**FOUNDATIONS OF SOFTWARE TESTING**

**ISTQB CERTIFICATION**

**Dorothy Graham**

**Erik van Veenendaal**

**Isabel Evans**

**Rex Black**

**CONTENTS**

*Acknowledgements* viii *Preface* ix

1 Fundamentals of testing 1

1.1 Why is testing necessary? 1

1.2 What is testing? 11

1.3 Testing principles 18

1.4 Fundamental test process 20

1.5 The psychology of testing 26

Chapter review 31

Sample exam questions 32

Exercise: Test psychology 33

Exercise solution 34

2 Testing throughout the software life cycle 35 2.1 Software development models 35

2.2 Test levels 41

2.3 Test types: the targets of testing 46

2.4 Maintenance testing 50

Chapter review 54

Sample exam questions 55

3 Static techniques 57

3.1 Reviews and the test process 57

3.2 Review process 59

3.3 Static analysis by tools 69

Chapter review 74

Sample exam questions 75

4 Test design techniques 77

4.1 Identifying test conditions and designing test cases 77 4.2 Categories of test design techniques 84

4.3 Specification-based or black-box techniques 87 4.4 Structure-based or white-box techniques 105 4.5 Experience-based techniques 112

4.6 Choosing a test technique 114

Chapter review 117

Sample exam questions 118

Exercises: Test design techniques 121

Exercise solutions 122

5 Test management 127

5.1 Test organization 127

5.2 Test plans, estimates, and strategies 132

5.3 Test progress monitoring and control 142

5.4 Configuration management 148

5.5 Risk and testing 149

5.6 Incident management 155

Chapter review 161

Sample exam questions 162

Exercise: Incident report 165

Exercise solution 166

6 Tool support for testing 169

6.1 Types of test tool 169

6.2 Effective use of tools: Potential benefits and risks 184 6.3 Introducing a tool into an organization 190 Chapter review 193

Sample exam questions 195

7 ISTQB Foundation Exam 197

Preparing for the exam 197

Taking the exam 199

Mock exam 201

**Glossary** 209

**Answers to sample exam questions** 227

**References** 231

**Authors** 237

**Companies** 239

**Index** 243

**CHAPTER 1**

**Fundamentals of testing**

I

n 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 dis tinctions 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 defi nition 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 mis takes. 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 hap pened 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 likeli hood 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 soft

ware 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 perplex ing technical or business problems, complex business processes, code or infra structure, 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 mag netic 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.

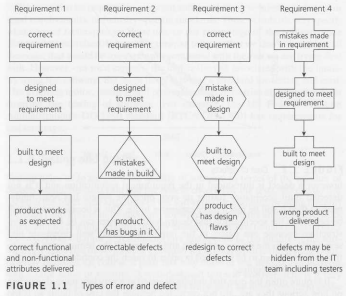
*When do defects arise?*

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 cor rectly to meet the design, and so deliver that requirement with the right attrib utes: 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.

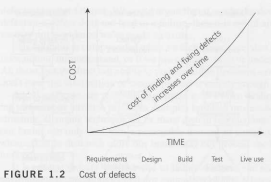
The defects introduced in requirement 3 are harder to deal with; we built exactly what we were told to but unfortunately the designer made some mis takes so there are defects in the design. Unless we check against the require ments definition, we will not spot those defects during testing. When we do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

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 observa tion 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 cor rected 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

however a defect is introduced in the requirement specification and it is not detected until acceptance testing or even once the system has been imple mented 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 con struction; 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 indus

try 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 soft

ware if it finds few or no defects, provided we are happy that the testing is suf ficiently 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 spec

ification?'). Of course, as well as wanting the right software system built cor rectly, 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 expecta tions. 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 cus tomers. 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 We will measure the attributes of the The tomatoes are the right size attributes of the product. software, e.g. its reliability in terms of and shape for packing for the mean time between failures (MBTF), supermarket. The tomatoes

and release when they reach a have a good taste and color,

specified level e.g. MTBF of 12 hours.

Quality is fitness for use. Quality can We will ask the users whether they The tomatoes are right for our have subjective aspects and not just can carry out their tasks; if they are recipe, quantitative aspects. satisfied that they can we will release

the software.

Quality is based on good manufacturing We will use a recognized software The tomatoes are organically processes, and meeting defined development process. We will only farmed. The tomatoes have no requirements. It is measured by testing, release the software if there are fewer blemishes and no pest inspection, and analysis of faults and than five outstanding high-priority damage, failures. defects once the planned tests are

complete.

Expectation of value for money. We have time-boxed the testing to The tomatoes have a good affordability, and a value-based trade-off two weeks to stay in the project shelf life. The tomatoes are between time, effort and cost aspects. budget. cheap or good value for We can afford to buy this software and money, we expect a return on investment.

Transcendent feelings - this is about the We like this software! It is fun and it's We get our tomatoes from a feelings of an individual or group of the latest thing! So what if it has a small local farm and we get on individuals towards a product or a few problems? We want to use it so well with the growers, supplier. anyway...

We really enjoy working with this

software team. So, there were a few

problems - they sorted them out

really quickly - we trust them.

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 inter

ested in using root cause analysis in your work, you'll find simple techniques described in [Evans], [TQMI] and [Robson]. For example, suppose an organi zation 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 exhaus tively. 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 prac tice, 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-

Section 2 What is testing? 11

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 combina

tions you would need 30 517 578 125 (515) 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 thor

ough, 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 prevent

ing 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 stake

holders, 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 dis cussed 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, inte gration and system testing), the main objective may be to cause as many fail ures 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 assess ment; 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

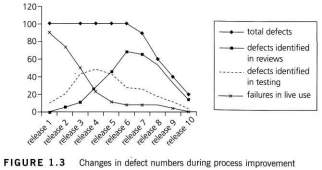
defect numbers dropping, as shown in Figure 1.3. The first part of the improvement enables us to reduce failures in operation; later the improve ments are making us more efficient and effective in producing the software with fewer defects in it.

As the 'hot spots' for bugs get cleaned up we need to move our focus else where, 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 **debug ging,** 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,



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 experimenta tion 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 soft ware - 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 soft ware 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 funda mental 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** Testing can show that defects are present, **presence of defects** 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** Testing everything (all combinations of **is impossible** 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** Testing is done differently in different **dependent** contexts. For example, safety-critical

software is tested differently from an

e-commerce site.

**Principle 7: Absence-of-errors** Finding and fixing defects does not help if **fallacy** 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 activi ties. 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 par ticularly 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 crite ria 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 approxi mately 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 trans formed 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 pre

vents 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 envi

ronment where we will run the tests and build our **test data.** Setting up environ ments 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 assess ment, we'll have set criteria against which we'll measure 'enough'. These crite ria 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 cov erage 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 infor

mation 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 com munication 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 build ing 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 ele gance 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 program mer cannot be a tester, although they often are separate roles. In fact, program mers *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, program mers 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 soft ware 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 methodolo gies 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, independ ent testing may be carried out at any level of testing and the choice of independ ence 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 relation

ships 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 require ments 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 pes simism, 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 cre ative ideas. When reviewing or testing, we need to communicate facts objec

tively, 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 pro viding 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 prob

lems. 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, debug ging, 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 cri

teria and reporting, and test closure. You should know the glossary terms **con firmation 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 impor tant to enable which of the following test objec tives?

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 relation ships 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.

CHAPTER 2

**Testing throughout the software life cycle**

T

esting is not a stand-alone activity. It has its place within a software development life cycle model and therefore the life cycle applied will largely determine how testing is organized. There are many different forms of testing. Because several disciplines, often with different interests,

are involved in the development life cycle, it is important to clearly understand and define the various test levels and types. This chapter discusses the most commonly applied software development models, test levels and test types. Maintenance can be seen as a specific instance of a development process. The way maintenance influences the test process, levels and types and how

testing can be organized is described in the last section of this chapter.

**2.1 SOFTWARE DEVELOPMENT MODELS**

**1 Understand the relationship between development, test activities and work products in the development life cycle and give examples based on project and product characteristics and context. (K2)**

**2 Recognize the fact that software development models must be adapted to the context of project and product characteristics. (Kl)**

**3 Recall reasons for different levels of testing and characteristics of good testing in any life cycle model. (Kl)**

The development process adopted for a project will depend on the project aims and goals. There are numerous development life cycles that have been developed in order to achieve different required objectives. These life cycles range from lightweight and fast methodologies, where time to market is of the essence, through to fully controlled and documented methodologies where quality and reliability are key drivers. Each of these methodologies has its place in modern software development and the most appropriate development process should be applied to each project. The

models specify the various stages of the process and the order in which they are carried out. The life cycle model that is adopted for a project will have a big impact on the testing that is carried out. Testing does not exist in isolation; test activities

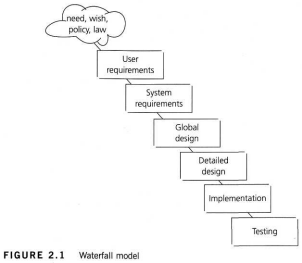
are highly related to software development activities. It will define the what, where, and when of our planned testing, influence regression testing, and largely determine which test techniques to use. The way testing is organized must fit the development life cycle or it will fail to deliver its benefit. If time to market is the key driver, then the testing must be fast and efficient. If a fully documented software development life cycle, with an audit trail of evidence, is required, the testing must be fully documented.

In every development life cycle, a part of testing is focused on **verification** testing and a part is focused on **validation** testing. Verification is concerned with evaluating a work product, component or system to determine whether it meets the requirements set. In fact, verification focuses on the question 'Is the deliverable built according to the specification?'. Validation is concerned with evaluating a work product, component or system to determine whether it meets the user needs and requirements. Validation focuses on the question 'Is the deliverable fit for purpose, e.g. does it provide a solution to the problem?'.

**2.1.1 V-model**

Before discussing the **V-model,** we will look at the model which came before it. The waterfall model was one of the earliest models to be designed. It has a natural timeline where tasks are executed in a sequential fashion. We start at the top of the waterfall with a feasibility study and flow down through the various

project tasks finishing with implementation into the live environment. Design flows through into development, which in turn flows into build, and finally on into test. Testing tends to happen towards the end of the project life cycle so defects are detected close to the live implementation date. With this model it has been difficult to get feedback passed backwards up the waterfall and there are

difficulties if we need to carry out numerous iterations for a particular phase.

