is provided in a way that helps them. One book that you might find interesting in this context is *Six Thinking Hats* [de Bono]. It is not about testing but describes a way to communicate different information: facts; our emotions; pessimistic and optimistic thoughts; and creative ideas. When reviewing or testing, we need to communicate facts objectively, but the other types of information are useful too: 'This happened; this is how I felt about it; this is what was good; this is what might go wrong; here is a way we could solve the problem'. As part of supplying the risk assessment, we can help the managers and customers make risk-based decisions based on the cost and time impact of a defect. If we test and find a defect that would cost $15 000 to fix and re-test/regression test, is it worth fixing? If it would cause a business impact of $50 000 in the live environment the customer may want it fixed. If it has a potential business impact of $10 000 but any fix is difficult to do and likely to have adverse impact elsewhere, it may be better not to fix. By providing the team with information about the defect in terms they find useful, we can help them to make the right decision about fixing or not fixing the problems. Generally we say that defects found and fixed during testing will save time and money later and reduce risks, so we need to show that is the case in order for the testing to be valued.

To help you think about the psychology of testing, there is an exercise at the end of the chapter, following the practice examination questions.

CHAPTER REVIEW

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

From Section 1.1, you should now be able to explain why testing is necessary and support that explanation with examples and evidence. You should be able to give examples of negative consequences of a software defect or bug for people, companies, and the environment. You should be able to contrast a defect with its symptoms. You should be able to discuss the ways in which testing fits into and supports higher quality. You should know the glossary terms **bug, defect, error, failure, fault, mistake, quality, risk, software, testing** and **exhaustive testing.**

From Section 1.2, you should now know what testing is. You should be able to remember the common objectives of testing. You should be able to describe how testing can find defects, provide confidence and information and prevent defects. You should be able to explain the fundamental principles of testing, summarized in Section 1.3. You should know the glossary terms **code, debugging, development of software, requirement, review, test basis, test case, testing** and **test objective.**

From Section 1.4, you should now recognize the fundamental test process, as well as being aware of some other related ways to model the test process. You should be able to recall the main testing activities related to test planning and control, analysis and design, implementation and execution, evaluating exit criteria and reporting, and test closure. You should know the glossary terms **confirmation testing, exit criteria, incident, regression testing, test basis, test condition, test coverage, test data, test execution, test log, test plan, test strategy, test summary report** and **testware.**

Finally, from Section 1.5, you now should be able to explain the psychology of testing and how people influence testing success. You should recall the importance of clear objectives, the right mix of self-testing and independent testing and courteous, respectful communication between testers and others on the project team, especially about defects. You should be able to explain and contrast the mindsets of testers and programmers and why these differences can lead to conflicts. You should know the glossary term **independence.**

SAMPLE EXAM QUESTIONS

Question 1 A company recently purchased a commercial off-the-shelf application to automate their bill-paying process. They now plan to run an acceptance test against the package prior to putting it into production. Which of the following is their most likely reason for testing?

a. To build confidence in the application.
b. To detect bugs in the application.
c. To gather evidence for a lawsuit.
d. To train the users.

Question 2 According to the ISTQB Glossary, the word 'bug' is synonymous with which of the following words?

a. Incident
b. Defect
c. Mistake
d. Error

Question 3 According to the ISTQB Glossary, a risk relates to which of the following?

a. Negative feedback to the tester.
b. Negative consequences that will occur.
c. Negative consequences that could occur.
d. Negative consequences for the test object.

Question 4 Ensuring that test design starts during the requirements definition phase is important to enable which of the following test objectives?

a. Preventing defects in the system.
b. Finding defects through dynamic testing.
c. Gaining confidence in the system.
d. Finishing the project on time.

Question 5 A test team consistently finds between 90% and 95% of the defects present in the system under test. While the test manager understands that this is a good defect-detection percentage for her test team and industry, senior management and executives remain disappointed in the test group, saying that the test team misses too many bugs. Given that the users are generally happy with the system and that the failures which have occurred have generally been low impact, which of the following testing principles is most likely to help the test manager explain to these managers and executives why some defects are likely to be missed?

a. Exhaustive testing is impossible
b. Defect clustering
c. Pesticide paradox
d. Absence-of-errors fallacy

Question 6 According to the ISTQB Glossary, regression testing is required for what purpose?

a. To verify the success of corrective actions.
b. To prevent a task from being incorrectly consid ered completed.
c. To ensure that defects have not been introduced by a modification.
d. To motivate better unit testing by the program mers.

Question 7 Which of the following is most important to promote and maintain good relationships between testers and developers?

a. Understanding what managers value about testing.
b. Explaining test results in a neutral fashion.
c. Identifying potential customer work-arounds for bugs.
d. Promoting better quality software whenever possible.

Question 8 Which of the statements below is the best assessment of how the test principles apply across the test life cycle?

a. Test principles only affect the preparation for testing.
b. Test principles only affect test execution activi ties.
c. Test principles affect the early test activities such as review.
d. Test principles affect activities throughout the test life cycle.