The V-model was developed to address some of the problems experienced using the traditional waterfall approach. Defects were being found too late in the life cycle, as testing was not involved until the end of the project. Testing also added lead time due to its late involvement. The V-model pro

vides guidance that testing needs to begin as early as possible in the life cycle, which, as we've seen in Chapter 1, is one of the fundamental princi ples of structured testing. It also shows that testing is not only an execution based activity. There are a variety of activities that need to be performed before the end of the coding phase. These activities should be carried out in *parallel* with development activities, and testers need to work with devel opers and business analysts so they can perform these activities and tasks and produce a set of test deliverables. The work products produced by the developers and business analysts during development are the basis of testing in one or more levels. By starting test design early, defects are often found in the test basis documents. A good practice is to have testers involved even earlier, during the review of the (draft) test basis documents. The V-model is a model that illustrates how testing activities (verification and validation) can be integrated into each phase of the life cycle. Within the V-model, validation testing takes place especially during the early stages, e.g. reviewing the user requirements, and late in the life cycle, e.g. during user acceptance testing.

Although variants of the V-model exist, a common type of V-model uses four **test levels.** The four test levels used, each with their own objectives, are:

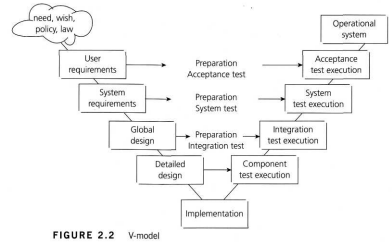
• component testing: searches for defects in and verifies the functioning of software components (e.g. modules, programs, objects, classes etc.) that are separately testable;

• integration testing: tests interfaces between components, interactions to dif ferent parts of a system such as an operating system, file system and hard ware or interfaces between systems;

• system testing: concerned with the behavior of the whole system/product as defined by the scope of a development project or product. The main focus of system testing is verification against specified requirements;

• acceptance testing: validation testing with respect to user needs, require ments, and business processes conducted to determine whether or not to accept the system.

The various test levels are explained and discussed in detail in Section 2.2. In practice, a V-model may have more, fewer or different levels of devel opment and testing, depending on the project and the software product. For example, there may be component integration testing after component testing and system integration testing after system testing. Test levels can be combined or reorganized depending on the nature of the project or the system architecture. For the integration of a **commercial off-the-shelf (COTS) software** product into a system, a purchaser may perform only inte gration testing at the system level (e.g. integration to the infrastructure and other systems) and at a later stage acceptance testing.

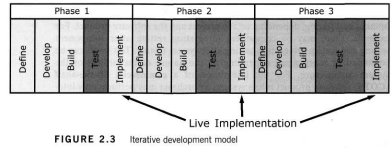


Note that the types of work products mentioned in Figure 2.2 on the left side of the V-model are just an illustration. In practice they come under many dif ferent names. References for generic work products include the Capability Maturity Model Integration (CMMi) or the 'Software life cycle processes' from ISO/IEC 12207. The CMMi is a framework for process improvement for both system engineering and software engineering. It provides guidance on where to focus and how, in order to increase the level of process maturity [Chrissis *et ah,* 2004]. ISO/IEC 12207 is an integrated software life cycle process standard that is rapidly becoming more popular.

2.1.2 Iterative life cycles

Not all life cycles are sequential. There are also iterative or incremental life cycles where, instead of one large development time line from beginning to end, we cycle through a number of smaller self-contained life cycle phases for the

same project. As with the V-model, there are many variants of iterative life cycles.



A common feature of iterative approaches is that the delivery is divided into increments or builds with each increment adding new functionality. The initial increment will contain the infrastructure required to support the initial build functionality. The increment produced by an iteration may be tested at several levels as part of its development. Subsequent increments will need testing for the new functionality, regression testing of the existing functionality, and inte gration testing of both new and existing parts. Regression testing is increasingly important on all iterations after the first one. This means that more testing will be required at each subsequent delivery phase which must be allowed for in the project plans. This life cycle can give early market presence with critical func tionality, can be simpler to manage because the workload is divided into smaller pieces, and can reduce initial investment although it may cost more in the long run. Also early market presence will mean validation testing is carried out at each increment, thereby giving early feedback on the business value and fitness for-use of the product.

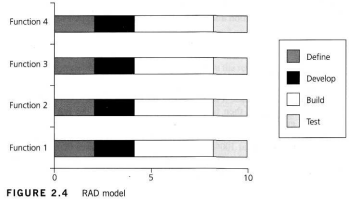
Examples of iterative or **incremental development models** are prototyping, Rapid Application Development (RAD), Rational Unified Process (RUP) and agile development. For the purpose of better understanding iterative develop ment models and the changing role of testing a short explanation of both RAD

and agile development is provided.

*Rapid Application Development*

Rapid Application Development (RAD) is formally a parallel development of functions and subsequent integration.

Components/functions are developed in parallel as if they were mini proj ects, the developments are time-boxed, delivered, and then assembled into a working prototype. This can very quickly give the customer something to see and use and to provide feedback regarding the delivery and their requirements. Rapid change and development of the product is possible using this methodol ogy. However the product specification will need to be developed for the product at some point, and the project will need to be placed under more formal controls prior to going into production. This methodology allows early



validation of technology risks and a rapid response to changing customer requirements.

Dynamic System Development Methodology [DSDM] is a refined RAD process that allows controls to be put in place in order to stop the process from getting out of control. Remember we still need to have the essentials of good development practice in place in order for these methodologies to work. We need to maintain strict configuration management of the rapid changes that we are making in a number of parallel development cycles. From the testing perspective we need to plan this very carefully and update our plans regularly as things will be changing very rapidly (see Chapter 5 for more on test plans).

The RAD development process encourages active customer feedback. The customer gets early visibility of the product, can provide feedback on the design and can decide, based on the existing functionality, whether to proceed with the development, what functionality to include in the next delivery cycle or even to halt the project if it is not delivering the expected value. An early business-focused solution in the market place gives an early return on investment (ROI) and can provide valuable marketing information for the business. Validation with the RAD development process is thus an early and major activity.

*Agile development*

Extreme Programming (XP) is currently one of the most well-known agile development life cycle models. (See [Agile] for ideas behind this approach.) The methodology claims to be more human friendly than traditional develop ment methods. Some characteristics of XP are:

• It promotes the generation of business stories to define the functionality. • It demands an on-site customer for continual feedback and to define and carry out functional acceptance testing.

• It promotes pair programming and shared code ownership amongst the developers.

• It states that component test scripts shall be written before the code is written and that those tests should be automated.

• It states that integration and testing of the code shall happen several times a day.

• It states that we always implement the simplest solution to meet today's problems.

With XP there are numerous iterations each requiring testing. XP develop ers write every test case they can think of and automate them. Every time a change is made in the code it is component tested and then integrated with the existing code, which is then fully integration-tested using the full set of test cases. This gives continuous integration, by which we mean that changes are incorporated continuously into the software build. At the same time, all test cases must be running at 100% meaning that all the test cases that have been identified and automated are executed and pass. XP is not about doing extreme activities during the development process, it is about doing known vajue-adding activities in an extreme manner.

2.1.3 Testing within a life cycle model

In summary, whichever life cycle model is being used, there are several charac teristics of good testing:

• for every development activity there is a corresponding testing activity; • each test level has test objectives specific to that level;

• the analysis and design of tests for a given test level should begin during the corresponding development activity;

• testers should be involved in reviewing documents as soon as drafts are avail able in the development cycle.

**2.2 TEST LEVELS**

**1 Compare the different levels of testing: major objectives, typical objects of testing, typical targets of testing (e.g. functional or structural) and related work products, people who test, types of defects and failures to be identified.** (K2)

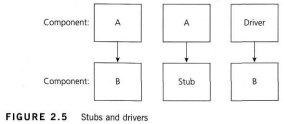
The V-model for testing was introduced in Section 2.1. This section looks in more detail at the various test levels. The key characteristics for each test level are discussed and defined to be able to more clearly separate the various test levels. A thorough understanding and definition of the various test levels will identify missing areas and prevent overlap and repetition. Sometimes we may wish to introduce deliberate overlap to address specific risks. Understanding whether we want overlaps and removing the gaps will make the test levels more complementary thus leading to more effective and efficient testing.

2.2.1 Component testing

**Component testing,** also known as unit, module and program testing, searches for defects in, and verifies the functioning of software (e.g. modules, programs, objects, classes, etc.) that are separately testable.

Component testing may be done in isolation from the rest of the system depend ing on the context of the development life cycle and the system. Most often **stubs** and **drivers** are used to replace the missing software and simulate the interface between the software components in a simple manner. A stub is called from the software component to be tested; a driver calls a component to be tested (see Figure 2.5).

Component testing may include testing of functionality and specific non functional characteristics such as resource-behavior (e.g. memory leaks), per formance or **robustness testing,** as well as structural testing (e.g. decision coverage). Test cases are derived from work products such as the software design or the data model.



Typically, component testing occurs with access to the code being tested and with the support of the development environment, such as a unit test frame work or debugging tool, and in practice usually involves the programmer who wrote the code. Sometimes, depending on the applicable level of risk, compo nent testing is carried out by a different programmer thereby introducing inde pendence. Defects are typically fixed as soon as they are found, without formally recording the incidents found.

One approach in component testing, used in Extreme Programming (XP), is to prepare and automate test cases before coding. This is called a test-first approach or **test-driven development.** This approach is highly iterative and is based on cycles of developing test cases, then building and integrating small pieces of code, and executing the component tests until they pass.

2.2.2 Integration testing

**Integration testing** tests interfaces between components, interactions to dif ferent parts of a system such as an operating system, file system and hard ware or interfaces between systems. Note that integration testing should be differentiated from other integration activities. Integration testing is often carried out by the integrator, but preferably by a specific integration tester or test team.

There may be more than one level of integration testing and it may be carried out on test objects of varying size. For example:

• component integration testing tests the interactions between software com ponents and is done after component testing;

• system integration testing tests the interactions between different systems and may be done after system testing. In this case, the developing organiza tion may control only one side of the interface, so changes may be destabi

lizing. Business processes implemented as workflows may involve a series of systems that can even run on different platforms.

The greater the scope of integration, the more difficult it becomes to isolate failures to a specific interface, which may lead to an increased risk. This leads to varying approaches to integration testing. One extreme is that all compo nents or systems are integrated simultaneously, after which everything is tested as a whole. This is called 'big-bang' integration testing. Big-bang testing has the advantage that everything is finished before integration testing starts. There is no need to simulate (as yet unfinished) parts. The major disadvantage is that in

general it is time-consuming and difficult to trace the cause of failures with this late integration. So big-bang integration may seem like a good idea when plan ning the project, being optimistic and expecting to find no problems. If one thinks integration testing will find defects, it is a good practice to consider whether time might be saved by breaking the down the integration test process. Another extreme is that all programs are integrated one by one, and a test is carried out after each step (incremental testing). Between these two extremes, there is a range of variants. The incremental approach has the advantage that the defects are found early in a smaller assembly when it is relatively easy to detect the cause. A disadvantage is that it can be time-consuming since stubs and drivers have to be developed and used in the test. Within incremental inte

gration testing a range of possibilities exist, partly depending on the system architecture:

• Top-down: testing takes place from top to bottom, following the control flow or architectural structure (e.g. starting from the GUI or main menu). Components or systems are substituted by stubs.