EXERCISE: TEST PSYCHOLOGY

Read the email below, and see what clues you find to help you identify problems in the scenario described. Categorize the clues/problems as:

- possible people, psychology and attitude problems;
- other problems, e.g. possible test management and role problems, possible product problems.

Hi there!
Well, I nearly caused a panic today because I thought I had found a mega showstopper on the trading system we are testing. The test manager and others got involved examining databases first on the server and then on the gateway that feeds the clients, checking update logs from processes that ran overnight as well as checking data passed to the client. Eventually I found the problem. I had mis-clicked on a .bat file when running up a client and had run up the wrong client environment. By that time the test manager was ready to say a few short words in my ear, particularly as the development people had started to get involved and they have zero tolerance for mistakes made by testers. The only saving grace was that I found the mistake and not one of the developers.

It was, objectively, an interesting mistake. When you log into the server test environments, the panels always show the environment to which you are connected. In our case we have two test environments called Systest14 and Systest15 and my tests were set up in Systest15. To run up the clients, we have to run .bat files for either a 14 or 15 client. I had started two clients, that is two exchange participants, so I could do some trading between them.

It appears I started the first client OK in environment 15 but when I started the second, I accidentally moved the mouse a fraction so it ran the 14 .bat file that is next to it in the Explorer file list. To make matters worse, the client screens do not show the environment to which you are attached.

At first I felt a bit stupid having caused much hectic and wasted activity. On reflection I thought that if I, as a reasonably competent person, can make a mistake like this then something is wrong. On the server side when I log on to a test environment, I have to enter the environment name and it's shown on all the panels. On the client side, I run a client test environment by selecting a .bat file from a list of many and have to ensure I click on the right file. There is neither a display nor the ability to determine the client environment in which I am working.

So I am going to log this as a high priority, or even showstopper, error - the client does not show the environment. In real life terms, it means a real user could be connected to the production system and think he is connected to a test system and screw up trading. I know this happened once on the equities trading system, when a trader entered a load of test transactions into the production system by mistake and caused mayhem.

As an addendum to this story, a couple of days later one of the testers found what appeared to be another mega showstopper. He and the test manager spent three hours crawling all over the system before they discovered the 'error'. A new filter had been added to the client software to filter transactions displayed in panels by geographical market. Unknown to them, it was set to a default of the German market, whereas they thought they were in the UK market. Consequently, at first sight, it appeared there were fundamental problems with the network transaction bus and the message-broadcasting systems. Apart from the issue that they should have been informed of this change, it raised a similar problem to the one I had experienced -the client system does not display the market in which you are trading.

Well - I'm off for another happy day at the
office! All the best

EXERCISE SOLUTION

People, psychology and attitude problems include, for example:

- Poor relationships between the test team and the test manager, and the testers and developers, e.g. 'By that time the test manager was ready to say a few short words in my ear, particularly as the development people had started to get involved and they have zero tolerance for mistakes made by testers. The only saving grace was that I found the mistake and not one of the developers.'
- Emotive use of language - understandable in the circumstances but not suitable for reporting problems, e.g. 'Well, I nearly caused a panic today because I thought I had found a mega showstopper on the trading system we are testing,' and 'As an addendum to this story, a couple of days later one of the testers found what appeared to be another mega-showstopper.'
- Initial diffidence overcome by revisiting the problem - if one person can make this mistake then others will. 'At first I felt a bit stupid having caused much hectic and wasted activity. On reflection I thought that if I, as a reasonably competent person, can make a mistake like this then something is wrong.'
- Understandable use of sarcasm ... 'Well - I'm off for another happy day at the office!'

Other problems include test management and role problems, for example:

- Configuration management and release control - A new filter had been added to the client software to filter transactions displayed in panels by geographical market.'
- Configuration management, relationships, communications - Apart from the issue that they should have been informed of this change ....'
- Does the test manager really understand his role? 'He and the test manager spent three hours crawling all over the system before they discovered the "error",' and 'The test manager and others got involved examining databases.'

There are some product problems, although no functionality or technical problems. Not all the problems we encounter as testers are functionality or technical problems. There are some non-functional problems - specifically, usability - which indicate that a real user might be inconvenienced or worse by this problem:

- 'I had mis-clicked on a .bat file ...'
- 'In real life terms, it means a real user could be connected to the production system and think he is connected to a test system and screw up trading. I know this happened once ... when a trader entered a load of test transactions into the production system by mistake and caused mayhem.'
- 'It raised a similar problem to the one I had experienced - the client system does not display the market in which you are trading.'
- 'There is neither a display nor the ability to determine the client environment in which I am working.' And 'To make matters worse, the client screens do not show the environment to which you are attached.'
- 'Unknown to them, it was set to a default of the German market, whereas they thought they were in the UK market.'

Note that we will return to this exercise at the end of Chapter 5, where we deal with writing a good incident report.