• Bottom-up: testing takes place from the bottom of the control flow upwards. Components or systems are substituted by drivers.

• Functional incremental: integration and testing takes place on the basis of the functions or functionality, as documented in the functional specification.

The preferred integration sequence and the number of integration steps required depend on the location in the architecture of the high-risk interfaces. The best choice is to start integration with those interfaces that are expected to cause most problems. Doing so prevents major defects at the end of the inte

gration test stage. In order to reduce the risk of late defect discovery, integra tion should normally be incremental rather than 'big-bang'. Ideally testers should understand the architecture and influence integration planning. If inte gration tests are planned before components or systems are built, they can be developed in the order required for most efficient testing.

At each stage of integration, testers concentrate solely on the integration itself. For example, if they are integrating component A with component B they are interested in testing the communication between the components, not the functionality of either one. Both functional and structural approaches may be used. Testing of specific non-functional characteristics (e.g. performance) may also be included in integration testing. Integration testing may be carried out by the developers, but can be done by a separate team of specialist integration testers, or by a specialist group of developers/integrators including non-func tional specialists.

2.2.3 System testing

**System testing** is concerned with the behavior of the whole system/product as defined by the scope of a development project or product. It may include tests based on risks and/or requirements specification, business processes, use cases, or other high level descriptions of system behavior, interactions with the oper

ating system, and system resources. System testing is most often the final test on behalf of development to verify that the system to be delivered meets the spec ification and its purpose may be to find as many defects as possible. Most often

it is carried out by specialist testers that form a dedicated, and sometimes inde pendent, test team within development, reporting to the development manager or project manager. In some organizations system testing is carried out by a third party team or by business analysts. Again the required level of independ ence is based on the applicable risk level and this will have a high influence on the way system testing is organized.

System testing should investigate both **functional** and **non-functional requirements** of the system. Typical non-functional tests include performance and reliability. Testers may also need to deal with incomplete or undocumented requirements. System testing of functional requirements starts by using the most appropriate specification-based (black-box) techniques for the aspect of the system to be tested. For example, a decision table may be created for com

binations of effects described in business rules. Structure-based (white-box) techniques may also be used to assess the thoroughness of testing elements such as menu dialog structure or web page navigation (see Chapter 4 for more on the various types of technique).

System testing requires a controlled **test environment** with regard to, amongst other things, control of the software versions, testware and the test data (see Chapter 5 for more on configuration management). A system test is executed by the development organization in a (properly controlled) environ

ment. The test environment should correspond to the final target or production environment as much as possible in order to minimize the risk of environment specific failures not being found by testing.

2.2.4 Acceptance testing

When the development organization has performed its system test and has cor rected all or most defects, the system will be delivered to the user or customer for **acceptance testing.** The acceptance test should answer questions such as: 'Can the system be released?', 'What, if any, are the outstanding (business) risks?' and 'Has development met their obligations?'. Acceptance testing is most often the responsibility of the user or customer, although other stakehold ers may be involved as well. The execution of the acceptance test requires a test environment that is for most aspects, representative of the production environ ment ('as-if production').

The goal of acceptance testing is to establish confidence in the system, part of the system or specific non-functional characteristics, e.g. usability, of the system. Acceptance testing is most often focused on a validation type of testing, whereby we are trying to determine whether the system is fit for purpose. Finding defects should not be the main focus in acceptance testing. Although it assesses the system's readiness for deployment and use, it is not necessarily the final level of testing. For example, a large-scale system integration test may come after the acceptance of a system.

Acceptance testing may occur at more than just a single level, for example:

• A Commercial Off The Shelf (COTS) software product may be acceptance tested when it is installed or integrated.

• Acceptance testing of the usability of a component may be done during com ponent testing.

• Acceptance testing of a new functional enhancement may come before system testing.

Within the acceptance test for a business-supporting system, two main test types can be distinguished; as a result of their special character, they are usually prepared and executed separately. The user acceptance test focuses mainly on the functionality thereby validating the fitness-for-use of the system by the business user, while the **operational acceptance test** (also called production acceptance test) validates whether the system meets the requirements for operation. The user acceptance test is performed by the users and application managers. In terms of planning, the user acceptance test usually links tightly to the system test and will, in many cases, be organ ized partly overlapping in time. If the system to be tested consists of a number of more or less independent subsystems, the acceptance test for a subsystem that complies to the exit criteria of the system test can start while another subsystem may still be in the system test phase. In most organiza

tions, system administration will perform the operational acceptance test shortly before the system is released. The operational acceptance test may include testing of backup/restore, disaster recovery, maintenance tasks and periodic check of security vulnerabilities.

Other types of acceptance testing that exist are contract acceptance testing and **compliance acceptance testing.** Contract acceptance testing is performed against a contract's acceptance criteria for producing custom-developed soft ware. Acceptance should be formally defined when the contract is agreed. Compliance acceptance testing or regulation acceptance testing is performed against the regulations which must be adhered to, such as governmental, legal or safety regulations.

If the system has been developed for the mass market, e.g. commercial off the-shelf software (COTS), then testing it for individual users or customers is not practical or even possible in some cases. Feedback is needed from potential or existing users in their market before the software product is put out for sale commercially. Very often this type of system undergoes two stages of accept ance test. The first is called **alpha testing.** This test takes place at the devel oper's site. A cross-section of potential users and members of the developer's organization are invited to use the system. Developers observe the users and note problems. Alpha testing may also be carried out by an independent test team. **Beta testing,** or field testing, sends the system to a cross-section of users who install it and use it under real-world working conditions. The users send records of incidents with the system to the development organization where the defects are repaired.

Note that organizations may use other terms, such as factory acceptance testing and site acceptance testing for systems that are tested before and after being moved to a customer's site.

**2.3 TEST TYPES: THE TARGETS OF TESTING**

**1 Compare four software test types (functional, non-functional, structural**

**and change-related) by example. (K2)**

**2 Recognize that functional and structural tests occur at any test level.**

**(Kl)**

**3 Identify and describe non-functional test types based on non functional requirements. (K2)**

**4 Identify and describe test types based on the analysis of a software**

**system's structure or architecture. (K2)**

**5 Describe the purpose of confirmation testing and regression testing.**

**(K2)**

Test types are introduced as a means of clearly defining the objective of a certain test level for a programme or project. We need to think about differ ent types of testing because testing the functionality of the component or system may not be sufficient at each level to meet the overall test objectives. Focusing the testing on a specific test objective and, therefore, selecting the appropriate type of test helps making and communicating decisions against test objectives easier.

A **test type** is focused on a particular test objective, which could be the testing of a function to be performed by the component or system; a non functional quality characteristic, such as reliability or usability; the structure or architecture of the component or system; or related to changes, i.e. con firming that defects have been fixed (confirmation testing, or re-testing) and looking for unintended changes (regression testing). Depending on its objec tives, testing will be organized differently. For example, component testing aimed at performance would be quite different to component testing aimed at achieving decision coverage.

2.3.1 Testing of function (functional testing)

The function of a system (or component) is 'what it does'. This is typically described in a requirements specification, a functional specification, or in use cases. There may be some functions that are 'assumed' to be provided that are not documented that are also part of the requirement for a system, though it is difficult to test against undocumented and implicit requirements. Functional tests are based on these functions, described in documents or understood by the testers and may be performed at all test levels (e.g. test for components may be based on a component specification).

**Functional testing** considers the specified behavior and is often also referred to as **black-box testing.** This is not entirely true, since black-box testing also includes non-functional testing (see Section 2.3.2).

**Function** (or functionality) **testing** can, based upon ISO 9126, be done focus ing on suitability, **interoperability, security,** accuracy and compliance. Security testing, for example, investigates the functions (e.g. a firewall) relating to detec tion of threats, such as viruses, from malicious outsiders.

Testing functionality can be done from two perspectives: requirements-based or business-process-based.

Requirements-based testing uses a specification of the functional require ments for the system as the basis for designing tests. A good way to start is to use the table of contents of the requirements specification as an initial test inventory or list of items to test (or not to test). We should also prioritize the requirements based on risk criteria (if this is not already done in the specifica tion) and use this to prioritize the tests. This will ensure that the most impor tant and most critical tests are included in the testing effort.

Business-process-based testing uses knowledge of the business processes. Business processes describe the scenarios involved in the day-to-day business use of the system. For example, a personnel and payroll system may have a busi ness process along the lines of: someone joins the company, he or she is paid on a regular basis, and he or she finally leaves the company. Use cases originate from object-oriented development, but are nowadays popular in many develop ment life cycles. They also take the business processes as a starting point, although they start from tasks to be performed by users. Use cases are a very useful basis for test cases from a business perspective.

The techniques used for functional testing are often **specification-based,** but experienced-based techniques can also be used (see Chapter 4 for more on test techniques). Test conditions and test cases are derived from the functionality of the component or system. As part of test designing, a model may be developed, such as a process model, state transition model or a plain-language specification.

**2.3.2 Testing of software product characteristics**

**(non-functional testing)**

A second target for testing is the testing of the quality characteristics, or non functional attributes of the system (or component or integration group). Here we are interested in how well or how fast something is done. We are testing something that we need to measure on a scale of measurement, for example time to respond.

Non-functional testing, as functional testing, is performed at all test levels. Non-functional testing includes, but is not limited to, **performance testing, load testing, stress testing,** usability testing, maintainability testing, reliability testing and portability testing. It is the testing of 'how well' the system works.

Many have tried to capture software quality in a collection of characteristics and related sub-characteristics. In these models some elementary characteris tics keep on reappearing, although their place in the hierarchy can differ. The International Organization for Standardization (ISO) has defined a set of quality characteristics [ISO/IEC 9126, 2001]. This set reflects a major step towards consensus in the IT industry and thereby addresses the general notion of software quality. The ISO 9126 standard defines six quality characteristics and the subdivision of each quality characteristic into a number of

sub-characteristics. This standard is getting more and more recognition in the industry, enabling development, testing and their stakeholders to use a common terminology for quality characteristics and thereby for non-functional testing.

The characteristics and their sub-characteristics are, respectively:

• functionality, which consists of five sub-characteristics: suitability, accuracy, security, interoperability and compliance; this characteristic deals with func tional testing as described in Section 2.3.1;

• **reliability,** which is defined further into the sub-characteristics maturity (robustness), fault-tolerance, recoverability and compliance; • **usability,** which is divided into the sub-characteristics understandability, learnability, operability, attractiveness and compliance;

• **efficiency,** which is divided into time behavior (performance), resource uti lization and compliance;

• **maintainability,** which consists of five sub-characteristics: analyzability, changeability, stability, testability and compliance;

• **portability,** which also consists of five sub-characteristics: adaptability, installability, co-existence, replaceability and compliance.

**2.3.3 Testing of software structure/architecture (structural testing)**

The third target of testing is the structure of the system or component. If we are talking about the structure of a system, we may call it the system architecture. Structural testing is often referred to as **'white-box'** or 'glass-box' because we are interested in what is happening 'inside the box'.

Structural testing is most often used as a way of measuring the thoroughness of testing through the coverage of a set of structural elements or coverage items. It can occur at any test level, although is it true to say that it tends to be mostly applied at component and integration and generally is less likely at higher test levels, except for business-process testing. At component integration level it may be based on the architecture of the system, such as a calling hierar

chy. A system, system integration or acceptance testing test basis could be a business model or menu structure.

At component level, and to a lesser extent at component integration testing, there is good tool support to measure **code coverage.** Coverage measurement tools assess the percentage of executable elements (e.g. state ments or decision outcomes) that have been exercised (i.e. covered) by a **test suite.** If coverage is not 100%, then additional tests may need to be written and run to cover those parts that have not yet been exercised. This of course depends on the exit criteria. (Coverage techniques are covered in Chapter 4.)

The techniques used for structural testing are structure-based techniques, also referred to as **white-box techniques.** Control flow models are often used to support structural testing.

**2.3.4 Testing related to changes (confirmation and regression testing)**

The final target of testing is the testing of changes. This category is slightly dif ferent to the others because if you have made a change to the software, you will have changed the way it functions, the way it performs (or both) and its struc ture. However we are looking here at the specific types of tests relating to changes, even though they may include all of the other test types.

*Confirmation testing (re-testing)*

When a test fails and we determine that the cause of the failure is a software defect, the defect is reported, and we can expect a new version of the software that has had the defect fixed. In this case we will need to execute the test again to confirm that the defect has indeed been fixed. This is known as **confirmation testing** (also known as re-testing).

When doing confirmation testing, it is important to ensure that the test is executed in exactly the same way as it was the first time, using the same inputs, data and environment. If the test now passes does this mean that the software is now correct? Well, we now know that at least one part of the software is correct - where the defect was. But this is not enough. The fix may have intro

duced or uncovered a different defect elsewhere in the software. The way to detect these 'unexpected side-effects' of fixes is to do regression testing.

*Regression testing*

Like confirmation testing, regression testing involves executing test cases that have been executed before. The difference is that, for regression testing, the test cases probably passed the last time they were executed (compare this with the test cases executed in confirmation testing - they failed the last time).

The term **'regression testing'** is something of a misnomer. It would be better if it were called 'anti-regression' testing because we are executing tests with the intent of checking that the system has not regressed (that is, it does not now have more defects in it as a result of some change). More specifically, the purpose of regression testing is to verify that modifications in the software or the environment have not caused unintended adverse side effects and that the system still meets its requirements.

It is common for organizations to have what is usually called a regression test suite or regression test pack. This is a set of test cases that is specifically used for regression testing. They are designed to collectively exercise most functions (cer tainly the most important ones) in a system but not test any one in detail. It is appropriate to have a regression test suite at every level of testing (component testing, integration testing, system testing, etc.). All of the test cases in a regression test suite would be executed every time a new version of software is produced and this makes them ideal candidates for **automation.** If the regression test suite is very large it may be more appropriate to select a subset for execution.

Regression tests are executed whenever the software changes, either as a result of fixes or new or changed functionality. It is also a good idea to execute them when some aspect of the environment changes, for example when a new version of a database management system is introduced or a new version of a source code compiler is used.

Maintenance of a regression test suite should be carried out so it evolves over time in line with the software. As new functionality is added to a system new regression tests should be added and as old functionality is changed or removed so too should regression tests be changed or removed. As new tests are added a regression test suite may become very large. If all the tests have to be executed manually it may not be possible to execute them all every time the regression suite is used. In this case a subset of the test cases has to be chosen. This selection should be made in light of the latest changes that have been made to the software. Sometimes a regression test suite of automated tests can become so large that it is not always possible to execute them all. It may be pos sible and desirable to eliminate some test cases from a large regression test suite for example if they are repetitive (tests which exercise the same condi

tions) or can be combined (if they are always run together). Another approach is to eliminate test cases that have not found a defect for a long time (though this approach should be used with some care!).

**2.4 MAINTENANCE TESTING**

**1 Compare maintenance testing (testing an operational system) to testing**

**a new application with respect to test types, triggers for testing and**

**amount of testing. (K2)**

**2 Identify reasons for maintenance testing (modifications, migration and**

**retirement). (K2)**

**3 Describe the role of regression testing and impact analysis in mainte**

**nance. (K2)**

Once deployed, a system is often in service for years or even decades. During this time the system and its operational environment is often corrected, changed or extended. Testing that is executed during this life cycle phase is called **'maintenance testing'.**

Note that maintenance testing is different from **maintainability testing,** which defines how easy it is to maintain the system.

The development and test process applicable to new developments does not change fundamentally for maintenance purposes. The same test process steps will apply and, depending on the size and risk of the changes made, several levels of testing are carried out: a component test, an integration test, a system test and an acceptance test. A maintenance test process usually begins with the receipt of an application for a change or a release plan. The test manager will use this as a basis for producing a test plan. On receipt of the new or changed specifications, corresponding test cases are specified or adapted. On receipt of the test object, the new and modified tests and the regression tests are executed. On completion of the testing, the testware is once again preserved.

Comparing maintenance testing to testing a new application is merely a matter of an approach from a different angle, which gives rise to a number of

changes in emphasis. There are several areas where most differences occur, for example regarding the test basis. A 'catching-up' operation is frequently required when systems are maintained. Specifications are often 'missing', and a set of testware relating to the specifications simply does not exist. It may well be possible to carry out this catching-up oper

ation along with testing a new maintenance release, which may reduce the cost. If it is impossible to compile any specifications from which test cases can be written, including expected results, an alternative test basis, e.g. a **test oracle,** should be sought by way of compromise. A search should be made for documentation which is closest to the specifications and which can be managed by developers as well as testers. In such cases it is advis

able to draw the customer's attention to the lower test quality which may be achieved. Be aware of possible problems of 'daily production'. In the worst case nobody knows what is being tested, many test cases are execut ing the same scenario and if an incident is found it is often hard to trace it back to the actual defect since no traceability to test designs and/or requirements exists. Note that reproducibility of tests is also important for maintenance testing.

One aspect which, in many cases, differs somewhat from the development situation is the test organization. New development and their appropriate test activities are usually carried out as parts of a project, whereas maintenance tests are normally executed as an activity in the regular organization. As a result, there is often some lack of resources and flexibility, and the test process may experience more competition from other activities.

2.4.1 Impact analysis and regression testing

Usually maintenance testing will consist of two parts:

• testing the changes

• regression tests to show that the rest of the system has not been affected by the maintenance work.

In addition to testing what has been changed, maintenance testing includes extensive regression testing to parts of the system that have not been changed. A major and important activity within maintenance testing is impact analysis. During **impact analysis,** together with stakeholders, a deci

sion is made on what parts of the system may be unintentionally affected and therefore need careful regression testing. Risk analysis will help to decide where to focus regression testing - it is unlikely that the team will have time to repeat all the existing tests.

If the test specifications from the original development of the system are kept, one may be able to reuse them for regression testing and to adapt them for changes to the system. This may be as simple as changing the expected results for your existing tests. Sometimes additional tests may need to be built. Extension or enhancement to the system may mean new areas have been spec

ified and tests would be drawn up just as for the development. It is also possi ble that updates are needed to an automated test set, which is often used to support regression testing.

2.4.2 Triggers for maintenance testing

As stated maintenance testing is done on an existing operational system. It is triggered by modifications, migration, or retirement of the system. Modifications include planned enhancement changes (e.g. release-based), cor rective and emergency changes, and changes of **environment,** such as planned operating system or database upgrades, or patches to newly exposed or discov ered vulnerabilities of the operating system. Maintenance testing for migration (e.g. from one platform to another) should include **operational testing** of the new environment, as well as the changed software. Maintenance testing for the retirement of a system may include the testing of data migration or archiving, if long data-retention periods are required.

Since modifications are most often the main part of maintenance testing foi most organizations, this will be discussed in more detail. From the point of view of testing, there are two types of modifications. There are modifications in which testing may be planned, and there are ad-hoc corrective modifications, which cannot be planned at all. Ad-hoc corrective maintenance takes place when the search for solutions to defects cannot be delayed. Special test proce

dures are required at that time.

*Planned modifications*

The following types of planned modification may be identified:

• perfective modifications (adapting software to the user's wishes, for instance by supplying new functions or enhancing performance);

• adaptive modifications (adapting software to environmental changes such as new hardware, new systems software or new legislation);

• corrective planned modifications (deferrable correction of defects).

The standard structured test approach is almost fully applicable to planned modifications. On average, planned modification represents over 90% of all maintenance work on systems. [Pol and van Veenendaal]

*Ad-hoc corrective modifications*

Ad-hoc corrective modifications are concerned with defects requiring an imme diate solution, e.g. a production run which dumps late at night, a network that goes down with a few hundred users on line, a mailing with incorrect addresses. There are different rules and different procedures for solving problems of this kind. It will be impossible to take the steps required for a structured approach to testing. If, however, a number of activities are carried out prior to a possible malfunction, it may be possible to achieve a situation in which reliable tests car. be executed in spite of 'panic stations' all round. To some extent this type of maintenance testing is often like first aid - patching up - and at a later stage the standard test process is then followed to establish a robust fix, test it and estab

lish the appropriate level of documentation.

A risk analysis of the operational systems should be performed in order to establish which functions or programs constitute the greatest risk to the opera tional services in the event of disaster. It is then established - in respect of the functions at risk - which (test) actions should be performed if a particular mal function occurs. Several types of malfunction may be identified and there are

various ways of responding to them for each function at risk. A possible reac tion might be that a relevant function at risk should always be tested, or that, under certain circumstances, testing might be carried out in retrospect (the next day, for instance). If it is decided that a particular function at risk should always be tested whenever relevant, a number of standard tests, which could be exe cuted almost immediately, should be prepared for this purpose. The standard tests would obviously be prepared and maintained in accordance with the struc tured test approach.

Even in the event of ad-hoc modifications, it is therefore possible to bring about an improvement in quality by adopting a specific test approach. It is important to make a thorough risk analysis of the system and to specify a set of standard tests accordingly.

**CHAPTER REVIEW**

Let's review what you have learned in this chapter. From Section 2.1, you should now understand the relationship between development and testing within a development life cycle, including the test activities and test (work) products. You should know that the development model to use should fit, or must be adapted to fit, the project and product characteristics. You should be able to recall the reasons for different levels of testing and characteristics of good testing in any life cycle model. You should know the glossary terms **(commercial) off-the-shelf software (COTS), incremental development model, test level, validation, verification** and **V-model.**

From Section 2.2, you should know the typical levels of testing. You should be able to compare the different levels of testing with respect to their major objectives, typical objects of testing, typical targets of testing (e.g. functional or structural) and related work products. You should also know which persons perform the testing activities at the various test levels, the types of defects found and failures to be identified. You should know the glossary terms **alpha testing, beta testing, component testing, driver, functional requirements, integration, integration testing, non-functional testing, operational testing, regulation acceptance testing (compliance testing), robustness testing, stub, system testing, test-driven development, test environment** and **user acceptance testing.**

From Section 2.3, you should know the four major types of test (functional, non-functional, structural and change-related) and should be able to provide some concrete examples for each of these. You should understand that functional and structural tests occur at any test level and be able to explain how they are applied in the various test levels. You should be able to identify and describe non-functional test types based on non-functional requirements and product quality characteristics. Finally you should be able to explain the purpose of confirmation testing (re testing) and regression testing in the context of change-related testing. You should know the glossary terms **black-box testing, code coverage, confirmation testing (re-testing), functional testing, interoperability testing, load testing, maintainability testing, performance testing, portability testing, regression testing, reliability testing, security testing, specification-based testing, stress testing, structural testing, test suite, usability testing** and **white-box testing**

From Section 2.4, you should be able to compare maintenance testing to testing of new applications. You should be able to identify triggers and reasons for maintenance testing, such as modifications, migration and retirement. Finally you should be able to describe the role of regression testing and impact analysis within maintenance testing. You should know the glossary terms **impact analysis** and **maintenance testing.**

**SAMPLE EXAM QUESTIONS**

Question 1 What are good practices for testing within the development life cycle?

a. Early test analysis and design.

b. Different test levels are defined with specific objectives.

c. Testers will start to get involved as soon as

coding is done.

d. A and B above.

Question 2 Which option best describes objec tives for test levels with a life cycle model? a. Objectives should be generic for any test level.

b. Objectives are the same for each test level. c. The objectives of a test level don't need to be defined in advance.

d. Each level has objectives specific to that level.

Question 3 Which of the following is a test type?

a. Component testing

b. Functional testing

c. System testing

d. Acceptance testing

Question 4 Which of the following is a non functional quality characteristic?

a. Feasibility

b. Usability

c. Maintenance

d. Regression

Question 5 Which of these is a functional test? a. Measuring response time on an on-line booking

system.

b. Checking the effect of high volumes of traffic in

a call-center system.

c. Checking the on-line bookings screen informa tion and the database contents against the infor

mation on the letter to the customers. d. Checking how easy the system is to use.

Question 6 Which of the following is a true statement regarding the process of fixing emergency changes?

a. There is no time to test the change before it goes live, so only the best developers should do

this work and should not involve testers as they

slow down the process.

b. Just run the retest of the defect actually fixed.

c. Always run a full regression test of the whole system in case other parts of the system have been adversely affected.

d. Retest the changed area and then use risk assessment to decide on a reasonable subset of

the whole regression test to run in case other parts of the system have been adversely affected.

Question 7 A regression test:

a. Is only run once.

b. Will always be automated.

c. Will check unchanged areas of the software to see if they have been affected.

d. Will check changed areas of the software to see

if they have been affected.

Question 8 Non-functional testing includes: a. Testing to see where the system does not func tion correctly.

b. Testing the quality attributes of the system

including reliability and usability.

c. Gaining user approval for the system. d. Testing a system feature using only the software

required for that function.

Question 9 Beta testing is:

a. Performed by customers at their own site. b. Performed by customers at the software devel oper's site.

c. Performed by an independent test team. d. Useful to test software developed for a specific

customer or user.

CHAPTER THREE

**Static techniques**

S

tatic test techniques provide a powerful way to improve the quality and productivity of software development. This chapter describes static test techniques, including reviews, and provides an overview of how they are conducted. The fundamental objective of static testing is to improve the quality of software work products by assisting engineers to recognize and fix their own defects early in the software development process. While static testing techniques will not solve all the problems, they are enormously effective. Static techniques can improve both quality and productivity by impressive factors. Static testing is not magic and it should not be considered a replacement for dynamic testing, but all software organizations should consider using reviews in all major aspects of their work including requirements, design, implementation, testing, and maintenance. Static analysis tools implement automated checks, e.g. on code.

**3.1 REVIEWS AND THE TEST PROCESS**

**1 Recognize software work products that can be examined by different static techniques. (Kl)**

**2 Describe the importance and value of considering static techniques for the assessment of software work products. (K2)**

**3 Explain the difference between static and dynamic techniques. (K2)**

In Chapter 1, several **testing** terms were presented. Also testing itself was defined. The latter definition is repeated here as a means for explaining the two major types of testing.

The definition of testing outlines objectives that relate to evaluation, revealing defects and quality. As indicated in the definition two approaches can be used to achieve these objectives, **static testing** and **dynamic testing.**

With dynamic testing methods, software is executed using a set of input values and its output is then examined and compared to what is expected. During static testing, software work products are examined manually, or with a set of tools, but not executed. As a consequence, dynamic testing can only be applied to software code. Dynamic execution is applied as a technique to detect

defects and to determine quality attributes of the code. This testing option is not applicable for the majority of the software work products. Among the ques tions that arise are: How can we evaluate or analyze a requirements document, a design document, a test plan, or a user manual? How can we effectively pre examine the source code before execution? One powerful technique that can be used is static testing, e.g. **reviews.** In principle all software work products can be tested using review techniques.

Dynamic testing and static testing are complementary methods, as they tend to find different types of defects effectively and efficiently. Types of defects that are easier to find during static testing are: deviations from standards, missing requirements, design defects, non-maintainable code and inconsistent interface specifications. Note that in contrast to dynamic testing, static testing finds defects rather than failures.

In addition to finding defects, the objectives of reviews are often also informational, communicational and educational, whereby participants learn about the content of software work products to help them understand the role of their own work and to plan for future stages of development. Reviews often represent project milestones, and support the establishment of a baseline for a software product. The type and quantity of defects found during reviews can also help testers focus their testing and select effective classes of tests. In some cases customers/users attend the review meeting and provide feedback to the development team, so reviews are also a means of customer/user communication.

Studies have shown that as a result of reviews, a significant increase in pro ductivity and product quality can be achieved [Gilb and Graham, 1993], [van Veenendaal, 1999]. Reducing the number of defects early in the product life cycle also means that less time has to be spent on testing and maintenance. To summarize, the use of static testing, e.g. reviews, on software work products has various advantages:

• Since static testing can start early in the life cycle, early feedback on quality issues can be established, e.g. an early validation of user requirements and not just late in the life cycle during acceptance testing.

• By detecting defects at an early stage, rework costs are most often relatively low and thus a relatively cheap improvement of the quality of software prod ucts can be achieved.

• Since rework effort is substantially reduced, development productivity figures are likely to increase.

• The evaluation by a team has the additional advantage that there is an exchange of information between the participants.

• Static tests contribute to an increased awareness of quality issues.

In conclusion, static testing is a very suitable method for improving the quality of software work products. This applies primarily to the assessed products themselves, but it is also important that the quality improvement is not achieved once but has a more structural character. The feedback from the static testing process to the development process allows for process improvement, which supports the avoidance of similar errors being made in the future.

**3,2 REVIEW PROCESS**

**1 Recall the phases, roles and responsibilities of a typical formal review.**

**(Kl)**

**2 Explain the differences between different types of review: informal**

**review, technical review, walkthrough and inspection. (K2) 3 Explain the factors for successful performance of reviews. (K2)**

Reviews vary from very **informal** to **formal** (i.e. well structured and regulated). Although inspection is perhaps the most documented and formal review tech nique, it is certainly not the only one. The formality of a review process is related to factors such as the maturity of the development process, any legal or regulatory requirements or the need for an audit trail. In practice the informal review is perhaps the most common type of review. Informal reviews are applied at various times during the early stages in the life cycle of a document. A two-person team can conduct an informal review, as the author can ask a col league to review a document or code. In later stages these reviews often involve more people and a meeting. This normally involves peers of the author, who try to find defects in the document under review and discuss these defects in a review meeting. The goal is to help the author and to improve the quality of the document. Informal reviews come in various shapes and forms, but all have one characteristic in common - they are not documented.

3.2.1 Phases of a formal review

In contrast to informal reviews, formal reviews follow a formal process. A typical formal review process consists of six main steps:

1 Planning

2 Kick-off

3 Preparation

4 Review meeting

5 Rework

**6** Follow-up.

*Planning*

The review process for a particular review begins with a 'request for review' by the author to the **moderator** (or inspection leader). A moderator is often assigned to take care of the scheduling (dates, time, place and invitation) of the review. On a project level, the project planning needs to allow time for review and rework activities, thus providing engineers with time to thoroughly participate in reviews. For more formal reviews, e.g. inspections, the moderator always performs an entry check and defines at this stage formal exit criteria. The entry check is

carried out to ensure that the reviewers' time is not wasted on a document that is not ready for review. A document containing too many obvious mistakes is clearly not ready to enter a formal review process and it could even be very harmful to the review process. It would possibly de-motivate both reviewers and the author. Also, the review is most likely not effective because the numerous obvious and minor defects will conceal the major defects.

Although more and other **entry criteria** can be applied, the following can be regarded as the minimum set for performing the entry check:

• A short check of a product sample by the moderator (or expert) does not reveal a large number of major defects. For example, after 30 minutes of checking, no more than 3 major defects are found on a single page or fewer than 10 major defects in total in a set of 5 pages.

• The document to be reviewed is available with line numbers. • The document has been cleaned up by running any automated checks that apply.

• References needed for the inspection are stable and available. • The document author is prepared to join the review team and feels confident with the quality of the document.

If the document passes the entry check, the moderator and author decide which part of the document to review. Because the human mind can com prehend a limited set of pages at one time, the number should not be too high. The maximum number of pages depends, among other things, on the objective, review type and document type and should be derived from prac tical experiences within the organization. For a review, the maximum size is usually between 10 and 20 pages. In formal inspection, only a page or two may be looked at in depth in order to find the most serious defects that are not obvious.

After the document size has been set and the pages to be checked have been selected, the moderator determines, in co-operation with the author, the com position of the review team. The team normally consists of four to six partici pants, including moderator and author. To improve the effectiveness of the review, different roles are assigned to each of the participants. These roles help the **reviewers** focus on particular types of defects during checking. This reduces the chance of different reviewers finding the same defects. The moderator assigns the roles to the reviewers.

Figure 3.1 shows the different roles within a review. The roles represent views of the document under review.

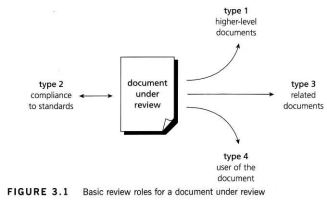
Within reviews the following focuses can be identified:

• focus on higher-level documents, e.g. does the design comply to the requirements;

• focus on standards, e.g. internal consistency, clarity, naming conventions, templates;

• focus on related documents at the same level, e.g. interfaces between soft ware functions;

• focus on usage, e.g. for testability or maintainability.

The author may raise additional specific roles and questions that have to be addressed. The moderator has the option to also fulfil a role, alongside the task of being a review leader. Checking the document improves the moderator's ability to

lead the meeting, because it ensures better understanding. Furthermore, it improves the review efficiency because the moderator replaces an engineer that would other wise have to check the document and attend the meeting. It is recommended that the moderator take the role of checking compliance to standards, since this tends to be a highly objective role, which leads to less discussion of the defects found.

*Kick-off*

An optional step in a review procedure is a kick-off meeting. The goal of this meeting is to get everybody on the same wavelength regarding the document under review and to commit to the time that will be spent on checking. Also the result of the entry check and defined exit criteria are discussed in case of a more

formal review. In general a kick-off is highly recommended since there is a strong positive effect of a kick-off meeting on the motivation of reviewers and thus the effectiveness of the review process. At customer sites, we have meas ured results up to 70% more major defects found per page as a result of per forming a kick-off, [van Veenendaal and van der Zwan, 2000]

During the kick-off meeting the reviewers receive a short introduction on the objectives of the review and the documents. The relationships between the doc ument under review and the other documents (sources) are explained, espe cially if the number of related documents is high.

Role assignments, checking rate, the pages to be checked, process changes and possible other questions are also discussed during this meeting. Of course the distribution of the document under review, source documents and other related documentation, can also be done during the kick-off.

*Preparation*

The participants work individually on the document under review using the related documents, procedures, rules and checklists provided. The individual participants identify defects, questions and comments, according to their

understanding of the document and role. All issues are recorded, preferably using a logging form. Spelling mistakes are recorded on the document under review but not mentioned during the meeting. The annotated document will be given to the author at the end of the logging meeting. Using checklists during this phase can make reviews more effective and efficient, for example a specific checklist based on perspectives such as user, maintainer, tester or operations, or a checklist for typical coding problems.

A critical success factor for a thorough preparation is the number of pages checked per hour. This is called the checking rate. The optimum checking rate is the result of a mix of factors, including the type of document, its com plexity, the number of related documents and the experience of the reviewer. Usually the checking rate is in the range of five to ten pages per hour, but may be much less for formal inspection, e.g. one page per hour. During preparation, participants should not exceed this criterion. By collecting data and measuring the review process, company-specific criteria for checking rate and document size (see planning phase) can be set, preferably specific to a document type.

*Review meeting*

The meeting typically consists of the following elements (partly depending on the review type): logging phase, discussion phase and decision phase. During the logging phase the issues, e.g. defects, that have been identified during the preparation are mentioned page by page, reviewer by reviewer and are logged either by the author or by a scribe. A separate person to do the logging (a scribe) is especially useful for formal review types such as an inspec tion. To ensure progress and efficiency, no real discussion is allowed during the logging phase. If an issue needs discussion, the item is logged and then handled in the discussion phase. A detailed discussion on whether or not an issue is a defect is not very meaningful, as it is much more efficient to simply log it and proceed to the next one. Furthermore, in spite of the opinion of the team, a dis cussed and discarded defect may well turn out to be a real one during rework. Every defect and its severity should be logged. The participant who identifies the defect proposes the severity. Severity classes could be:

• *Critical:* defects will cause downstream damage; the scope and impact of the defect is beyond the document under inspection.

• *Major,* defects could cause a downstream effect (e.g. a fault in a design can result in an error in the implementation).

• *Minor,* defects are not likely to cause downstream damage (e.g. non-compli ance with the standards and templates). ,

In order to keep the added value of reviews, spelling errors are not part of the defect classification. Spelling defects are noted, by the participants, in the document under review and given to the author at the end of the meeting or could be dealt with in a separate proofreading exercise.

During the logging phase the focus is on logging as many defects as possible within a certain timeframe. To ensure this, the moderator tries to keep a good logging rate (number of defects logged per minute). In a well-led and disci plined formal review meeting, the logging rate should be between one and two defects logged per minute.

For a more formal review, the issues classified as discussion items will be handled during this meeting phase. Informal reviews will often not have a sep arate logging phase and will start immediately with discussion. Participants can take part in the discussion by bringing forward their comments and rea soning. As chairman of the discussion meeting, the moderator takes care of people issues. For example, the moderator prevents discussions from getting too personal, rephrases remarks if necessary and calls for a break to cool down 'heated' discussions and/or participants.

Reviewers who do not need to be in the discussion may leave, or stay as a learning exercise. The moderator also paces this part of the meeting and ensures that all discussed items either have an outcome by the end of the meeting, or are defined as an action point if a discussion cannot be solved during the meeting. The outcome of discussions is documented for future reference.

At the end of the meeting, a decision on the document under review has to be made by the participants, sometimes based on formal **exit criteria.** The most important exit criterion is the average number of critical and/or major defects found per page (e.g. no more than three critical/major defects per page). If the number of defects found per page exceeds a certain level, the document must be reviewed again, after it has been reworked. If the document complies with the exit criteria, the document will be checked during follow-up by the moder ator or one or more participants. Subsequently, the document can leave the review process.

If a project is under pressure, the moderator will sometimes be forced to skip re-reviews and exit with a defect-prone document. Setting, and agreeing, quantified exit level criteria helps the moderator to make firm decisions at all times.

In addition to the number of defects per page, other exit criteria are used that measure the thoroughness of the review process, such as ensuring that all pages have been checked at the right rate. The average number of defects per page is only a valid quality indicator if these process criteria are met.

*Rework*

Based on the defects detected, the author will improve the document under review step by step. Not every defect that is found leads to rework. It is the author's responsibility to judge if a defect has to be fixed. If nothing is done about an issue for a certain reason, it should be reported to at least indicate that the author has considered the issue.

Changes that are made to the document should be easy to identify during follow-up. Therefore the author has to indicate where changes are made (e.g. using 'Track changes' in word-processing software).

*Follow-up*

The moderator is responsible for ensuring that satisfactory actions have been taken on all (logged) defects, process improvement suggestions and change requests. Although the moderator checks to make sure that the author has taken action on all known defects, it is not necessary for the moderator to check all the corrections in detail. If it is decided that all participants will check the updated document, the moderator takes care of the distribution and collects

***f***

the feedback. For more formal review types the moderator checks for compli ance to the exit criteria.

In order to control and optimize the review process, a number of measure ments are collected by the moderator at each step of the process. Examples of such measurements include number of defects found, number of defects found per page, time spent checking per page, total review effort, etc. It is the respon sibility of the moderator to ensure that the information is correct and stored for future analysis.

3.2.2 Roles and responsibilities

The participants in any type of formal review should have adequate knowledge of the review process. The best, and most efficient, review situation occurs when the participants gain some kind of advantage for their own work during review ing. In the case of an inspection or technical review, participants should have been properly trained as both types of review have proven to be far less success ful without trained participants. This indeed is perceived as being a critical success factor.

The best formal reviews come from well-organized teams, guided by trained moderators (or review leaders). Within a review team, four types of participants can be distinguished: moderator, author, scribe and reviewer. In addition man agement needs to play a role in the review process.

*The moderator*

The moderator (or review leader) leads the review process. He or she deter mines, in co-operation with the author, the type of review, approach and the composition of the review team. The moderator performs the entry check and the follow-up on the rework, in order to control the quality of the input and output of the review process. The moderator also schedules the meeting, disseminates documents before the meeting, coaches other team members, paces the meeting, leads possible discussions and stores the data that is collected.

*The author*

As the writer of the document under review, the author's basic goal should be to learn as much as possible with regard to improving the quality of the document, but also to improve his or her ability to write future documents. The author's task is to illuminate unclear areas and to understand the defects found.

*The scribe*

During the logging meeting, the **scribe** (or recorder) has to record each defect mentioned and any suggestions for process improvement. In practice it is often the author who plays this role, ensuring that the log is readable and understand able. If authors record their own defects, or at least make their own notes in their own words, it helps them to understand the log better during rework. However, having someone other than the author take the role of the scribe (e.g. the moderator) can have significant advantages, since the author is freed up to think about the document rather than being tied down with lots of writing.

*The reviewers*

The task of the reviewers (also called checkers or inspectors) is to check any material for defects, mostly prior to the meeting. The level of thoroughness required depends on the type of review. The level of domain knowledge or tech nical expertise needed by the reviewers also depends on the type of review. Reviewers should be chosen to represent different perspectives and roles in the review process. In addition to the document under review, the material review ers receive includes source documents, standards, checklists, etc. In general, the fewer source and reference documents provided, the more domain expertise regarding the content of the document under review is needed.

*The manager*

The manager is involved in the reviews as he or she decides on the execution of reviews, allocates time in project schedules and determines whether review process objectives have been met. The manager will also take care of any review training requested by the participants. Of course a manager can also be involved in the review itself depending on his or her background, playing the role of a reviewer if this would be helpful.

**3.2.3 Types of review**

A single document may be the subject of more than one review. If more than one type of review is used, the order may vary. For example, an informal review may be carried out before a technical review, or an inspection may be carried out on a requirements specification before a walkthrough with customers. It is apparent that none of the following types of review is the 'winner', but the dif

ferent types serve different purposes at different stages in the life cycle of a document.

The main review types, their main characteristics and common objectives are described below.

*Walkthrough*

**A walkthrough** is characterized by the author of the document under review guiding the participants through the document and his or her thought processes, to achieve a common understanding and to gather feedback. This is especially useful if people from outside the software discipline are present, who are not used to, or cannot easily understand software development documents. The content of the document is explained step by step by the author, to reach consensus on changes or to gather information.

Within a walkthrough the author does most of the preparation. The partici pants, who are selected from different departments and backgrounds, are not required to do a detailed study of the documents in advance. Because of the way the meeting is structured, a large number of people can participate and this larger audience can bring a great number of diverse viewpoints regarding the contents of the document being reviewed as well as serving an educational purpose. If the audience represents a broad cross-section of skills and disci plines, it can give assurance that no major defects are 'missed' in the walk through. A walkthrough is especially useful for higher-level documents, such as requirement specifications and architectural documents.

The specific goals of a walkthrough depend on its role in the creation of the document. In general the following goals can be applicable:

• to present the document to stakeholders both within and outside the soft ware discipline, in order to gather information regarding the topic under documentation;

• to explain (knowledge transfer) and evaluate the contents of the docu ment;

• to establish a common understanding of the document;

• to examine and discuss the validity of proposed solutions and the viability of alternatives, establishing consensus.

Key characteristics of walkthroughs are:

• The meeting is led by the authors; often a separate scribe is present. • Scenarios and dry runs may be used to validate the content. • Separate pre-meeting preparation for reviewers is optional.

*Technical review*

A **technical review** is a discussion meeting that focuses on achieving con sensus about the technical content of a document. Compared to inspec tions, technical reviews are less formal and there is little or no focus on defect identification on the basis of referenced documents, intended read ership and rules. During technical reviews defects are found by experts, who focus on the content of the document. The experts that are needed for a technical review are, for example, architects, chief designers and key users. In practice, technical reviews vary from quite informal to very formal.

The goals of a technical review are to:

• assess the value of technical concepts and alternatives in the product and project environment;

• establish consistency in the use and representation of technical concepts; • ensure, at an early stage, that technical concepts are used correctly; • inform participants of the technical content of the document. Key characteristics of a technical review are:

• It is a documented defect-detection process that involves peers and technical experts.

• It is often performed as a peer review without management partici pation.

• Ideally it is led by a trained moderator, but possibly also by a technical expert.

• A separate preparation is carried out during which the product is examined and the defects are found.

• More formal characteristics such as the use of checklists and a logging list or issue log are optional.

*Inspection*

**Inspection** is the most formal review type. The document under inspection is prepared and checked thoroughly by the reviewers before the meeting, compar ing the work product with its sources and other referenced documents, and using rules and checklists. In the inspection meeting the defects found are logged and any discussion is postponed until the discussion phase. This makes the inspection meeting a very efficient meeting.

The reason for carrying out inspections can be explained by using Weinberg's concept of egoless engineering [Weinberg, 1971]. Weinberg refers to the human tendency to self-justify actions. Since we tend not to see evidence that conflicts with our strong beliefs, our ability to find errors in our own work is impaired. Because of this tendency, many engineering organizations have established independent test groups that specialize in finding defects. Similar principles have led to the introduction of inspections and reviews in general.

Depending on the organization and the objectives of a project, inspections can be balanced to serve a number of goals. For example, if the time to market is extremely important, the emphasis in inspections will be on efficiency. In a safety-critical market, the focus will be on effectiveness.

The generally accepted goals of inspection are to:

• help the author to improve the quality of the document under inspection; • remove defects efficiently, as early as possible;

• improve product quality, by producing documents with a higher level of quality;

• create a common understanding by exchanging information among the inspection participants;

• train new employees in the organization's development process; • learn from defects found and improve processes in order to prevent recur rence of similar defects;

• sample a few pages or sections from a larger document in order to measure the typical quality of the document, leading to improved work by individuals in the future, and to process improvements.

Key characteristics of an inspection are:

• It is usually led by a trained moderator (certainly not by the author). • It uses defined roles during the process.

• It involves peers to examine the product.

• Rules and checklists are used during the preparation phase. • A separate preparation is carried out during which the product is examined and the defects are found.

• The defects found are documented in a logging list or issue log. • A formal follow-up is carried out by the moderator applying exit criteria. • Optionally, a causal analysis step is introduced to address process improve ment issues and learn from the defects found.

• Metrics are gathered and analyzed to optimize the process.

**3.2.4 Success factors for reviews**

Implementing (formal) reviews is not easy as there is no one way to success and there are numerous ways to fail. The next list contains a number of critical success factors that improve the chances of success when implementing reviews. It aims to answer the question, 'How do you start (formal) reviews?'.

*Find a 'champion'*

*A* champion is needed, one who will lead the process on a project or organiza tional level. They need expertise, enthusiasm and a practical mindset in order to guide moderators and participants. The authority of this champion should be clear to the entire organization. Management support is also essential for success. They should, amongst other things, incorporate adequate time for review activities in project schedules.

*Pick things that really count*

Select the documents for review that are most important in a project. Reviewing highly critical, upstream documents like requirements and architec ture will most certainly show the benefits of the review process to the project. These invested review hours will have a clear and high return on investment. In addition make sure each review has a clear objective and the correct type of review is selected that matches the defined objective. Don't try and review everything by inspection; fit the review to the risk associated with the docu ment. Some documents may only warrant an informal review and others will repay using inspection. Of course it is also of utmost importance that the right people are involved.

*Explicitly plan and track review activities*

To ensure that reviews become part of the day-to-day activities, the hours to be spent should be made visible within each project plan. The engineers involved are prompted to schedule time for preparation and, very impor tantly, rework. Tracking these hours will improve planning of the next review. As stated earlier, management plays an important part in planning of review activities.

*Train participants*

It is important that training is provided in review techniques, especially the more formal techniques, such as inspection. Otherwise the process is likely to be impeded by those who don't understand the process and the reasoning behind it. Special training should be provided to the moderators to prepare them for their critical role in the review process.

*Manage people issues*

Reviews are about evaluating someone's document. Some reviews tend to get too personal when they are not well managed by the moderator. People issues and psychological aspects should be dealt with by the moderator and should be part of the review training, thus making the review a positive experience for the author. During the review, defects should be welcomed and expressed objectively.

*Follow the rules but keep it simple*

Follow all the formal rules until you know why and how to modify them, but make the process only as formal as the project culture or maturity level allows. Do not become too theoretical or too detailed. Checklists and roles are recom mended to increase the effectiveness of defect identification.

*Continuously improve process and tools*

Continuous improvement of process and supporting tools (e.g. checklists), based upon the ideas of participants, ensures the motivation of the engineers involved. Motivation is the key to a successful change process. There should also be an emphasis, in addition to defect finding, on learning and process improvement.

*Report results*

Report quantified results and benefits to all those involved as soon as possible, and discuss the consequences of defects if they had not been found this early. Costs should of course be tracked, but benefits, especially when problems don't occur in the future, should be made visible by quantifying the benefits as well as the costs.

*Just do it!*

The process is simple but not easy. Each step of the process is clear, but expe rience is needed to execute them correctly. So, try to get experienced people to observe and help where possible. But most importantly, start doing reviews and start learning from every review.

**3.3 STATIC ANALYSIS BY TOOLS**

**1 Describe the objective of static analysis and compare it to dynamic**

**testing. (K2)**

**2 Recall typical defects identified by static analysis and compare them to**

**reviews and dynamic testing. (Kl)**

**3 List typical benefits of static analysts. (Kl)**

**4 List typical code and design defects that may be identified by static**

**analysis tools. (Kl)**

There is much to be done examining software work products without actually running the system. For example, we saw in the previous section that we can carefully review requirements, designs, code, test plans and more, to find defects and fix them before we deliver a product to a customer. In this section, we focus on a different kind of static testing, where we carefully examine requirements, designs and code, usually with automated assistance to ferret out

additional defects before the code is actually run. Thus, what is called **static analysis** is just another form of testing.

Static analysis is an examination of requirements, design and code that differs from more traditional dynamic testing in a number of important ways:

• Static analysis is performed on requirements, design or code without actually executing the software artifact being examined.

• Static analysis is ideally performed before the types of formal review dis cussed in Section 3.2.

• Static analysis is unrelated to dynamic properties of the requirements, design and code, such as test coverage.

• The goal of static analysis is to find defects, whether or not they may cause failures. As with reviews, static analysis finds defects rather than failures.

For static analysis there are many tools, and most of them focus on soft ware code. Static analysis tools are typically used by developers before, and sometimes during, component and integration testing and by designers during software modeling. The tools can show not only structural attributes (code metrics), such as depth of nesting or cyclomatic number and check against coding standards, but also graphic depictions of control flow, data relationships and the number of distinct paths from one line of code to another. Even the **compiler** can be considered a static analysis tool, since it builds a symbol table, points out incorrect usage and checks for non-compli

ance to coding language conventions (syntax).

One of the reasons for using static analysis (coding standards and the like) is related to the characteristics of the programming languages themselves. One may think that the languages are safe to use, because at least the stan dards committee knows where the problems are. But this would be wrong. Adding to the holes left by the standardization process, programmers con tinue to report features of the language, which though well-defined, lead to recognizable fault modes. By the end of the 1990s, approximately 700 of these additional problems had been identified in standard C. It is now clear that such fault modes exist. It can be demonstrated that they frequently escape the scrutiny of conventional dynamic testing, ending up in commer cial products. These problems can be found by using static analysis tools to detect them. In fact, many of the 700 fault modes reported in C can be detected in this way! In a typical C program, there is an average of approxi mately eight such faults per 1000 lines of source code; they are embedded in the released code, just waiting to cause the code to fail [Hatton, 1997]. Dynamic testing simply did not detect them. C is not the culprit here; this exercise can be carried out for other languages with broadly the same results. All programming languages have problems and programmers cannot assume that they are protected against them. And nothing in the current interna tional process of standardizing languages will prevent this from happening in the future.

The various features of static analysis tools are discussed below, with a special focus toward static code analysis tools since these are the most common in day-to-day practice. Note that static analysis tools analyze software code, as well as generated output such as HTML and XML.

3.3.1 Coding standards

Checking for adherence to coding standards is certainly the most well-known of all features. The first action to be taken is to define or adopt a coding stan dard. Usually a coding standard consists of a set of programming rules (e.g. 'Always check boundaries on an array when copying to that array'), naming conventions (e.g. 'Classes should start with capital C) and layout specifica tions (e.g. 'Indent 4 spaces'). It is recommended that existing standards are adopted. The main advantage of this is that it saves a lot of effort. An extra reason for adopting this approach is that if you take a well-known coding stan dard there will probably be checking tools available that support this standard. It can even be put the other way around: purchase a static code analyzer and declare (a selection of) the rules in it as your coding standard. Without such tools, the enforcement of a coding standard in an organization is likely to fail. There are three main causes for this: the number of rules in a coding standard is usually so large that nobody can remember them all; some context-sensitive rules that demand reviews of several files are very hard to check by human beings; and if people spend time checking coding standards in reviews, that will distract them from other defects they might otherwise find, making the review process less effective.

3.3.2 Code metrics

As stated, when performing static code analysis, usually information is calculated about structural attributes of the code, such as comment fre quency, depth of nesting, cyclomatic number and number of lines of code. This information can be computed not only as the design and code are being created but also as changes are made to a system, to see if the design or code is becoming bigger, more complex and more difficult to understand and maintain. The measurements also help us to decide among several design alternatives, especially when redesigning portions of existing code.

There are many different kinds of structural measures, each of which tells us something about the effort required to write the code in the first place, to understand the code when making a change, or to test the code using particu lar tools or techniques.

Experienced programmers know that 20% of the code will cause 80% of the problems, and complexity analysis helps to find that all-important 20%, which relate back to the principle on defect clustering as explained in Chapter 1. **Complexity** metrics identify high risk, complex areas.

The **cyclomatic complexity** metric is based on the number of decisions in a program. It is important to testers because it provides an indication of the amount of testing (including reviews) necessary to practically avoid defects. In other words, areas of code identified as more complex are can

didates for reviews and additional dynamic tests. While there are many ways to calculate cyclomatic complexity, the easiest way is to sum the number of binary decision statements (e.g. if, while, for, etc.) and add 1 to it. A more formal definition regarding the calculation rules is provided in the glossary.

Below is a simple program as an

example:

IF A = 354

THEN IF B > C

THEN A = B

ELSEA= C

ENDIF

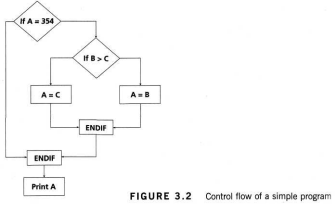
ENDIF

Print A

The **control flow** generated from the program would look like Figure 3.2.

The control flow shows seven nodes (shapes) and eight edges (lines), thus using the formal formula the cyclomatic complexity is 8-7 + 2 = 3. In this case there is no graph called or subroutine. Alternatively one may calculate the cyclomatic complexity using

the decision points rule. Since there are two decision points, the cyclomatic complexity is 2 + 1 = 3.

3.3.3 Code structure

There are many different kinds of structural measures, each of which tells us something about the effort required to write the code in the first place, to understand the code when making a change, or to test the code using particular tools or techniques. It is often assumed that

a large module takes longer to specify, design, code and test than a smaller one. But in fact the code's structure plays a big part. There are several aspects of code structure to consider:

• control flow structure;

• data flow structure;

• data structure.

The control flow structure addresses the sequence in which the instructions are executed. This aspect of structure reflects the iterations and loops in a program's design. If only the size of a program is measured, no information is provided on how often an instruction is executed as it is run. Control flow analysis can also be used to identify unreachable (dead) code. In fact many of the code metrics relate to the control flow structure, e.g. number of nested levels or cyclomatic complexity.

Data flow structure follows the trail of a data item as it is accessed and mod ified by the code. Many times, the transactions applied to data are more complex than the instructions that implement them. Thus, using data flow measures it is shown how the data act as they are transformed by the program. Defects can be found such as referencing a variable with an undefined value and variables that are never used.

Data structure refers to the organization of the data itself, independent of the program. When data is arranged as a list, queue, stack, or other well-defined structure, the algorithms for creating, modifying or deleting them are more likely to be well-defined, too. Thus, the data structure provides a lot of informa

tion about the difficulty in writing programs to handle the data and in designing test cases to show program correctness. That is, sometimes a program is complex because it has a complex data structure, rather than because of complex control or data flow.

The important thing for the tester is to be aware that the above mentioned static analysis measures can be used as early warning signals of how good the code is likely to be when it is finished.

In summary the value of static analysis is especially for:

• early detection of defects prior to test execution;

• early warning about suspicious aspects of the code, design or requirements; • identification of defects not easily found in dynamic testing; • improved maintainability of code and design since engineers work according to documented standards and rules;

• prevention of defects, provided that engineers are willing to learn from their errors and continuous improvement is practised.

**CHAPTER REVIEW**

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

From Section 3.1, you should be able to explain the importance and advan tages of static testing. You should know the difference between static testing and dynamic testing, and also understand the concept of reviews. You should be able to recognize the software work products that can be examined by static testing. You should know the glossary terms **static testing, dynamic testing** and **reviews.**

From Section 3.2, you should understand the difference between formal and informal reviews. You should be able to recall the main phases of a typical formal review. The main roles within reviews and their responsibilities should be clear to you. You should know the differences between the various types of formal review: technical review, walkthrough and inspection. Finally you should be able to explain the factors for successful performance of reviews. You should know the glossary terms **entry criteria, exit criteria, formal review, informal review, inspection, moderator, reviewer, scribe, technical review** and **walkthrough.**

From Section 3.3, you should understand the objective of static analysis and be able to compare it to static and dynamic testing. You should be able to describe the main features of static analysis and recall typical defects that can be found using static analysis. Finally, you should be able to recall the benefits of using static analysis. You should know the glossary terms **compiler, cyclo matic complexity, control flow, data flow** and **static analysis.**

**SAMPLE EXAM QUESTIONS**

Question 1 Which of the following artifacts can be examined by using review techniques?

a. Software code

b. Requirements specification

c. Test designs

d. All of the above

Question 2 Which statement about the function of a static analysis tool is true? a. Gives quality information about the code without executing it.

b. Checks expected results against actual results.

c. Can detect memory leaks.

d. Gives information about what code has and has not been exercised.

Question 3 Which is not a type of review?

a. Walkthrough

b. Inspection

c. Informal review

d. Management approval

Question 4 What statement about reviews is true?

a. Inspections are led by a trained moderator,

whereas technical reviews are not necessarily.

b. Technical reviews are led by a trained leader, inspections are not.

c. In a walkthrough, the author does not attend.

d. Participants for a walkthrough always need to be thoroughly trained.

Question 5 What is the main difference between a walkthrough and an

inspection?

a. An inspection is led by the authors, whilst a walk

through is led by a trained moderator. b. An inspection has a trained leader, whilst a walk through has no leader. c. Authors are not present during inspections, whilst

they are during walkthroughs.

d. A walkthrough is led by the author, whilst an

inspection is led by a trained

moderator.

Question 6 Which of the following characteristics and types of review processes belong together?

1. Led by the author

2. Undocumented

3. No management participation 4. Led by a trained moderator or leader 5. Uses entry and exit criteria

s. Inspection

t. Technical review

u. Informal review

v. Walkthrough

a. s = 4, t = 3, u = 2 and 5, v = 1 b. s = 4 and 5, t = 3, u = 2, v = 1 c. s = 1 and 5, t = 3, u = 2, v = 4 d. s = 5, t = 4, u = 3, v = 1 and 2

Question 7 What statement about static analysis is true?

a. With static analysis, defects can be found that are

difficult to find with dynamic testing. b. Compiling is not a form of static analysis.

c. When properly performed, static analysis makes

functional testing redundant.

d. Static analysis finds all faults.

Question 8 Which of the following statements about early test design are true and which are false?

1. Defects found during early test design are more expensive to fix.

2. Early test design can find defects. 3. Early test design can cause changes to the requirements.

4. Early test design takes more effort. a. 1 and 3 are true. 2 and 4 are false. b. 2 is true. 1, 3 and 4 are false.

c. 2 and 3 are true. 1 and 4 are false. d. 2, 3 and 4 are true. 1 is false.

Question 9 Static code analysis typically identifies all but one of the following problems. Which is it?

a. Unreachable code

b. Undeclared variables

c. Faults in the requirements

d. Too few comments

CHAPTER FOUR

**Test design techniques**

C

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

**4.1 IDENTIFYING TEST CONDITIONS AND DESIGNING TEST CASES**

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

**2 Compare the terms test condition, test case and test procedure. (K2) 3 Write test cases: (K3)**

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

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

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

**4.1.1 Introduction**

Before we can actually execute a test, we need to know what we are trying to test, the inputs, the results that should be produced by those inputs, and how we actually get ready for and run the tests. In this section we are looking at three things: test conditions, test cases and test procedures (or scripts) - they are described in the sections below. Each is specified in its own document, according to the Test Documentation Standard [IEEE829].

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

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

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

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

4.1.2 Formality of test documentation

Testing may be performed with varying degrees of formality. Very formal testing would have extensive documentation which is well controlled, and would expect the documented detail of the tests to include the exact and specific input and expected outcome of the test. Very informal testing may have no documenta

tion at all, or only notes kept by individual testers, but we'd still expect the testers to have in their minds and notes some idea of what they intended to test and what they expected the outcome to be. Most people are probably some where in between! The right level of formality for you depends on your context: a commercial safety-critical application has very different needs than a one-off

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

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

4.1.3 Test analysis: identifying test conditions

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

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

For example, if we are testing a customer management and marketing system for a mobile phone company, we might have test conditions that are related to a marketing campaign, such as age of customer (pre-teen, teenager, young adult, mature), gender, postcode or zip code, and purchasing preference (pay

as-you-go or contract). A particular advertising campaign could be aimed at male teenaged customers in the mid-west of the USA on pay-as-you-go, for example.

Testing experts use different names to represent the basic idea of 'a list of things that we could test'. For example, Marick refers to 'test requirements' as things that should be tested. Although it is not intended to imply that everything must be tested, it is too easily interpreted in that way. [Marick, 1994] In con

trast, Hutcheson talks about the 'test inventory' as a list of things that could be tested [Hutcheson, 2003]; Craig talks about 'test objectives' as broad categories of things to test and 'test inventories' as the actual list of things that need to be tested [Craig, 2002]. These authors are all referring to what the ISTQB glossary calls a test condition.

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

In Chapter 1 we explained that testing everything is known as exhaustive testing (defined as exercising every combination of inputs and preconditions) and we demonstrated that this is an impractical goal. Therefore, as we cannot test everything, we have to select a subset of all possible tests. In practice the subset we select may be a very small subset and yet it has to have a high proba

bility of finding most of the defects in a system. We need some intelligent thought processes to guide our selection; **test techniques** (i.e. test design tech niques) are such thought processes.

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

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

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

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

Why is traceability important? Consider these examples: