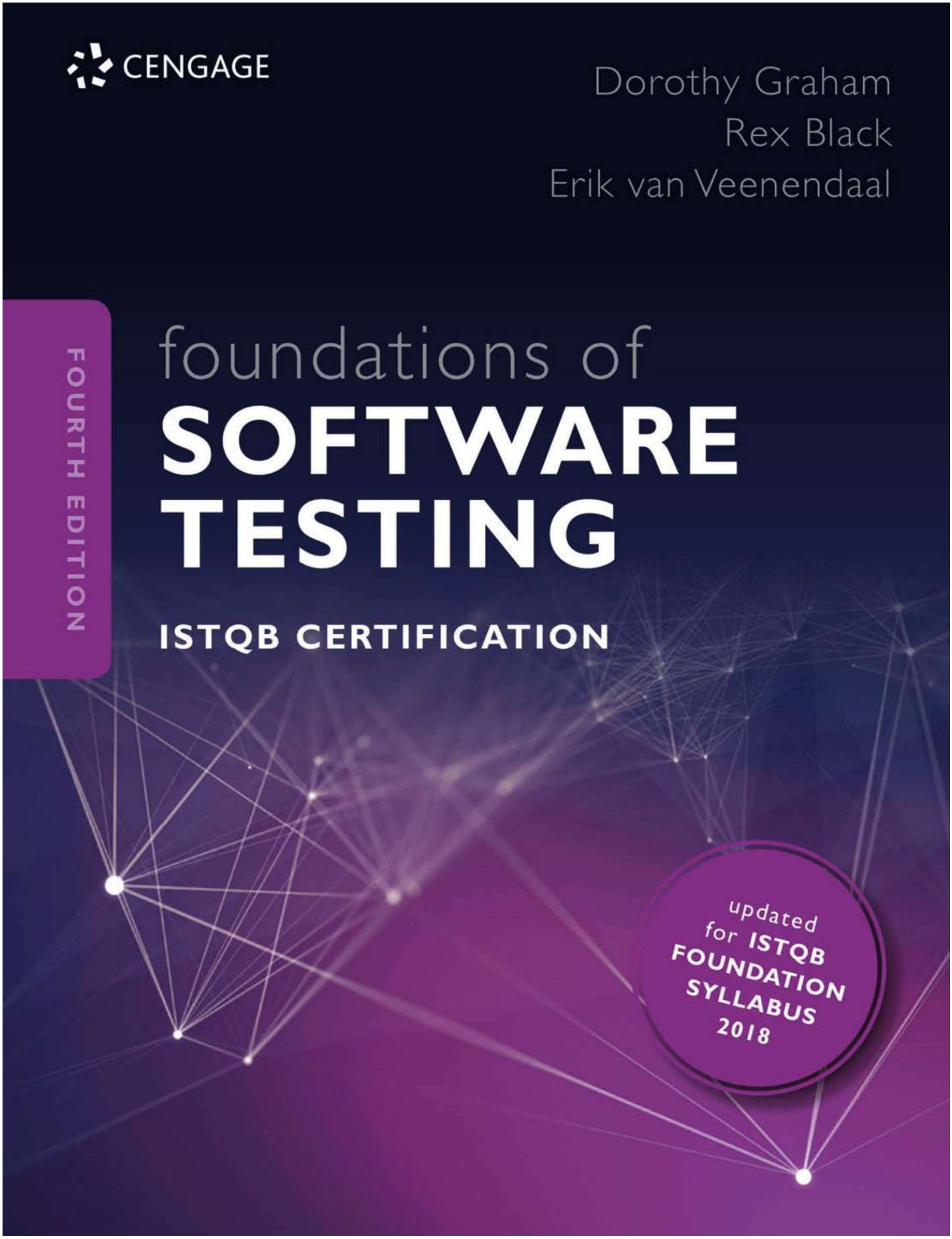
CENGAGE
Dorothy Graham
Rex Black
Erik van Veenendaal
FOURTH EDITION
foundations of
SOFTWARE
TESTING
ISTQB CERTIFICATION
updated
for ISTQB
FOUNDATION
SYLLABUS
2018

# FOUNDATIONS OF SOFTWARE TESTING

## ISTQB CERTIFICATION

## FOURTH EDITION

**Dorothy Graham**

**Rex Black**

**Erik van Veenendaal**

CENGAGE

Australia • Brazil • Mexico • Singapore • United Kingdom • United States

***Foundations of Software Testing: ISTQB Certification,* 4th Edition**
**Dorothy Graham, Rex Black, Erik van Veenendaal**

Publisher: Annabel Ainscow

List Manager: Virginia Thorp

Marketing Manager: Anna Reading

Senior Content Project Manager: Melissa Beavis

Manufacturing Buyer: Elaine Bevan

Typesetter: SPi Global

Text Design: SPi Global

Cover Design: Jonathan Bargus

Cover Image(s): © dem10/iStock/Getty Images



*British Library Cataloguing-in-Publication Data*
A catalogue record for this book is available from the British Library.

ISBN: 978-1-4737-6479-8

**Cengage Learning, EMEA**
Cheriton House, North Way,
Andover, Hampshire, SP10 5BE
United Kingdom

Cengage Learning is a leading provider of customized learning solutions with employees residing in nearly 40 different countries and sales in more than 125 countries around the world. Find your local representative at: **www.cengage.co.uk.**

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

To learn more about Cengage platforms and services, register or access your online learning solution, or purchase materials for your course, visit **www.cengage.com.**

Printed in the United Kingdom by CPI, Antony Rowe
Print Number: 01 Print Year: 2019

# CONTENTS

# FIGURES AND TABLES

# ACKNOWLEDGEMENTS

The materials in this book are based on the ISTQB Foundation Syllabus 2018. The Foundation Syllabus is copyrighted to the ISTQB (International Software Testing Qualification Board). Permission has been granted by the ISTQB to the authors to use these materials as the basis of a book, provided that recognition of authorship and copyright of the Syllabus itself is given.

The ISTQB Glossary of Testing Terms, released as version 3.2 by the ISTQB in 2018 is used as the source of definitions in this book.

The co-authors would like to thank Dorothy Graham for her effort in updating this book to be fully aligned with the 2018 version of the ISTQB Foundation Syllabus and version 3.2 of the ISTQB Glossary.

Be aware that there are some defects in this book! The Syllabus, Glossary and this book were written by people – and people make mistakes. Just as with testing, we have applied reviews and tried to identify as many defects as we could, but we also needed to release the manuscript to the publisher. Please let us know of defects that you find in our book so that we can correct them in future printings.

The authors wish to acknowledge the contribution of Isabel Evans to a previous edition of this book. We also acknowledge contributions to this edition from Gerard Bargh, Mark Fewster, Graham Freeburn, Tim Fretwell, Gary Rueda Sandoval, Melissa Tondi, Nathalie van Delft, Seretta Gamba, and Tebogo Makaba.

Dorothy Graham, Macclesfield, UK
Rex Black, Texas, USA
Erik van Veenendaal, Hato, Bonaire
2019

# PREFACE

The purpose of this book is to support the ISTQB Foundation Syllabus 2018, which is the basis for the International Foundation Certificate in Software Testing. The authors have been involved in helping to establish this qualification, donating their time and energy to the Syllabus, terminology Glossary and the International Software Testing Qualifications Board (ISTQB).

The authors of this book are all passionate about software testing. All have been involved in this area for most or all of their working lives, and have contributed to the field through practical work, training courses and books. They have written this book to help to promote the discipline of software testing.

The initial idea for this collaboration came from Erik van Veenendaal, author of *The Testing Practitioner*, a book to support the ISEB Software Testing Practitioner Certificate. The other authors agreed to work together as equals on this book. Please note that the order of the authors' names does not indicate any seniority of authorship, but simply which author was the last to update the book as the Foundation Syllabus evolved.

We intend that this book will increase your chances of passing the Foundation Certificate exam. If you are taking a course (or class) to prepare for the exam, this book will give you detailed and additional background about the topics you have covered. If you are studying for the exam on your own, this book will help you be more prepared. This book will give you information about the topics covered in the Syllabus, as well as worked exercises and practice exam questions (including a full 40-question mock exam paper in Chapter 7).

This book is a useful reference work about software testing in general, even if you are not interested in the exam. The Foundation Certificate represents a distilling of the essential aspects of software testing at the time of writing (2019), and this book will give you a good grounding in software testing.

## ISTQB AND CERTIFICATION

ISTQB stands for International Software Testing Qualifications Board and is an organization consisting of software testing professionals from each of the countries who are members of the ISTQB. Each representative is a member of a Software Testing Board in their own country. The purpose of the ISTQB is to provide internationally accepted and consistent qualifications in software testing. ISTQB sets the Syllabus and gives guidelines for each member country to implement the qualification in their own country. The Foundation Certificate is the first internationally accepted qualification in software testing and its Syllabus forms the basis of this book.

From the first qualification in 1998 until the end of 2018, around 700,000 people have taken the Foundation Certificate exam administered by a National Board of the ISTQB, or by an Exam Board contracted to a National Board. This represents 86% of all ISTQB certifications. All ISTQB National Boards and Exam Boards recognize each other's Foundation Certificates as valid.

The ISTQB qualification is independent of any individual training provider. Any training organization can offer a course based on this publicly available Syllabus. However, the National Boards associated with ISTQB give special approval to organizations that meet their requirements for the quality of the training. Such organizations are accredited and are allowed to have an invigilator or proctor from an authorized National Board or Exam Board to give the exam as part of the accredited course. The exam is also available independently from accrediting organizations or National Boards.

Why is certification of testers important? The objectives of the qualification are listed in the Syllabus. They include:

- Recognition for testing as an essential and professional software engineering specialization.
- Enabling professionally qualified testers to be recognized by employers, customers and peers.
- Raising the profile of testers.
- Promoting consistent and good testing practices within all software engineering disciplines internationally, for reasons of opportunity, communication and sharing of knowledge and resources internationally.

## FINDING YOUR WAY AROUND THIS BOOK

This book is divided into seven chapters. The first six chapters of the book each cover one chapter of the Syllabus, and each has some practice exam questions.

Chapter 1 is the start of understanding. We'll look at some fundamental questions: what is testing and why is it necessary? We'll examine why testing is not just running tests. We'll also look at why testing can damage relationships and how bridges between colleagues can be rebuilt.

In Chapter 2, we'll concentrate on testing in relation to the common software development models, including iterative and waterfall models. We'll see that different types of testing are used at different stages in the software development life cycle.

In Chapter 3, we'll concentrate on test techniques that can be used early in the software development life cycle. These include reviews and static analysis: tests done before compiling the code.

Chapter 4 covers test techniques. We'll show you techniques including equivalence partitioning, boundary value analysis, decision tables, state transition testing, use case testing, statement and decision coverage and experience-based techniques. This chapter is about how to become a better tester in terms of designing tests. There are exercises for the most significant techniques included in this chapter.

Chapter 5 is about the management and control of testing, including estimation, risk assessment, defect management and reporting. Writing a good defect report is a key skill for a good tester, so we have an exercise for that too.

In Chapter 6, we'll show you how tools support all the activities in the test process, and how to select and implement tools for the greatest benefit.

Chapter 7 contains general advice about taking the exam and has the full 40-question mock paper. This is a key learning aid to help you pass the real exam.

The appendices of the book include a full list of references and a copy of the ISTQB testing terminology Glossary, as well as the answers to all the practice exam questions.

## TO HELP YOU USE THE BOOK

1 **Get a copy of the Syllabus:** You should download the Syllabus from the ISTQB website so that you have the current version, and so that you can check off the Syllabus objectives as you learn. This is available at https://www.istqb.org/downloads/syllabi/foundation-level-syllabus.html

2 **Understand what is meant by learning objectives and knowledge levels:** In the Syllabus, you will see learning objectives and knowledge (or cognitive) levels at the start of each section of each chapter. These indicate what you need to know and the depth of knowledge required for the exam. We have used the timings in the Syllabus and knowledge levels to guide the space allocated in the book, both for the text and for the exercises. You will see the learning objectives and knowledge levels at the start of each section within each chapter. The knowledge levels expected by the Syllabus are:

- **K1: remember, recognize, recall:** you will recognize, remember and recall a term or concept. For example, you could recognize one definition of failure as 'Non-delivery of service to an end user or any other stakeholder'.
- **K2: understand, explain, give reasons, compare, classify, summarize:** you can select the reasons or explanations for statements related to the topic, and can summarize, compare, classify and give examples for the testing concept. For example, you could explain that one reason why tests should be designed as early as possible is to find defects when they are cheaper to remove.
- **K3: apply:** you can select the correct application of a concept or technique and apply it to a given context. For example, you could identify boundary values for valid and invalid partitions, and you could select test cases from a given state transition diagram in order to cover all transitions.

Remember, as you go through the book, if a topic has a learning objective marked K1 you just need to recognize it. If it has a learning objective of K3 you will be expected to apply your knowledge in the exam, for example.

3 **Use the Glossary of terms:** Each chapter of the Syllabus has a number of terms listed in it. You are expected to remember these terms at least at K1 level, even if they are not explicitly mentioned in the learning objectives. You will see a number of **definitions** throughout this book, as in the sidebar.

**Definition** A description of the meaning of a word.

All definitions of software testing terms (called keywords in the chapters) are taken from the *ISTQB Glossary* (version 3.2), which is available online at www.glossary.istqb.org. A copy of this Glossary is also at the back of the book. All the terms that are specifically mentioned in the Syllabus, that is, the ones you need to learn for the exam, are mentioned in each section of this book.

You will notice that some terms in the Glossary at the back of this book are underlined. These are terms that are mentioned specifically as keywords in the Syllabus. These are the terms that you need to be familiar with for the exam.

4 **Use the references sensibly:** We have referenced all the books used by the Syllabus authors when they constructed the Syllabus. You will see these underlined in the list at the end of the book. We also added references to some other books, papers and websites that we thought useful or which we referred to when writing. You do not need to read all referenced books for the exam! However, you may find some of them useful for further reading to increase your knowledge after the exam, and to help you apply some of the ideas you will come across in this book.

5 **Do the practice exams:** When you get to the end of a chapter (for Chapters 1 to 6), answer the exam questions, and then turn to 'Answers to the Sample Exam Questions' to check if your answers were correct. After you have completed all of the six chapters, then take the full mock exam in Chapter 7. If you would like the most realistic exam conditions, then allow yourself just an hour to take the exam in Chapter 7. Also take the free sample exams from the ISTQB web site. You can download both the exam and the answers including justifications for the correct (and wrong) answers.

CENGAGE

# Teaching & Learning Support Resources

**Cengage's peer reviewed content for higher and further education courses is accompanied by a range of digital teaching and learning support resources. The resources are carefully tailored to the specific needs of the instructor, student and the course.**

A password protected area for instructors.

An open-access area for students.

Lecturers: to discover the dedicated teaching digital support resources accompanying this textbook please register here for access:
**cengage.com/dashboard/#login**

Students: to discover the dedicated Learning digital support resources accompanying this textbook, please search for Foundations of Software Testing: ISTQB Certification, Fourth Edition on: **cengage.com**

BE UNSTOPPABLE!

Learn more at **cengage.com**

CHAPTER ONE

# Fundamentals of testing

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

Note that the learning objectives start with 'FL' rather than 'LO' to show that they are learning objectives for the Foundation Level qualification.

## 1.1 WHAT IS TESTING?

**SYLLABUS LEARNING OBJECTIVES FOR 1.1 WHAT IS TESTING? (K2)**

**FL-1.1.1 Identify typical objectives of testing (K1)**

**FL-1.1.2 Differentiate testing from debugging (K2)**

In this section, we will kick off the book by looking at what testing is, some misconceptions about testing, the typical objectives of testing and the difference between testing and debugging.

Within each section of this book, there are terms that are important – they are used in the section (and may be used elsewhere as well). They are listed in the Syllabus as keywords, which means that you need to know the definition of the term and it could appear in an exam question. We will give the definition of the relevant keyword terms in the margin of the text, and they can also be found in the Glossary (including the ISTQB online Glossary). We also show the keyword in **bold** within the section or subsection where it is defined and discussed.

In this section, the relevant keyword terms are **debugging**, **test object**, **test objective**, **testing**, **validation** and **verification**.

### *Software is everywhere*

The last 100 years have seen an amazing human triumph of technology. Diseases that once killed and paralyzed are routinely treated or prevented – or even eradicated entirely, as with smallpox. Some children who stood amazed as they watched the first gasoline-powered automobile in their town are alive today, having seen people walk on the moon, an event that happened before a large percentage of today's workforce was even born.

Perhaps the most dramatic advances in technology have occurred in the arena of information technology. Software systems, in the sense that we know them, are a recent innovation, less than 70 years old, but have already transformed daily life around the world. Thomas Watson, the one-time head of IBM, famously predicted that only about five computers would be needed in the whole world. This vastly inaccurate prediction was based on the idea that information technology was useful only for business and government applications, such as banking, insurance and conducting a census. (The Hollerith punch-cards used by computers at the time Watson made his prediction were developed for the United States census.) Now, everyone who drives a car is using a machine not only designed with the help of computers, but which also contains more computing power than the computers used by NASA to get Apollo missions to and from the Moon. Mobile phones are now essentially handheld computers that get smarter with every new model. The Internet of Things (IoT) now gives us the ability to see who is at our door or turn on the lights when we are nowhere near our home.

However, in the software world, the technological triumph has not been perfect. Almost every living person has been touched by information technology, and most of us have dealt with the frustration and wasted time that occurs when software fails and exhibits unexpected behaviours. Some unfortunate individuals and companies have experienced financial loss or damage to their personal or business reputations as a result of defective software. A highly unlucky few have even been injured or killed by software failures, including by self-driving cars.

One way to help overcome such problems is software testing, when it is done well. Testing covers activities throughout the life cycle and can have a number of different objectives, as we will see in Section 1.1.1.

### *Testing is more than running tests*

**Testing** The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.

An ongoing misperception, although less common these days, about **testing** is that it only involves running tests. Specifically, some people think that testing involves nothing beyond carrying out some sequence of actions on the system under test, submitting various inputs along the way and evaluating the observed results. Certainly, these activities are one element of testing – specifically, these activities make up the bulk of the test execution activities – but there are many other activities involved in the test process.

We will discuss the test process in more detail later in this chapter (in Section 1.4), but testing also includes (in addition to test execution): test planning, analyzing, designing and implementing tests, reporting test progress and results, and reporting defects. As you can see, there is a lot more to it than just running tests.

Notice that there are major test activities both before and after test execution. In addition, in the ISTQB definition of software testing, you will see that testing includes both static and dynamic testing. Static testing is any evaluation of the software or related work products (such as requirements specifications or user stories) that occurs without executing the software itself. Dynamic testing is an evaluation of that software or related work products that does involve executing the software. As such, the ISTQB definition of testing not only includes a number of pre-execution and post-execution activities that non-testers often do not consider 'testing', but also includes software quality activities (for example, requirements reviews and static analysis of code) that non-testers (and even sometimes testers) often do not consider 'testing' either.

The reason for this broad definition is that both dynamic testing (at whatever level) and static testing (of whatever type) often enable the achievement of similar project objectives. Dynamic testing and static testing also generate information that can help achieve an important process objective – that of understanding and improving the software development and testing processes. Dynamic testing and static testing are complementary activities, each able to generate information that the other cannot.

### *Testing is more than verification*

Another common misconception about testing is that it is only about checking correctness; that is, that the system corresponds to its requirements, user stories or other specifications. Checking against a specification (called **verification**) is certainly part of testing, where we are asking the question, 'Have we built the system correctly?' Note the emphasis in the definition on 'specified requirements'.

**Verification**
Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled.

But just conforming to a specification is not sufficient testing, as we will see in Section 1.3.7 (Absence-of-errors is a fallacy). We also need to test to see if the delivered software and system will meet user and stakeholder needs and expectations in its operational environment. Often it is the tester who becomes the advocate for the end-user in this kind of testing, which is called **validation**. Here we are asking the question, 'Have we built the right system?' Note the emphasis in the definition on 'intended use'.

**Validation**
Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled.

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; for example, does it provide a solution to the problem?'

## 1.1.1 Typical objectives of testing

The following are some **test objectives** given in the Foundation Syllabus:

**Test objective** A reason or purpose for designing and executing a test.

- To evaluate work products such as requirements, user stories, design and code by using static testing techniques, such as reviews.
- To verify whether all specified requirements have been fulfilled, for example, in the resulting system.
- To validate whether the test object is complete and works as the users and other stakeholders expect – for example, together with user or stakeholder groups.
- To build confidence in the level of quality of the test object, such as when those tests considered highest risk pass, and when the failures that are observed in the other tests are considered acceptable.
- To prevent defects, such as when early test activities (for example, requirements reviews or early test design) identify defects in requirements specifications that are removed before they cause defects in the design specifications and subsequently the code itself. Both reviews and test design serve as a verification and validation of these test basis documents that will reveal problems that otherwise would not surface until test execution, potentially much later in the project.

- To find failures and defects; this is typically a prime focus for software testing.
- To provide sufficient information to stakeholders to allow them to make informed decisions, especially regarding the level of quality of the **test object** – for example, by the satisfaction of entry or exit criteria.
- To reduce the level of risk of inadequate software quality (e.g. previously undetected failures occurring in operation).
- To comply with contractual, legal or regulatory requirements or standards, and/or to verify the test object's compliance with such requirements or standards.

**Test object** The component or system to be tested.

These objectives are not universal. Different test viewpoints, test levels and test stakeholders can have different objectives. While many levels of testing, such as component, integration and system testing, focus on discovering as many failures as possible in order to find and remove defects, in acceptance testing the main objective is confirmation of correct system operation (at least under normal conditions), together with building confidence that the system meets its requirements. The context of the test object and the software development life cycle will also affect what test objectives are appropriate. Let's look at some examples to illustrate this.

When evaluating a software package that might be purchased or integrated into a larger software system, the main objective of testing might be the assessment of the quality of the software. Defects found may not be fixed, but rather might support a conclusion that the software be rejected.

During component testing, one objective at this level may be to achieve a given level of code coverage by the component tests – that is, to assess how much of the code has actually been exercised by a set of tests and to add additional tests to exercise parts of the code that have not yet been covered/tested. Another objective may be to find as many failures as possible so that the underlying defects are identified and fixed as early as possible.

During user acceptance testing, one objective may be to confirm that the system works as expected (validation) and satisfies requirements (verification). Another objective of testing here is to focus on providing stakeholders with an evaluation of the risk of releasing the system at a given time. Evaluating risk can be part of a mix of objectives, or it can be an objective of a separate level of testing, as when testing a safety-critical system, for example.

During maintenance testing, our objectives often include checking whether developers have introduced any regressions (new defects not present in the previous version) while making changes. Some forms of testing, such as operational testing, focus on assessing quality characteristics such as reliability, security, performance or availability.

## 1.1.2 Testing and debugging

Let's end this section by saying what testing is not, but is often thought to be. Testing is not **debugging**. While dynamic testing often locates failures which are caused by defects, and static testing often locates defects themselves, testing does not fix defects. It is during debugging, a development activity, that a member of the project team finds, analyzes and removes the defect, the underlying cause of the failure. After debugging, there is a further testing activity associated with the defect, which is called confirmation testing. This activity ensures that the fix does indeed resolve the failure.

**Debugging** The process of finding, analyzing and removing the causes of failures in software.

In terms of roles, dynamic testing is a testing role, debugging is a development role and confirmation testing is again a testing role. However, in Agile teams, this distinction may be blurred, as testers may be involved in debugging and component testing.

Further information about software testing concepts can be found in the ISO standard ISO/IEC/IEEE 29119-1 [2013].

## 1.2 WHY IS TESTING NECESSARY?

**SYLLABUS LEARNING OBJECTIVES FOR 1.2 WHY IS TESTING NECESSARY? (K2)**

**FL-1.2.1 Give examples of why testing is necessary (K2)**

**FL-1.2.2 Describe the relationship between testing and quality assurance and give examples of how testing contributes to higher quality (K2)**

**FL-1.2.3 Distinguish between error, defect and failure (K2)**

**FL-1.2.4 Distinguish between the root cause of a defect and its effects (K2)**

In this section, we discuss how testing contributes to success and the relationship between testing and quality assurance. We will describe the difference between errors, defects and failures and illustrate how software defects or bugs can cause problems for people, the environment or a company. We will draw important distinctions between defects, their root causes and their effects.

As we go through this section, watch for the Syllabus terms **defect**, **error**, **failure**, **quality**, **quality assurance** and **root cause**.

Testing can help to reduce the risk of failures occurring during operation, provided it is carried out in a rigorous way, including reviews of documents and other work products. Testing both verifies that a system is correctly built and validates that it will meet users' and stakeholders' needs, even though no testing is ever exhaustive (see Principle 2 in Section 1.3, Exhaustive testing is impossible). In some situations, testing may not only be helpful, but may be necessary to meet contractual or legal requirements or to conform to industry-specific standards, such as automotive or safety-critical systems.

### 1.2.1 Testing's contributions to success

As we mentioned in Section 1.1, all of us have experienced software problems; for example, an app fails in the middle of doing something, a website freezes while taking your payment (did it go through or not?) or inconsistent prices for exactly the same flights on travel sites. Failures like these are annoying, but failures in safety-critical software can be life-threatening, such as in medical devices or self-driving cars.

The use of appropriate test techniques, applied with the right level of test expertise at the appropriate test levels and points in the software development life cycle, can be of significant help in identifying problems so that they can be fixed before the

software or system is released into use. Here are some examples where testing could contribute to more successful systems:

- Having testers involved in requirements reviews or user story refinement could detect defects in these work products before any design or coding is done for the functionality described. Identifying and removing defects at this stage reduces the risk of the wrong software (incorrect or untestable) being developed.
- Having testers work closely with system designers while the system is being designed can increase each party's understanding of the design and how to test it. Since misunderstandings are often the cause for defects in software, having a better understanding at this stage can reduce the risk of design defects. A bonus is that tests can be identified from the design – thinking about how to test the system at this stage often results in better design.
- Having testers work closely with developers while the code is under development can increase each party's understanding of the code and how to test it. As with design, this increased understanding, and the knowledge of how the code will be tested, can reduce the risk of defects in the code (and in the tests).
- Having testers verify and validate the software prior to release can detect failures that might otherwise have been missed – this is traditionally where the focus of testing has been. As we see with the previous examples, if we leave it until release, we will not be nearly as efficient as we would have been if we had caught these defects earlier. However, it is still necessary to test just before release, and testers can also help to support debugging activities, for example, by running confirmation and regression tests. Thus, testing can help the software meet stakeholder needs and satisfy requirements.

In addition to these examples, achieving the defined test objectives (see Section 1.1.1) also contributes to the overall success of software development and maintenance.

## 1.2.2 Quality assurance and testing

Is quality assurance (QA) the same as testing? Many people refer to 'doing QA' when they are actually doing testing, and some job titles refer to QA when they really mean testing. The two are not the same. Quality assurance is actually one part of a larger concept, quality management, which refers to all activities that direct and control an organization with regard to quality in all aspects. Quality affects not only software development but also human resources (HR) procedures, delivery processes and even the way people answer the company's telephones.

**Quality assurance** Part of quality management focused on providing confidence that quality requirements will be fulfilled.

**Quality** The degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations.

Quality management consists of a number of activities, including **quality assurance** and quality control (as well as setting quality objectives, quality planning and quality improvement). Quality assurance is associated with ensuring that a company's standard ways of performing various tasks are carried out correctly. Such procedures may be written in a quality handbook that everyone is supposed to follow. The idea is that if processes are carried out correctly, then the products produced will be of higher **quality**. Root cause analysis and retrospectives are used to help to improve processes for more effective quality assurance. If they are following a recognized quality management standard, companies may be audited to ensure that they do actually follow their prescribed processes (say what you do, and do what you say).

Quality control is concerned with the quality of products rather than processes, to ensure that they have achieved the desired level of quality. Testing is looking at work products, including software, so it is actually a quality control activity rather than a quality assurance activity, despite common usage. However, testing also has processes that should be followed correctly, so quality assurance does support good testing in this way. Sections 1.1.1 and 1.2.1 describe how testing contributes to the achievement of quality.

So, we see that testing plays an essential supporting role in delivering quality software. However, testing by itself is not sufficient. Testing should be integrated into a complete, team-wide and development process-wide set of activities for quality assurance. Proper application of standards, training of staff, the use of retrospectives to learn lessons from defects and other important elements of previous projects, rigorous and appropriate software testing: all of these activities and more should be deployed by organizations to ensure acceptable levels of quality and quality risk upon release.

## 1.2.3 Errors, defects and failures

Why does software fail? Part of the problem is that, ironically, while computerization has allowed dramatic automation of many professions, software engineering remains a human-intensive activity. And humans are fallible beings. So, software is fallible because humans are fallible.

The precise chain of events goes something like this. A developer makes an **error** (or mistake), such as forgetting about the possibility of inputting an excessively long string into a field on a screen. The developer thus puts a **defect** (or fault or bug) into the program, such as omitting a check on input strings for length prior to processing them. When the program is executed, if the right conditions exist (or the wrong conditions, depending on how you look at it), the defect may result in unexpected behaviour; that is, the system exhibits a **failure**, such as accepting an over-long input that it should reject, with subsequent corruption of other data.

**Error** (mistake) A human action that produces an incorrect result.

**Defect** (bug, fault) An imperfection or deficiency in a work product where it does not meet its requirements or specifications.

**Failure** An event in which a component or system does not perform a required function within specified limits.

Other sequences of events can result in eventual failures, too. A business analyst can introduce a defect into a requirement, which can escape into the design of the system and further escape into the code. For example, a business analyst might say that an e-commerce system should support 100 simultaneous users, but actually peak load should be 1,000 users. If that defect is not detected in a requirements review (see Chapter 3), it could escape from the requirements phase into the design and implementation of the system. Once the load exceeds 100 users, resource utilization may eventually spike to dangerous levels, leading to reduced response time and reliability problems.

A technical writer can introduce a defect into the online help screens. For example, suppose that an accounting system is supposed to multiply two numbers together, but the help screens say that the two numbers should be added. In some cases, the system will appear to work properly, such as when the two numbers are both 0 or both 2. However, most frequently the program will exhibit unexpected results (at least based on the help screens).

So, human beings are fallible and thus, when they work, they sometimes introduce defects. It is important to point out that the introduction of defects is not a purely random accident, though some defects may be introduced randomly, such as when a phone rings and distracts a systems engineer in the middle of a complex series of design decisions. The rate at which people make errors increases when they are under time pressure, when they are working with complex systems, interfaces or code, and when they are dealing with changing technologies or highly interconnected systems.

While we commonly think of failures being the result of 'bugs in the code', a significant number of defects are introduced in work products such as requirements specifications and design specifications. Capers Jones reports that about 20% of defects are introduced in requirements, and about 25% in design. The remaining 55% are introduced during implementation or repair of the code, metadata or documentation [Jones 2008]. Other experts and researchers have reached similar conclusions, with one organization finding that as many as 75% of defects originate in requirements and design. Figure 1.1 shows four typical scenarios, the upper stream being correct requirements, design and implementation, the lower three streams showing defect introduction at some phase in the software life cycle.

Ideally, defects are removed in the same phase of the life cycle in which they are introduced. (Well, ideally defects are not introduced at all, but this is not possible because, as discussed before, people are fallible.) The extent to which defects are removed in the phase of introduction is called phase containment. Phase containment is important because the cost of finding and removing a defect increases each time that defect escapes to a later life cycle phase. Multiplicative increases in cost, of the sort seen in Figure 1.2, are not unusual. The specific increases vary considerably, with Boehm reporting cost increases of 1:5 (from requirements to after release) for simple systems, to as high as 1:100 for complex systems [Boehm 1986]. If you are curious about the economics of software testing and other quality-related activities, you can see Gilb [1993], Black [2004] or Black [2009].

Defects may result in failures, or they may not, depending on inputs and other conditions. In some cases, a defect can exist that will never cause a failure in actual use, because the conditions that could cause the failure can never arise. In other cases, a defect can exist that will not cause a failure during testing, but which always results in failures in production. This can happen with security, reliability and performance defects, especially if the test environments do not closely replicate the production environment(s).

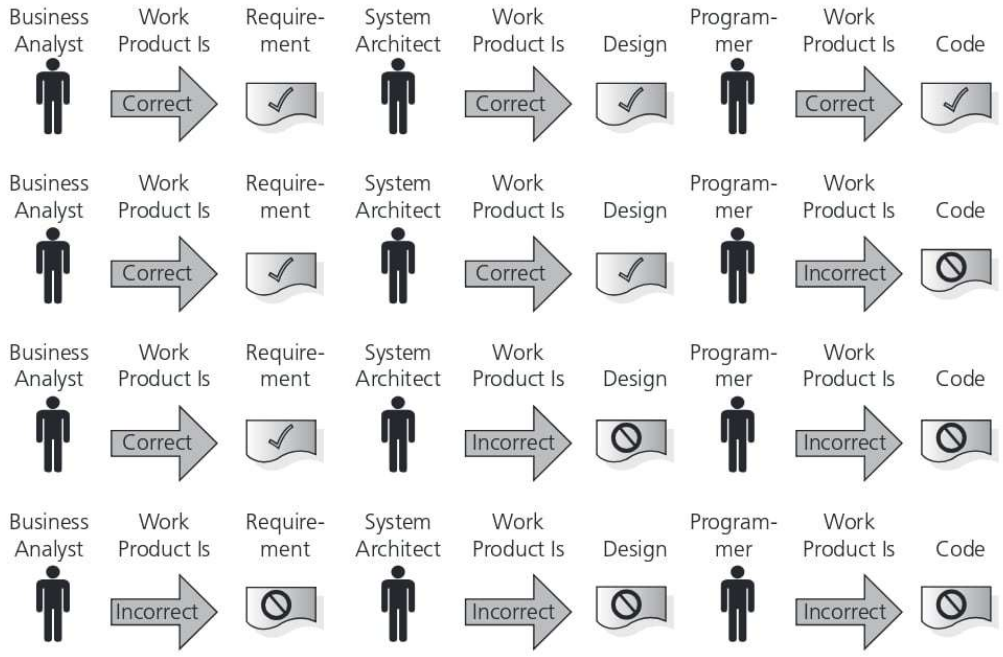


**FIGURE 1.1** Four typical scenarios

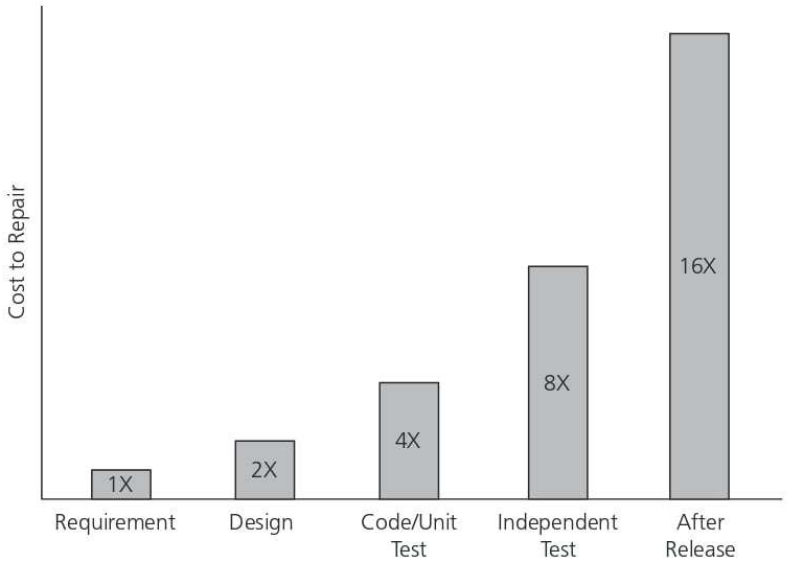


**FIGURE 1.2** Multiplicative increases in cost

It can also happen that expected and actual results do not match for reasons other than a defect. In some cases, environmental conditions can lead to unexpected results that do not relate to a software defect. Radiation, magnetism, electronic fields and pollution can damage hardware or firmware, or simply change the conditions of the hardware or firmware temporarily in a way that causes the software to fail.

## 1.2.4 Defects, root causes and effects

Testing also provides a learning opportunity that allows for improved quality if lessons are learned from each project. If root cause analysis is carried out for the defects found on each project, the team can improve its software development processes to avoid the introduction of similar defects in future systems. Through this simple process of learning from past mistakes, organizations can continuously improve the quality of their processes and their software. A **root cause** is generally an organizational issue, whereas a cause for a defect is an individual action. So, for example, if a developer puts a 'less than' instead of 'greater than' symbol, this error may have been made through carelessness, but the carelessness may have been made worse because of intense time pressure to complete the module quickly. With more time for checking his or her work, or with better review processes, the defect would not have got through to the final product. It is human nature to blame individuals when in fact organizational pressure makes errors almost inevitable.

**Root cause** A source of a defect such that if it is removed, the occurrence of the defect type is decreased or removed.

The Syllabus gives a good example of the difference between defects, root causes and effects: suppose that incorrect interest payments result in customer complaints. There is just a single line of code that is incorrect. The code was written for a user story that was ambiguous, so the developer interpreted it in a way that they thought was sensible (but it was wrong). How did the user story come to be ambiguous? In this example, the product owner misunderstood how interest was to be calculated, so was unable to clearly specify what the interest calculation should have been. This misunderstanding could lead to a lot of similar defects, due to ambiguities in other user stories as well.

The failure here is the incorrect interest calculations for customers. The defect is the wrong calculation in the code. The root cause was the product owner's lack of knowledge about how interest should be calculated, and the effect was customer complaints.

The root cause can be addressed by providing additional training in interest rate calculations to the product owner, and possibly additional reviews of user stories by interest calculation experts. If this is done, then incorrect interest calculations due to ambiguous user stories should be a thing of the past.

Root cause analysis is covered in more detail in two other ISTQB qualifications: Expert Level Test Management, and Expert Level Improving the Test Process.

## 1.3 SEVEN TESTING PRINCIPLES

### SYLLABUS LEARNING OBJECTIVES FOR 1.3 SEVEN TESTING PRINCIPLES (K2)

**FL-1.3.1 Explain the seven testing principles (K2)**

In this section, we will review seven fundamental principles of testing that have been observed over the last 40+ years. These principles, while not always understood or noticed, are in action on most if not all projects. Knowing how to spot these principles, and how to take advantage of them, will make you a better tester.

In addition to the descriptions of each principle below, you can refer to Table 1.1 for a quick reference of the principles and their text as written in the Syllabus.

### *Principle 1. Testing shows the presence of defects, not their absence*

As mentioned in the previous section, a typical objective of many testing efforts is to find defects. Many testing organizations that the authors have worked with are quite effective at doing so. One of our exceptional clients consistently finds, on average, 99.5% of the defects in the software it tests. In addition, the defects left undiscovered are less important and unlikely to happen frequently in production. Sometimes, it turns out that this test team has indeed found 100% of the defects that would matter to customers, as no previously unreported defects are reported after release. Unfortunately, this level of effectiveness is not common.

However, no test team, test technique or test strategy can guarantee to achieve 100% defect-detection percentage (DDP) – or even 95%, which is considered excellent. Thus, it is important to understand that, while testing can show that defects are present, it cannot prove that there are no defects left undiscovered. Of course, as testing continues, we reduce the likelihood of defects that remain undiscovered, but eventually a form of Zeno's paradox takes hold: each additional test run may cut the risk of a remaining defect in half, but only an infinite number of tests can cut the risk down to zero.

That said, testers should not despair or let the perfect be the enemy of the good. While testing can never prove that the software works, it can reduce the remaining level of risk to product quality to an acceptable level, as mentioned before. In any endeavour worth doing, there is some risk. Software projects – and software testing – are endeavours worth doing.

**TABLE 1.1** Testing principles

| | | |
|---|---|---|
| **Principle 1:** | **Testing shows the presence of defects, not their absence** | 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, testing is not a proof of correctness. |
| **Principle 2:** | **Exhaustive testing is impossible** | Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Rather than attempting to test exhaustively, risk analysis, test techniques and priorities should be used to focus test efforts. |
| **Principle 3:** | **Early testing saves time and money** | To find defects early, both static and dynamic test activities should be started as early as possible in the software development life cycle. Early testing is sometimes referred to as 'shift left'. Testing early in the software development life cycle helps reduce or eliminate costly changes (see Chapter 3, Section 3.1). |
| **Principle 4:** | **Defects cluster together** | A small number of modules usually contains most of the defects discovered during pre-release testing, or they are responsible for most of the operational failures. Predicted defect clusters, and the actual observed defect clusters in test or operation, are an important input into a risk analysis used to focus the test effort (as mentioned in Principle 2). |
| **Principle 5:** | **Beware of the pesticide paradox** | If the same tests are repeated over and over again, eventually these tests no longer find any new defects. To detect new defects, existing tests and test data are changed and new tests need to be written. (Tests are no longer effective at finding defects, just as pesticides are no longer effective at killing insects after a while.) In some cases, such as automated regression testing, the pesticide paradox has a beneficial outcome, which is the relatively low number of regression defects. |
| **Principle 6:** | **Testing is context dependent** | Testing is done differently in different contexts. For example, safety-critical software is tested differently from an e-commerce mobile app. As another example, testing in an Agile project is done differently to testing in a sequential life cycle project (see Chapter 2, Section 2.1). |
| **Principle 7:** | **Absence-of-errors is a fallacy** | Some organizations expect that testers can run all possible tests and find all possible defects, but Principles 2 and 1, respectively, tell us that this is impossible. Further, it is a fallacy to expect that *just* finding and fixing a large number of defects will ensure the success of a system. For example, thoroughly testing all specified requirements and fixing all defects found could still produce a system that is difficult to use, that does not fulfil the users' needs and expectations or that is inferior compared to other competing systems. |

### Principle 2. Exhaustive testing is impossible

This principle is closely related to the previous principle. For any real-sized system (anything beyond the trivial software constructed in first-year software engineering courses), the number of possible test cases is either infinite or so close to infinite as to be practically innumerable.

Infinity is a tough concept for the human brain to comprehend or accept, so let's use an example. One of our clients mentioned that they had calculated the number of possible internal data value combinations in the Unix operating system as greater than the number of known molecules in the universe by four orders of magnitude. They further calculated that, even with their fastest automated tests, just to test all of these internal state combinations would require more time than the current age of the universe. Even that would not be a complete test of the operating system; it would only cover all the possible data value combinations.

So, we are confronted with a big, infinite cloud of possible tests; we must select a subset from it. One way to select tests is to wander aimlessly in the cloud of tests, selecting at random until we run out of time. While there is a place for automated random testing, by itself it is a poor strategy. We'll discuss testing strategies further in Chapter 5, but for the moment let's look at two.

One strategy for selecting tests is risk-based testing. In risk-based testing, we have a cross-functional team of project and product stakeholders perform a special type of risk analysis. In this analysis, stakeholders identify risks to the quality of the system, and assess the level of risk (often using likelihood and impact) associated with each risk item. We focus the test effort based on the level of risk, using the level of risk to determine the appropriate number of test cases for each risk item, and also to sequence the test cases.

Another strategy for selecting tests is requirements-based testing. In requirements-based testing, testers analyze the requirements specification (which would be user stories in Agile projects) to identify test conditions. These test conditions inherit the priority of the requirement or user story they derive from. We focus the test effort based on the priority to determine the appropriate number of test cases for each aspect, and also to sequence the test cases.

### Principle 3. Early testing saves time and money

This principle tells us that we should start testing as early as possible in order to find as many defects as possible. In addition, since the cost of finding and removing a defect increases the longer that defect is in the system, early testing also means we are likely to minimize the cost of removing defects.

So, the first principle tells us that we cannot find all the bugs, but rather can only find some percentage of them. The second principle tells us that we cannot run every possible test. The third principle tells us to start testing early. What can we conclude when we put these three principles together?

Imagine that you have a system with 1,000 defects. Suppose we wait until the very end of the project and run one level of testing, system test. You find and fix 90% of the defects. That still leaves 100 defects, which presumably will escape to the customers or users.

Instead, suppose that you start testing early and continue throughout the life cycle. You perform requirements reviews, design reviews and code reviews. You perform unit testing, integration testing and system testing. Suppose that, during each test activity, you find and remove only 45% of the defects – half as effective

as the previous system test level. Nevertheless, at the end of the process, fewer than 30 defects remain. Even though each test activity was only 45% effective at finding defects, the overall sequence of activities was 97% effective. Note that now we are doing both static testing (the reviews) and dynamic testing (the running of tests at the different test levels). This approach of starting test activities as early as possible is also called 'shift left' because the test activities are no longer all done on the right-hand side of a sequential life cycle diagram, but on the left-hand side at the beginning of development. Although unit test execution is of course on the right side of a sequential life cycle diagram, improving and spending more effort on unit testing early on is a very important part of the shift left paradigm.

In addition, defects removed early cost less to remove. Further, since much of the cost in software engineering is associated with human effort, and since the size of a project team is relatively inflexible once that project is underway, reduced cost of defects also means reduced duration of the project. That situation is shown graphically in Figure 1.3.

Now, this type of cumulative and highly efficient defect removal only works if each of the test activities in the sequence is focused on different, defined objectives. If we simply test the same test conditions over and over, we will not achieve the cumulative effect, for reasons we will discuss in a moment.

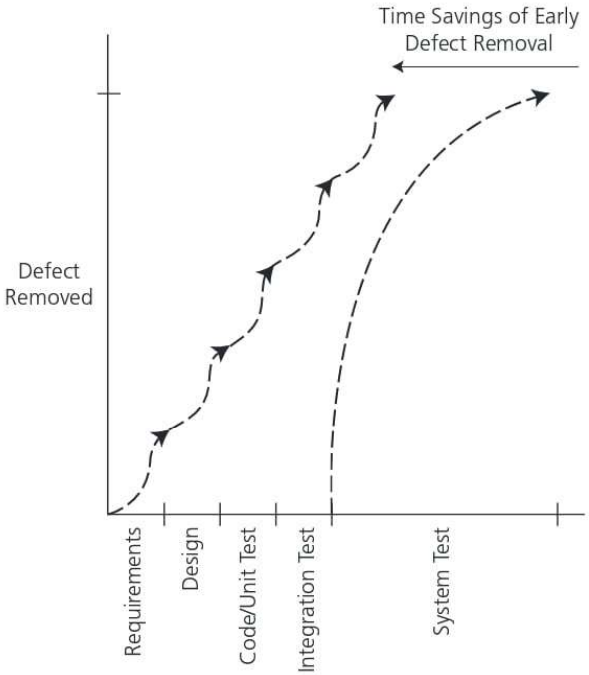


**FIGURE 1.3** Time savings of early defect removal

### *Principle 4. Defects cluster together*

This principle relates to something we discussed previously, that relying entirely on the testing strategy of a random walk in the infinite cloud of possible tests is relatively weak. Defects are not randomly and uniformly distributed throughout the software under test. Rather, defects tend to be found in clusters, with 20% (or fewer)

of the modules accounting for 80% (or more) of the defects. In other words, the defect density of modules varies considerably. While controversy exists about why defect clustering happens, the reality of defect clustering is well established. It was first demonstrated in studies performed by IBM in the 1960s [Jones 2008], and is mentioned in Myers [2011]. We continue to see evidence of defect clustering in our work with clients.

Defect clustering is helpful to us as testers, because it provides a useful guide. If we focus our test effort (at least in part) based on the expected (and ultimately observed) likelihood of finding a defect in a certain area, we can make our testing more effective and efficient, at least in terms of our objective of finding defects. Knowledge of and predictions about defect clusters are important inputs to the risk-based testing strategy discussed earlier. In a metaphorical way, we can imagine that bugs are social creatures who like to hang out together in the dark corners of the software.

### *Principle 5. Beware of the pesticide paradox*

This principle was coined by Boris Beizer [Beizer 1990]. He observed that, just as a pesticide repeatedly sprayed on a field will kill fewer and fewer bugs each time it is used, so too a given set of tests will eventually stop finding new defects when re-run against a system under development or maintenance. If the tests do not provide adequate coverage, this slowdown in defect finding will result in a false level of confidence and excessive optimism among the project team. However, the air will be let out of the balloon once the system is released to customers and users.

Using the right test strategies is the first step towards achieving adequate coverage. However, no strategy is perfect. You should plan to regularly review the test results during the project, and revise the tests based on your findings. In some cases, you need to write new and different tests to exercise different parts of the software or system. These new tests can lead to discovery of previously unknown defect clusters, which is a good reason not to wait until the end of the test effort to review your test results and evaluate the adequacy of test coverage.

The pesticide paradox is important when implementing the multilevel testing discussed previously in regards to the principle of early testing. Simply repeating our tests of the same conditions over and over will not result in good cumulative defect detection. However, when used properly, each type and level of testing has its own strengths and weaknesses in terms of defect detection, and collectively we can assemble a very effective sequence of defect filters from them. After such a sequence of complementary test activities, we can be confident that the coverage is adequate, and that the remaining level of risk is acceptable.

Sometimes the pesticide paradox can work in our favour, if it is not *new* defects that we are looking for. When we run automated regression tests, we are ensuring that the software that we are testing is still working as it was before; that is, there are no new unexpected side-effect defects that have appeared as a result of a change elsewhere. In this case, we are pleased that we have not found any new defects.

### *Principle 6. Testing is context dependent*

Our safety-critical clients test with a great deal of rigour and care – and cost. When lives are at stake, we must be extremely careful to minimize the risk of undetected defects. Our clients who release software on the web, such as e-commerce sites, or who develop mobile apps, can take advantage of the possibility to quickly change the software when necessary, leading to a different set of testing challenges – and opportunities. If you tried to apply safety-critical approaches to a mobile app, you

might put the company out of business; if you tried to apply e-commerce approaches to safety-critical software, you could put lives in danger. So, the context of the testing influences how much testing we do and how the testing is done.

Another example is the way that testing is done in an Agile project as opposed to a sequential life cycle project. Every sprint in an Agile project includes testing of the functionality developed in that sprint; the testing is done by everyone on the Agile team (ideally) and the testing is done continually over the whole of development. In sequential life cycle projects, testing may be done more formally, documented in more detail and may be focused towards the end of the project.

### *Principle 7. Absence-of-errors is a fallacy*

Throughout this section we have expounded the idea that a sequence of test activities, started early and targeting specific and diverse objectives and areas of the system, can effectively and efficiently find – and help a project team to remove – a large percentage of the defects. Surely that is all that is required to achieve project success?

Sadly, it is not. Many systems have been built that failed in user acceptance testing or in the marketplace, such as the initial launch of the US healthcare.gov website, which suffered from serious performance and web access problems.

Consider desktop computer operating systems. In the 1990s, as competition peaked for dominance of the PC operating system market, Unix and its variants had higher levels of quality than DOS and Windows. However, 25 years on, Windows dominates the desktop marketplace. One major reason is that Unix and its variants were too difficult for most users in the early 1990s.

Consider a system that perfectly conforms to its requirements (if that were possible), which has been tested thoroughly and all defects found have been fixed. Surely this would be a success, right? Wrong! If the requirements were flawed, we now have a perfectly working wrong system. Perhaps it is hard to use, as in the previous example. Perhaps the requirements missed some major features that users were expecting or needed to have. Perhaps this system is quite OK, but a competitor has come out with a competing system that is easier to use, includes the expected features and is cheaper. Our 'perfect' system is not looking so good after all, even though it has effectively 'no defects' in terms of 'conformance to requirements'.

## 1.4 TEST PROCESS

**SYLLABUS LEARNING OBJECTIVES FOR 1.4 TEST PROCESS (K2)**

**FL-1.4.1** **Explain the impact of context on the test process (K2)**

**FL-1.4.2** **Describe the test activities and respective tasks within the test process (K2)**

**FL-1.4.3** **Differentiate the work products that support the test process (K2)**

**FL-1.4.4** **Explain the value of maintaining traceability between the test basis and test work products (K2)**

In this section, we will describe the test process: tasks, activities and work products. We will talk about the influence of context on the test process and the importance of traceability.

In this section, there are a large number of Glossary keywords (19 in all): **coverage**, **test analysis**, **test basis**, **test case**, **test completion**, **test condition**, **test control**, **test data**, **test design**, **test execution**, **test execution schedule**, **test implementation**, **test monitoring**, **test oracle**, **test planning**, **test procedure**, **test suite**, **testware** and **traceability**.

In Section 1.1, we looked at the definition of testing, and identified misperceptions about testing, including that testing is not just test execution. Certainly, test execution is the most visible testing activity. However, effective and efficient testing requires test approaches that are properly planned and carried out, with tests designed and implemented to cover the proper areas of the system, executed in the right sequence and with their results reviewed regularly. This is a process, with tasks and activities that can be identified and need to be done, sometimes formally and other times very informally. In this section, we will look at the test process in detail.

There is no 'one size fits all' test process, but testing does need to include common sets of activities, or it may not achieve its objectives. An organization may have a test strategy where the test activities are specified, including how they are implemented and when they occur within the life cycle. Another organization may have a test strategy where test activities are not formally specified, but expertise about test activities is shared among team members informally. The 'right' test process for you is one that achieves your test objectives in the most efficient way. The best test process for you would not be the best for another organization (and vice versa).

Simply having a defined test strategy is not enough. One of our clients recently was a law firm that sued a company for a serious software failure. It turned out that while the company had a written test strategy, this strategy was not aligned with the testing best practices described in this book or the Syllabus. Further, upon close examination of their test work products, it was clear that they had not even carried out the strategy properly or completely. The company ended up paying a substantial penalty for their lack of quality. So, you must consider whether your actual test activities and tasks are sufficient.

## 1.4.1 Test process in context

As mentioned above, there is no one right test process that applies to everyone; each organization needs to adapt their test process depending on their context. The factors that influence the particular test process include the following (this list is not exhaustive):

- Software development life cycle model and project methodologies being used. An Agile project developing mobile apps will have quite a different test process to an organization producing medical devices such as pacemakers.
- Test levels and test types being considered; for example, a large complex project may have several types of integration testing, with a test process reflecting that complexity.
- Product and project risks (the lower the risks, the less formal the process needs to be, and vice versa).

- Business domain (e.g. mobile apps versus medical devices).
- Operational constraints, including:
  - budgets and resources
  - timescales
  - complexity
  - contractual and regulatory requirements.
- Organizational policies and practices.
- Required internal and external standards.

In the following sections, we will look in detail at the following topics:

- Test activities and tasks – the various things that testers do.
- Test work products – the things produced by and used by testers.
- Traceability between the test basis and test work products and why this is important.

First, one aspect to consider is **coverage**. Coverage is a partial measure of the thoroughness of testing. When examining the **test basis**, which is whatever the tests are being derived from (such as a requirement, user story, design or even code), a number of things can be identified as coverage items (we can tell whether or not we have tested them). For example, system level testing may want to ensure that every user story has been tested at least once; integration level testing may want to ensure that every communication path has been tested at least once; and, in component testing, developers may want to ensure that every code module, branch or statement has been tested at least once. When a test exercises the coverage item (user story, communication path or code element), then that item has been covered. Coverage is the percentage of coverage items that were exercised in a given test run.

**Coverage** The degree to which specified coverage items have been determined to have been exercised by a test suite expressed as a percentage.

**Test basis** The body of knowledge used as the basis for test analysis and design.

It is useful to know what you want to cover with your testing right from the start; coverage can act as a Key Performance Indicator (KPI) and help to measure the achievement of test objectives (if they are related to coverage).

In addition to the coverage items relating to specifications or code, we may also have environmental aspects that we want to cover. For example, tests for mobile apps may need to be tested on a number of mobile devices and configurations – these are also coverage items or coverage criteria. We could also consider coverage items of user personas, stakeholders or user success criteria. Measuring and reporting the coverage of these aspects can also give confidence to stakeholders that failures in operation would be less likely.

There is more about coverage in Section 4.3. Test processes are described in more detail in ISO/IEC/IEEE 29119-2 [2013].

## 1.4.2 Test activities and tasks

A test process consists of the following main groups of activities:

- Test planning.
- Test monitoring and control.
- Test analysis.
- Test design.

- Test implementation.
- Test execution.
- Test completion.

These activities appear to be logically sequential, in the sense that tasks within each activity often create the preconditions or precursor work products for tasks in subsequent activities. However, in many cases, the activities in the process may overlap or take place concurrently or iteratively, provided that these dependencies are fulfilled. Each group of activities consists of many individual tasks; these will vary for different projects or releases. For example, in Agile development, we have small iterations of software design, build and test that happen continuously, and planning is also a very dynamic activity throughout. If there are multiple teams, some teams may be doing test analysis while other teams are in the middle of test implementation, for example.

Note that this is 'a' test process, not 'the' test process. We have found that most of these activities, and many of the tasks within these activities, are carried out in some form or another on most successful test efforts. However, you should expect to have to tailor your test process, its main activities and the constituent tasks based on the organizational, project, process and product needs, constraints and other contextual realities. In sequential development, there will also be overlap, combination, concurrency or even omission of some tasks; this is why a test process is tailored for each project.

**Test planning** The activity of establishing or updating a test plan.

**Test plan** Documentation describing the test objectives to be achieved and the means and the schedule for achieving them, organized to coordinate testing activities.

(Note that we have included the definition of test plan here, even though it is not listed in the Syllabus as a term that you need to know for this chapter; otherwise the definition of test planning is not very informative.)

**Test monitoring** A test management activity that involves checking the status of testing activities, identifying any variances from the planned or expected status and reporting status to stakeholders.

**Test control** A test management task that deals with developing and applying a set of corrective actions to get a test project on track when monitoring shows a deviation from what was planned.

## *Test planning*

**Test planning** involves defining the objectives of testing and the approach for meeting those objectives within project constraints and contexts. This includes deciding on suitable test techniques to use, deciding what tasks need to be done, formulating a test schedule and other things.

Metaphorically, you can think of test planning as similar to figuring out how to get from one place to another (without using your GPS – there is no GPS for testing). For small, simple and familiar projects, finding the route merely involves taking an existing map, highlighting the route and jotting down the specific directions. For large, complex or new projects, finding the route can involve a sophisticated process of creating a new map, exploring unknown territory and blazing a fresh trail.

We will discuss test planning in more detail in Section 5.2.

## *Test monitoring and control*

To continue our metaphor, even with the best map and the clearest directions, getting from one place to another involves careful attention, watching the dashboard, minor (and sometimes major) course corrections, talking with our companions about the journey, looking ahead for trouble, tracking progress towards the ultimate destination and coping with finding an alternate route if the road we wanted is blocked. So, in **test monitoring**, we continuously compare actual progress against the plan, check on the progress of test activities and report the test status and any necessary deviations from the plan. In **test control**, we take whatever actions are necessary to meet the mission and objectives of the project, and/or adjust the plan.

Test monitoring is the ongoing comparison of actual progress against the test plan, using any test monitoring metrics that we have defined in the test plan. Test progress against the plan is reported to stakeholders in test progress reports or stakeholder meetings. One option that is often overlooked is that if things are going very wrong,

it may be time to stop the testing or even stop the project completely. In our driving analogy, once you find out that you are headed in completely the wrong direction, the best option is to stop and re-evaluate, not continue driving to the wrong place.

One way we can monitor test progress is by using exit criteria, also known as 'definition of done' in Agile development. For example, the exit criteria for test execution might include:

- Checking test results and logs against specified coverage criteria (we have not finished testing until we have tested what we planned to test).
- Assessing the level of component or system quality based on test results and logs (e.g. the number of defects found or ease of use).
- Assessing product risk and determining if more tests are needed to reduce the risk to an acceptable level.

We will discuss test planning, monitoring and control tasks in more detail in Chapter 5.

### *Test analysis*

In **test analysis**, we analyze the test basis to identify testable features and define associated **test conditions**. Test analysis determines 'what to test', including measurable coverage criteria. We can say colloquially that the test basis is everything upon which we base our tests. The test basis can include requirements, user stories, design specifications, risk analysis reports, the system design and architecture, interface specifications and user expectations.

**Test analysis** The activity that identifies test conditions by analyzing the test basis.

**Test condition** (charter) An aspect of the test basis that is relevant in order to achieve specific test objectives. See also: exploratory testing.

In test analysis, we transform the more general testing objectives defined in the test plan into tangible test conditions. The way in which these are specifically documented depends on the needs of the testers, the expectations of the project team, any applicable regulations and other considerations.

Test analysis includes the following major activities and tasks:

- Analyze the test basis appropriate to the test level being considered. Examples of a test basis include:
  - Requirement specifications, for example, business requirements, functional requirements, system requirements, user stories, epics, use cases or similar work products that specify desired functional and non-functional component or system behaviour. These specifications say what the component or system should do and are the source of tests to assess functionality as well as non-functional aspects such as performance or usability.
  - Design and implementation information, such as system or software architecture diagrams or documents, design specifications, call flows, modelling diagrams (for example, UML or entity-relationship diagrams), interface specifications or similar work products that specify component or system structure. Structures for implemented systems or components can be a useful source of coverage criteria to ensure that sufficient testing has been done on those structures.
  - The implementation of the component or system itself, including code, database metadata and queries, and interfaces. Use all information about any aspect of the system to help identify what should be tested.
  - Risk analysis reports, which may consider functional, non-functional and structural aspects of the component or system. Testing should be more thorough in the areas of highest risk, so more test conditions should be identified in the highest-risk areas.

- Evaluate the test basis and test items to identify various types of defects that might occur (typically done by reviews), such as:
  - ambiguities
  - omissions
  - inconsistencies
  - inaccuracies
  - contradictions
  - superfluous statements.
- Identify features and sets of features to be tested.
- Identify and prioritize test conditions for each feature, based on analysis of the test basis, and considering functional, non-functional and structural characteristics, other business and technical factors, and levels of risks.
- Capture bi-directional traceability between each element of the test basis and the associated test conditions. This traceability should be bi-directional (we can trace in both forward and backward directions) so that we can check which test basis elements go with which test conditions (and vice versa) and determine the degree of coverage of the test basis by the test conditions. See Sections 1.4.3 and 1.4.4 for more on traceability. Traceability is also very important for maintenance testing, as we will discuss in Chapter 2, Section 2.4.

How are the test conditions actually identified from a test basis? The test techniques, which are described in Chapter 4, are used to identify test conditions. Black-box techniques identify functional and non-functional test conditions, white-box techniques identify structural test conditions and experience-based techniques can identify other important test conditions. Using techniques helps to reduce the likelihood of missing important conditions and helps to define more precise and accurate test conditions.

Sometimes the test conditions identified can be used as test objectives for a test charter. In exploratory testing, an experience-based technique (see Chapter 4, Section 4.4.2), test charters are used as goals for the testing that will be carried out in an exploratory way – that is, test design, execution and learning in parallel. When these test objectives are traceable to the test basis, the coverage of those test conditions can be measured.

One of the most beneficial side effects of identifying what to test in test analysis is that you will find defects; for example, inconsistencies in requirements, contradictory statements between different documents, missing requirements (such as no 'otherwise' for a selection of options) or descriptions that do not make sense. Rather than being a problem, this is a great opportunity to remove these defects before development goes any further. This verification (and validation) of specifications is particularly important if no other review processes for the test basis documents are in place.

Test analysis can also help to validate whether the requirements properly capture customer, user and other stakeholder needs. For example, techniques such as behaviour-driven development (BDD) and acceptance test-driven development (ATDD) both involve generating test conditions (and test cases) from user stories. BDD focuses on the behaviour of the system and ATDD focuses on the user view of the system, and both techniques involve defining acceptance criteria. Since these acceptance criteria are produced before coding, they also verify and validate the user stories and the acceptance criteria. More about this is found in the ISTQB Foundation Level Agile Tester Extension qualification.

### Test design

Test analysis addresses 'what to test' and **test design** addresses the question 'how to test'; that is, what specific inputs and data are needed in order to exercise the software for a particular test condition. In test design, test conditions are elaborated (at a high level) in **test cases**, sets of test cases and other testware. Test analysis identifies general 'things' to test, and test design makes these general things specific for the component or system that we are testing.

Test design includes the following major activities:

- Design and prioritize test cases and sets of test cases.
- Identify the necessary **test data** to support the test conditions and test cases as they are identified and designed.
- Design the test environment, including set-up, and identify any required infrastructure and tools.
- Capture bi-directional traceability between the test basis, test conditions, test cases and test procedures (see also Section 1.4.4).

As with the identification of test conditions, test techniques are used to derive or elaborate test cases from the test conditions. These are described in Chapter 4, where test analysis and test design are discussed in more detail.

Just as in test analysis, test design can also identify defects – in the test basis and in the existing test conditions. Because test design is a deeper level of detail, some defects that were not obvious when looking at test basis at a high level, may become clear when deciding exactly what values to assign to test cases. For example, a test condition might be to check the boundary values of an input field, but when determining the exact values, we realize that a maximum value has not been specified in the test basis. Identifying defects at this point is a good thing because if they are fixed now, they will not cause problems later.

Which of these specific tasks applies to a particular project depends on various contextual issues relevant to the project, and these are discussed further in Chapter 5.

**Test design** The activity of deriving and specifying test cases from test conditions.

**Test case** A set of preconditions, inputs, actions (where applicable), expected results and postconditions, developed based on test conditions.

**Test data** Data created or selected to satisfy the execution preconditions and inputs to execute one or more test cases.

### Test implementation

In **test implementation**, we specify **test procedures** (or test scripts). This involves combining the test cases in a particular order, as well as including any other information needed for test execution. Test implementation also involves setting up the test environment and anything else that needs to be done to prepare for test execution, such as creating testware. Test design asked 'how to test', and test implementation asks 'do we now have everything in place to run the tests?'

Test implementation includes the following major activities:

- Develop and prioritize the test procedures and, potentially, create automated test scripts.
- Create **test suites** from the test procedures and automated test scripts (if any). See Chapter 6 for test automation.
- Arrange the test suites within a **test execution schedule** in a way that results in efficient test execution (see Chapter 5, Section 5.2.4).
- Build the test environment (possibly including test harnesses, service virtualization, simulators and other infrastructure items) and verify that everything needed has been set up correctly.

**Test implementation** The activity that prepares the testware needed for test execution based on test analysis and design.

**Test procedure** A sequence of test cases in execution order, and any associated actions that may be required to set up the initial preconditions and any wrap-up activities post execution.

**Test suite** (test case suite, test set) A set of test cases or test procedures to be executed in a specific test cycle.

**Test execution schedule** A schedule for the execution of test suites within a test cycle.

- Prepare test data and ensure that it is properly loaded in the test environment (including inputs, data resident in databases and other data repositories, and system configuration data).
- Verify and update the bi-directional traceability between the test basis, test conditions, test cases, test procedures and test suites (see also Section 1.4.4).

Ideally, all of these tasks are completed before test execution begins, because otherwise precious, limited test execution time can be lost on these types of preparatory tasks. One of our clients reported losing as much as 25% of the test execution period to what they called 'environmental shakedown', which turned out to consist almost entirely of test implementation activities that could have been completed before the software was delivered.

Note that although we have discussed test design and test implementation as separate activities, in practice they are often combined and done together.

Not only are test design and implementation combined, but many test activities may be combined and carried out concurrently. For example, in exploratory testing (see Chapter 4, Section 4.4.2), test analysis, test design, test implementation and test execution are done in an interactive way throughout an exploratory test session.

### *Test execution*

**Test execution** The process of running a test on the component or system under test, producing actual result(s).

**Testware** Work products produced during the test process for use in planning, designing, executing, evaluating and reporting on testing.

In **test execution**, the test suites that have been assembled in test implementation are run, according to the test execution schedule.

Test execution includes the following major activities:

- Record the identities and versions of all of the test items (parts of the test object to be tested), test objects (system or component to be tested), test tools and other **testware**.
- Execute the tests either manually or by using an automated test execution tool, according to the planned sequence.
- Compare actual results with expected results, observing where the actual and expected results differ. These differences may be the result of defects, but at this point we do not know, so we refer to them as anomalies.
- Analyze the anomalies in order to establish their likely causes. Failures may occur due to defects in the code or they may be false-positives. (A false-positive is where a defect is reported when there is no defect.) A failure may also be due to a test defect, such as defects in specified test data, in a test document or the test environment, or simply due to a mistake in the way the test was executed.
- Report defects based on the failures observed (see Chapter 5, Section 5.6). A failure due to a defect in the code means that we can write a defect report. Some organizations track test defects (i.e. defects in the tests themselves), while others do not.
- Log the outcome of test execution (e.g. pass, fail or blocked). This includes not only the anomalies observed and the pass/fail status of the test cases, but also the identities and versions of the software under test, test tools and testware.

- As necessary, repeat test activities when actions are taken to resolve discrepancies. For example, we might need to re-run a test that previously failed in order to confirm a fix (confirmation testing). We might need to run an updated test. We might also need to run additional, previously executed tests to see whether defects have been introduced in unchanged areas of the software or to see whether a fixed defect now makes another defect apparent (regression testing).
- Verify and update the bi-directional traceability between the test basis, test conditions, test cases, test procedures and test results.

As before, which of these specific tasks applies to a particular project depends on various contextual issues relevant to the project; these are discussed further in Chapter 5.

### *Test completion*

**Test completion** activities collect data from completed test activities to consolidate experience, testware and any other relevant information. Test completion activities should occur at major project milestones. These can include when a software system is released, when a test project is completed (or cancelled), when an Agile project iteration is finished (e.g. as part of a retrospective meeting), when a test level has been completed or when a maintenance release has been completed. The specific milestones that involve test completion activities should be specified in the test plan.

**Test completion** The activity that makes test assets available for later use, leaves test environments in a satisfactory condition and communicates the results of testing to relevant stakeholders.

Test completion includes the following major activities:

- Check whether all defect reports are closed, entering change requests or product backlog items for any defects that remain unresolved at the end of test execution.
- Create a test summary report to be communicated to stakeholders.
- Finalize and archive the test environment, the test data, the test infrastructure and other testware for later reuse.
- Hand over the testware to the maintenance teams, other project teams, and/or other stakeholders who could benefit from its use.
- Analyze lessons learned from completed test activities to determine changes needed for future iterations, releases and projects (i.e. perform a retrospective).
- Use the information gathered to improve test process maturity, especially as an input to test planning for future projects.

The degree and extent to which test completion activities occur, and which specific test completion activities do occur, depends on various contextual issues relevant to the project, which are discussed further in Chapter 5.

## 1.4.3 Test work products

'Work products' is the generic name given to any form of documentation, informal communication or artefact that is used in testing (or indeed in development). When testing is more formal, the majority of work products may be written documentation; when testing is very informal, the corresponding work product may just be a scrap of paper, or a note in someone's mobile phone. 'Work product' is a general term covering any type of information needed to do the work (testing or development).

Test work products are created as part of the test process, and there is significant variation in the types of work products created, in the ways they are organized and managed, and in the names used for them. The work products described in this section are in the ISTQB Glossary of terms. More information can be found in ISO/IEC/IEEE 29119-3 [2013].

Test work products can be captured, stored and managed in configuration management tools, or possibly in test management tools or defect management tools.

### Test planning work products

You will not be surprised to find that test planning work products include test plans. There may be different test plans for different test levels. The test plan typically includes information about the test basis, to which all the other work products will be related via traceability information (see Section 1.4.4). Test plans also include entry and exit criteria (also known as definition of ready and definition of done) for the testing within their scope – the exit criteria are used during test monitoring and control.

Beware of what people call a 'test plan'; we have seen this name applied to any kind of test document, including test case specifications and test execution schedules. A test plan is a planning document – it contains information about what is intended to happen in the future, and is similar to a project plan. It does not contain detail of test conditions, test cases or other aspects of testing.

The test plan needs to be understandable to those who need to know the information contained in it. The two-page cryptic diagram that was called a 'test plan' at one organization would not be the right sort of work product for other organizations.

Test plans can cover a whole project, or be specific to a test level or type of testing. Test plans are covered in more detail in Chapter 5, Section 5.2.

### Test monitoring and control work products

The work products associated with test monitoring and control typically include different types of test reports. Test progress reports are produced on an ongoing and/or a regular basis to keep stakeholders updated about progress, on a weekly or monthly basis, for example. Test summary reports are produced at test completion milestones, as a way of summarizing the testing for a particular unit of work or test project. Any test report needs to include details relevant to its intended audience, the date of the report and the time period covered by the report. Test reports may include test execution results once those are available, in summary form (e.g. number of tests run, failed, passed or blocked and number of defects raised of different severity), and the status compared to the exit criteria or definition of done.

Test monitoring and control work products should address project management concerns such as budget and schedule, task completion, resource allocation, usage and effort. If action needs to be taken on the basis of information reported in a test report, those actions should be summarized in the next report to ensure that the desired effect of the action has been achieved.

These work products are further explained in Chapter 5, Section 5.3.

### Test analysis work products

Test analysis work products include mainly test conditions, as this is the output of the test analysis activity. Each test condition is ideally traceable to the test basis (and vice versa). In exploratory testing, a test charter may be a test analysis work product. Defect reports about defects found in the test basis as a result of test analysis can also be considered a work product from test analysis.

Here is an example: The specification for an ordering system describes a sales discount feature when customers put in large orders. The test conditions might be to test all the discount values for various order values. The discount values would be coverage items – we want to make sure we have tested each of them at least once. The work product would be the list of test conditions, that is, the discount values.

Test conditions are further discussed in Chapter 4.

### *Test design work products*

The main work products resulting from test design are test cases and sets of test cases that exercise the test conditions identified in test analysis.

Sometimes the test cases at this stage are still rather vague and high-level, that is, without concrete values for inputs and expected results. For example, a test case for the sales discount might be to set up four existing customers, one who orders only a small amount so does not qualify for a discount, and the other three who order enough to qualify for a discount at each of the three discount levels respectively.

Having the test cases at a high level means that we can use the same test case across multiple test cycles with different specific or concrete data. For example, one application may have discounts of 2%, 5% and 10%, and another may have discounts of 10%, 20% and 25%. Our high-level test case adequately documents the scope of the test even though the details will be different in each application. The test case is traceable to and from the test condition that it is derived from.

We have seen that high-level test cases can have advantages, but there are also some aspects that you need to be aware of with high-level test cases. For example, it may be difficult to reproduce the test exactly; different testers may use different test data, so the test case is not exactly repeatable. A high-level test case is not directly automatable; the tool needs exact instructions and specific data in order to execute the test. The skill and domain knowledge of the tester is also critical; a junior new-hire with no domain knowledge may struggle to know what they are supposed to be doing, unless they are well supported by more experienced testers. These are not insurmountable problems, but they do need to be considered.

Test design work products may also include test data, the design of the test environment and the identification of infrastructure and tools. The extent and way in which these are documented may vary significantly from project to project or from one company to another.

When deriving test cases from the test conditions, we may also find defects or improvements that we could make to the test conditions, so the test conditions themselves may be further refined during test design. In our sales discount example, in test analysis we identified the three discounts as test conditions, but in test design, by looking at the test cases, we identified the 'no discount' test condition – a discount of 0%.

Test cases are further discussed in Chapter 4.

### *Test implementation work products*

Work products for test implementation include:

- test procedures and the sequencing of those procedures.
- test suites.
- a test execution schedule.

At this point, because we are preparing for test execution, we need to further refine any high-level test cases into low-level test cases that use concrete and specific data, both for test inputs and test data. In our loyalty discount example, we would now

need to specify the details of our existing customers, decide on exactly how much each order will come to and calculate the final amount they would pay, including the discount. So, for example, Mrs Smith puts in an order for $50.01. Because her order is over $50, she gets a 10% discount, so she pays $45.01. We calculate the expected result for the test using a **test oracle** – the source of what the correct answer should be (in this case simple arithmetic). We would also need to set up Mrs Smith and other customers in the database as part of the preconditions of running the test, and this would be included in the test procedure.

**Test oracle** (oracle) A source to determine expected results to compare with the actual result of the system under test.

In exploratory testing, we may be creating work products for test design and test implementation while doing test execution; traceability may be more difficult in this case.

Test implementation may also create work products that will be used by tools, for example, test scripts for test execution tools, and sometimes work products are created *by* tools, such as a test execution schedule. Service virtualization may also create test implementation work products.

As in test design, we may further refine test conditions (and high-level test cases) during test implementation. For example, by deciding on the concrete values for our sales discount example, we realize that a test condition we omitted was to consider two different ways of clients paying between $45.01 and $50.00 (that is, with or without a discount). This may not be important to include in our tests, but it is an additional test condition.

### Test execution work products

Work products for test execution include:

- documentation of the status of individual test cases or test procedures (e.g. ready to run, passed, failed, blocked, deliberately skipped, etc.)
- defect reports (see Chapter 5, Section 5.6)
- documentation about which test item(s), test object(s), test tools and testware were involved in the testing.

In test execution, we want to know what happened when the tests were run. We want to know about any problems encountered that may have blocked some tests running, for example if the network was down for a time. We want to know whether or not we were able to execute all of the tests that we planned to execute (which is not necessarily all of the tests that we have designed or implemented).

When test execution is finished (or is stopped), we should be able to report test results based on traceability, so that if a set of tests that were all related to the same requirement or user story failed, we will know about that. We also want to know which requirements have passed all of their planned tests, and any that have not yet been tested – that is, the tests are still waiting to be run. This is particularly important when test execution is stopped rather than completed.

Stakeholders are more likely to appreciate the implications of failing tests if they can be related to user stories or requirements, rather than just being told that a certain number of tests passed or failed. This is why traceability is important.

### Test completion work products

The work products for test completion include the following:

- test summary reports
- action items for improvement of subsequent projects or iterations (e.g. following an Agile project retrospective)
- change requests or product backlog items
- finalized testware.

Test completion work products give closure to the whole of the test process and should provide ongoing ideas for increasing the effectiveness and efficiency of testing within the organization in the future.

### 1.4.4 Traceability between the test basis and test work products

**Traceability** The degree to which a relationship can be established between two or more work products.

We have mentioned bi-directional **traceability** for all test work products covered in Section 1.4.3. Bi-directional means that we can trace, for example, a given requirement through test conditions and test cases to the test execution results, but we can also trace the test execution results back to test cases, test cases back to test conditions, and test conditions back to requirements. No matter what the work products are called, this cross-linking gives many benefits to the test process. We saw how traceability can aid in the measurement and reporting of test coverage in order to report coverage and defects related to requirements, which is more meaningful and of more value to stakeholders.

Good traceability also supports the following:

- analyzing the impact of changes, whether to requirements or to the component or system
- making testing auditable, and being able to measure coverage
- meeting IT governance criteria (where applicable)
- improving the coherence of test progress reports and test summary reports to stakeholders, as described above
- relating the technical aspects of testing to stakeholders in terms that they can understand
- providing information to assess product quality, process capability and project progress against business goals.

You may find that your test management or requirements management tool provides support for traceability of work products; if so, make use of that feature. Some organizations find that they have to build their own management systems in order to organize test work products in the way that they want and to ensure that they have bi-directional traceability. However the support is implemented, it is important to have automated support for traceability – it is not something that can be sustained without tool support.

## 1.5 THE PSYCHOLOGY OF TESTING

**SYLLABUS LEARNING OBJECTIVES FOR 1.5 THE PSYCHOLOGY OF TESTING (K2)**

**FL-1.5.1** **Identify the psychological factors that influence the success of testing (K1)**

**FL-1.5.2** **Explain the difference between the mindset required for test activities and the mindset required for development activities (K2)**

## 1.5.1 Human psychology and testing

In this section, we'll discuss the various psychological factors that influence testing and its success. We'll also contrast the mindset of a tester and a developer. For a good introduction to the psychology of testing see Weinberg [2008].

As mentioned earlier, the ISTQB definition of software testing includes both dynamic testing and static testing. Let's review the distinction again. Dynamic software testing involves actually executing the software or some part of it, such as checking an application-produced report for accuracy or checking response time to user input. Static software testing does not execute the software but uses two possible approaches: automated static analysis on the code (e.g. evaluating its complexity) or on a document (e.g. to evaluate the readability of a use case); or reviews of code or documents (e.g. to evaluate a requirement specification for consistency, ambiguity and completeness).

### *Finding defects*

While dynamic and static testing are very different types of activities, they have in common their ability to find defects. Static testing finds defects directly, while dynamic testing finds evidence of a defect through a failure of the software to behave as expected. Either way, people carrying out static or dynamic tests must be focused on the possibility – indeed, the high likelihood in many cases – of finding defects. Indeed, finding defects is often a primary objective of static and dynamic testing activities.

Identifying defects may unfortunately be perceived by developers as a criticism, not only of the product but also of its author – and in a sense, it is. But finding defects in testing should be *constructive* criticism, where testers have the best interest of the developer in mind. One meaning of the word 'criticism' is 'an examination, interpretation, analysis or judgement about something' – this is an objective assessment. But other meanings include disapproval by pointing out faults or shortcomings and even an attack on someone or something. Testing does not want to be the latter sense of criticism, but even when intended in the first sense, it can be perceived in the other ways. Testers need to be diplomats, along with everything else.

### *Bias*

However, there is another factor at work when we are reporting defects; the author (developer) believes that their code is correct – they obviously did not write it to be intentionally wrong. This confidence in their understanding is in some sense necessary for developers; they cannot proceed without it. But at the same time, this confidence creates confirmation bias. Confirmation bias makes it difficult to accept information that disagrees with your currently held beliefs. Simply put, the author of a work product has confidence that they have solved the requirements, design, metadata or code problem, at least in an acceptable fashion; however, strictly speaking, that is false confidence. Other biases may also be at work, and it is also human nature to blame the bearer of bad news (which defects are perceived to be).

Testers are not always aware of their biases either, and they do have biases of their own. Since those biases are different from the developers, that is a benefit, but a lack of awareness of those biases sets up potential conflict.

This reluctance to accept that their work is not perfect is why some people regard testing as a destructive activity (trying to destroy their work) rather than the constructive activity it is (trying to construct better quality software). Good testing contributes greatly to product quality and project quality, as we saw in Sections 1.1 and 1.2.

While some developers are aware of their biases when they participate in reviews and perform unit testing of their own work products, those biases act to impede their effectiveness at finding their own defects. The mental mistakes that caused them to create the defects remain in their minds in most cases. When proofreading our own work, for example, we see what we meant, not what we wrote.

### Review and test your own work?

Should software work product developers – business analysts, system designers, architects, database administrators and developers – review and test their own work? They certainly should; they have a deep understanding about the system, and quality is everyone's responsibility.

However, many business analysts, system designers, architects, database administrators and developers do not know the review, static analysis and dynamic testing techniques discussed in the Foundation Syllabus and this book. While that situation is gradually changing, much of the self-testing by software work product developers is either not done or is not done as effectively as it could be. The principles and techniques in the Foundation Syllabus and this book are intended to help either testers or others to be more effective at finding defects, both their own and those of others.

### Attitudes

It is a particular problem when a tester revels in being the bearer of bad news. For example, one tester made a revealing – and not very flattering – remark during an interview with one of the authors. When asked what he liked about testing, he responded, 'I like to catch the developers'. He went on to explain that, when he found a defect in someone's work, he would go and demonstrate the failure on the programmer's workstation. He said that he made sure that he found at least one defect in everyone's work on a project, and went through this process of ritually humiliating the programmer with each and every one of his colleagues. When asked why, he said, 'I want to prove to everyone that I am their intellectual equal'. This person, while possessing many of the skills and traits one would want in a tester, had exactly the wrong personality to be a truly professional tester.

Instead of seeing themselves as their colleagues' adversaries or social inferiors out to prove their equality, testers should see themselves as teammates. In their special role, testers provide essential services in the development organization. They should ask themselves, 'Who are the stakeholders in the work that I do as a tester?' Having identified these stakeholders, they should ask each stakeholder group, 'What services do you want from testing, and how well are we providing them?'

While the specific services are not always defined, it is common that mature and wise developers know that studying their mistakes and the defects they have introduced is the key to learning how to get better. Further, smart software development managers understand that finding and fixing defects during testing not only reduces the level of risk to the quality of the product, it also saves time and money when compared to finding defects in production.

### Communication

Clearly defined objectives and goals for testing, combined with constructive styles of communication on the part of test professionals, will help to avoid most negative personal or group dynamics between testers and their colleagues in the development team. Whenever defects are found, true testing professionals distinguish themselves by demonstrating good interpersonal skills. True testing professionals communicate

facts about defects, progress and risks in an objective and constructive way that counteracts these misperceptions as much as possible. This helps to reduce tensions and build positive relationships with colleagues, supporting the view of testing as a constructive and helpful activity. While this is not necessary, we have noticed that many consummate testing professionals have business analysts, system designers, architects, developers and other specialists with whom they work as close personal friends.

This applies not only to testers but also to test managers, and not just to defects and failures but to all communication about testing, such as test results, test progress and risks.

Having good communication skills is a complex topic, well beyond the scope of a book on fundamental testing techniques. However, we can give you some basics for good communication with your development colleagues:

- Remember to think of your colleagues as teammates, not as opponents or adversaries. The way you regard people has a profound effect on the way you treat them. You do not have to think in terms of kinship or achieving world peace, but you should keep in mind that everyone on the development team has the common goal of delivering a quality system, and everyone must work together to accomplish that. Start with collaboration, not battles.
- Make sure that you focus on and emphasize the value and benefits of testing. Remind your developer colleagues that defect information provided by testing can help them to improve their own skills and future work products. Remind managers that defects found early by testing and fixed as soon as possible will save time and money and reduce overall product quality risk. Also, be sure to respond well when developers find problems in your own test work products. Ask them to review them and thank them for their findings (just as you would like to be thanked for finding problems in their work).
- Recognize that your colleagues have pride in their work, just as you do, and as such you owe them a tactful communication about defects you have found. It is not really any harder to communicate your findings, especially the potentially embarrassing findings, in a neutral, fact-focused way. In fact, you will find that if you avoid criticizing people and their work products, but instead keep your written and verbal communications objective and factual, you will also avoid a lot of unnecessary conflict and drama with your colleagues.
- Before you communicate these potentially embarrassing findings, mentally put yourself in the position of the person who created the work product. How are they going to feel about this information? How might they react? What can you do to help them get the essential message that they need to receive without provoking a negative emotional reaction from them?
- Keep in mind the psychological element of cognitive dissonance. Cognitive dissonance is a defect – or perhaps a feature – in the human brain that makes it difficult to process unexpected information, especially bad news. So, while you might have been clear in what you said or wrote, the person on the receiving end might not have clearly understood. Cognitive dissonance is a two-way street, too, and it is quite possible that you are misunderstanding someone's reaction to your findings. So, before assuming the worst about someone and their motivations, confirm that the other person has understood what you have said and vice versa.

## 1.5.2 Tester's and developer's mindsets

Testers and developers actually have different thought processes and different objectives for their work. A mindset reflects an individual's assumptions and the way that they like to solve problems and make decisions.

A tester's mindset should include the following:

- Curiosity. Good testers are curious about why systems behave the way they do and how systems are built. When they see unexpected behaviour, they have a natural urge to explore further, to isolate the failure, to look for more generalized problems and to gain deeper understanding.
- Professional pessimism. Good testers expect to find defects and failures. They understand human fallibility and its implications for software development. (However, this is not to say that they are negative or adversarial, as we'll discuss in a moment.)
- A critical eye. Good testers couple this professional pessimism with a natural inclination to doubt the correctness of software work products and their behaviours as they look at them. A good tester has, as a personal slogan, 'If in doubt, it is a bug'.
- Attention to detail. Good testers notice everything, even the smallest details. Sometimes these details are cosmetic problems like font-size mismatches, but sometimes these details are subtle clues that a serious failure is about to happen. This trait is both a blessing and a curse. Some testers find that they cannot turn this trait off, so they are constantly finding defects in the real world – even when not being paid to find them.
- Experience. Good testers not only know a defect when they see one, they also know where to look for defects. Experienced testers have seen a veritable parade of bugs in their time, and they leverage this experience during all types of testing, especially experience-based testing such as error guessing (see Chapter 4).
- Good communication skills. All of these traits are essential, but, without the ability to effectively communicate their findings, testers will produce useful information that will, alas, be put to no use. Good communicators know how to explain the test results, even negative results such as serious defects and quality risks, without coming across as preachy, scolding or defeatist.

A good tester has the skills, the training, the certification and the mindset of a professional tester, and of these four skills, the most important – and perhaps the most elusive – is the mindset.

The tester's mindset is to think about what could go wrong and what is missing. The tester looks at a statement in a requirement or user story and asks, 'What if it isn't? What haven't they thought of here? What could go wrong?' That mindset is quite different from the mindset that a business analyst, system designer, architect, database administrator or developer must bring to creating the work products involved in developing software. While the testers (or reviewers) must assume that the work product under review or test is defective in some way – and it is their job to find those defects – the people developing that work product must have confidence that they understand how to do so properly. Looking at a statement in a requirement or user story, the developer thinks, 'How can I implement this? What technical challenges do I need to solve?'

Having a tester or group of testers who are organizationally separate from development, either as individuals or as an independent test team, can provide significant benefits, such as increased defect-detection percentage. A tester's mindset is a 'different pair of eyes' and independent testers can see things that developers do not see (because of confirmation bias discussed above). This is especially important for large, complex or safety-critical systems.

However, independence from the developers does not mean an adversarial relationship with them. In fact, such a relationship is toxic, often fatally so, to a test team's effectiveness.

The softer side of software testing is often the harder side to master. A tester may have adequate or even excellent technique skills and certifications, but if they do not have adequate interpersonal and communication skills, they will not be an effective tester. Such soft skills can be improved with training and practice. The best testers continuously strive to attain a more professional mindset, and it is a lifelong journey.

## CHAPTER REVIEW

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

From Section 1.1, you should now know what testing is. You should be able to remember the typical objectives of testing. You should know the difference between testing and debugging. You should know the Glossary keyword terms **debugging**, **test object**, **test objective**, **testing**, **validation** and **verification**.

From Section 1.2, you should now be able to explain why testing is necessary and support that explanation with examples. You should be able to explain the difference between testing and quality assurance and how they work together to improve quality. You should be able to distinguish between an error (made by a person), a defect (in a work product) and a failure (where the component or system does not perform as expected). You should know the difference between the root cause of a defect and the effects of a defect or failure. You should know the Glossary terms **defect**, **error**, **failure**, **quality**, **quality assurance** and **root cause**.

You should be able to explain the seven principles of testing, discussed in Section 1.3.

From Section 1.4, you should now recognize a test process. You should be able to recall the main testing activities of test planning, test monitoring and control, test analysis, test design, test implementation, test execution and test completion. You should be familiar with the work products produced by each test activity. You should know the Glossary terms **coverage**, **test analysis**, **test basis**, **test case**, **test completion**, **test condition**, **test control**, **test data**, **test design**, **test execution**, **test execution schedule**, **test implementation**, **test monitoring**, **test oracle**, **test planning**, **test procedure**, **test suite**, **testware** and **traceability**.

From Section 1.5, you now should be able to explain the psychological factors that influence the success of testing. You should be able to explain and contrast the mindsets of testers and developers, and why these differences can lead to problems.

## SAMPLE EXAM QUESTIONS

**Question 1** What is NOT a reason for testing?

a. To enable developers to code as quickly as possible.

b. To reduce the risk of failures in operation.

c. To contribute to the quality of the components or systems.

d. To meet any applicable contractual or legal requirements.

**Question 2** Consider the following definitions and match the term with the definition.

1. A reason or purpose for designing and executing a test.
2. The component or system to be tested.
3. Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled.

a. 1) test object, 2) test objective, 3) validation.

b. 1) test objective, 2) test object, 3) validation.

c. 1) validation, 2) test basis, 3) verification.

d. 1) test objective, 2) test object, 3) verification.

**Question 3** Which statement about quality assurance (QA) is true?

a. QA and testing are the same.

b. QA includes both testing and root cause analysis.

c. Testing is quality control, not QA.

d. QA does not apply to testing.

**Question 4** It is important to ensure that test design starts during the requirements definition. Which of the following test objectives supports this?

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 a large number of defects during development, including system testing. Although the test manager understands that this is good defect finding within their budget for her test team and industry, senior management and executives remain disappointed in the test group, saying that the test team misses some bugs that the users find after release. Given that the users are generally happy with the system and that the failures that 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** What are the benefits of traceability between the test basis and test work products?

a. Traceability means that test basis documents and test work products do not need to be reviewed.

b. Traceability ensures that test work products are limited in number to save time in producing them.

c. Traceability enables test progress and defects to be reported with reference to requirements, which is more understandable to stakeholders.

d. Traceability enables developers to produce code that is easier to test.

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

a. Understanding what managers value about testing.

b. Explaining test results in a neutral fashion.

c. Identifying potential customer work-arounds for bugs.

d. Promoting better quality software whenever possible.

**Question 8** Given the following test work products, identify the major activity in a test process that produces it.

1. Test execution schedule.
2. Test cases.
3. Test progress reports.
4. Defect reports.

a. 1) – Test planning, 2) – Test design 3) – Test execution, 4) – Test implementation.

b. 1) – Test execution, 2) – Test analysis 3) – Test completion, 4) – Test execution.

c. 1) – Test control, 2) – Test analysis, 3) – Test monitoring, 4) – Test implementation.

d. 1) – Test implementation, 2) – Test design, 3) – Test monitoring, 4) – Test execution.

CHAPTER TWO

# Testing throughout the software development life cycle

Testing 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 LIFE CYCLE MODELS

**SYLLABUS LEARNING OBJECTIVES FOR 2.1 SOFTWARE DEVELOPMENT LIFE CYCLE MODELS (K2)**

**FL-2.1.1 Explain the relationships between software development activities and test activities in the software development life cycle (K2)**

**FL-2.1.2 Identify reasons why software development life cycle models must be adapted to the context of project and product characteristics (K1)**

In this section, we'll discuss software development models and how testing fits into them. We'll discuss sequential models, focusing on the V-model approach rather than the waterfall. We'll discuss iterative and incremental models such as Rational Unified Process (RUP), Scrum, Kanban and Spiral (or prototyping).

As we go through this section, watch for the Syllabus terms **commercial off-the-shelf (COTS)**, **sequential development model**, and **test level**. You will find these keywords defined in the Glossary (and ISTQB website).

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. 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.

Whichever life cycle model is being used, there are several characteristics of good testing:

- For every development activity there is a corresponding test 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 software development activity.
- Testers should participate in discussions to help define and refine requirements and design. They should also be involved in reviewing work products as soon as drafts are available in the software development cycle.

Recall from Chapter 1, testing Principle 3: 'Early testing saves time and money'. By starting testing activities as early as possible in the software development life cycle, we find defects while they are still small green shoots (in requirements, for example) before they have had a chance to grow into trees (in production). We also prevent defects from occurring at all by being more aware of what should be tested from the earliest point in whatever software development life cycle we are using.

We will look at two categories of software development life cycle models: sequential and iterative/incremental.

## *Sequential development models*

A **sequential development model** is one where the development activities happen in a prescribed sequence – at least that is the idea. The models assume a linear sequential flow of activities; the next phase is only supposed to start when the previous phase is complete. Berard [1993] said that developing software from requirements is like walking on water – it is easier if it is frozen. But practice does not conform to theory: the activities will overlap, and things will be discovered in a later phase that may invalidate assumptions made in previous phases (which were supposed to be finished).

**Sequential development model** A type of development life cycle model in which a complete system is developed in a linear way of several discrete and successive phases with no overlap between them.

The waterfall model (in Figure 2.1) 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. Different models have different levels; the figure shows one possible model. With all waterfall models, however, testing tends to happen towards the end of the life cycle, so defects are detected close to the live implementation date. With this model, it is 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.

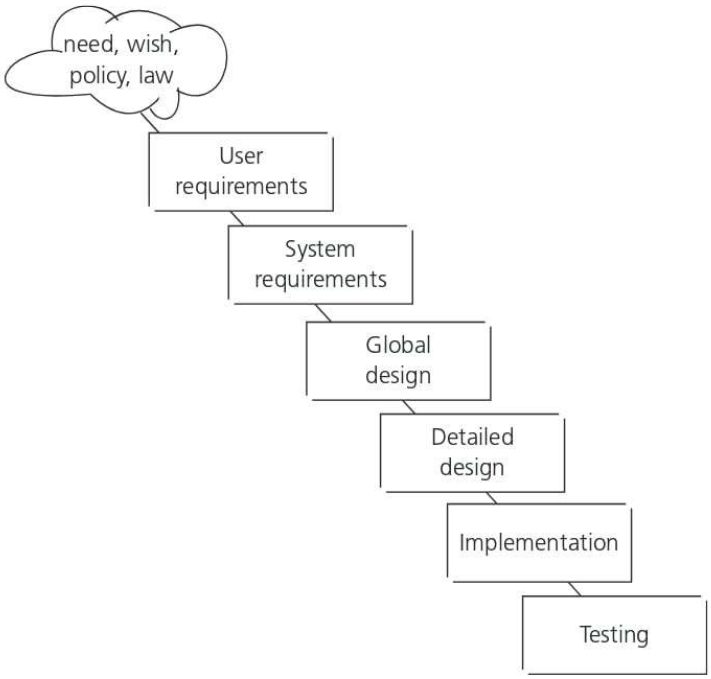


**FIGURE 2.1** Waterfall model

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 provides guidance on how testing begins as early as possible in the life cycle. 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 developers and business analysts so they can perform these activities and tasks, producing 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, for example reviewing the user requirements, and late in the life cycle, for example during user acceptance testing.

**Test level** (test stage) A specific instantiation of a test process.

Although variants of the V-model exist, a common type of V-model uses four **test levels**. (See Section 2.2 for more on 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 (for example modules, programs, objects, classes, etc.) that are separately testable.
- Integration testing: tests interfaces between components, interactions to different parts of a system such as an operating system, file system and hardware or interfaces between systems.

- System testing: concerned with the behaviour 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, requirements and business processes conducted to determine whether or not to accept the system.

In practice, a V-model may have more, fewer or different levels of development 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. In the V-model, there may also be overlapping of activities.

Sequential models aim to deliver all of the software at once, that is, the complete set of features required by stakeholders or users, or the software may be delivered in releases containing significant chunks of new functionality. However, typically this may take months or even years of development even for a single release.

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 different names.

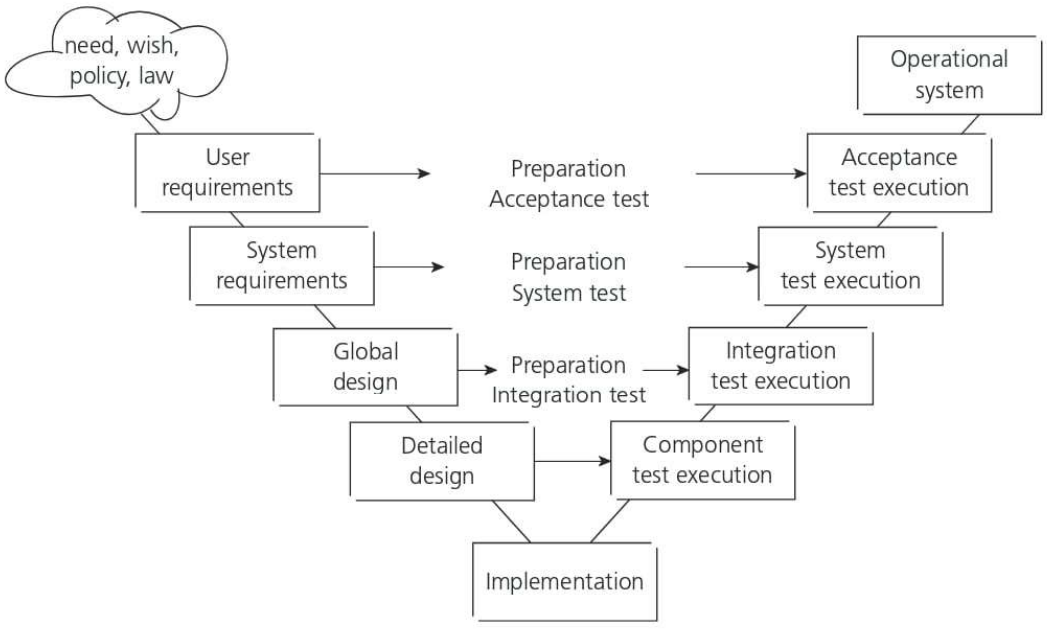


**FIGURE 2.2** V-model

### *Iterative and incremental development models*

Not all life cycles are sequential. There are also iterative and incremental life cycles where, instead of one large development timeline 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 and incremental life cycles.

To better understand the meaning of these two terms, consider the following two sequences of producing a painting.

- **Incremental:** complete one piece at a time (scheduling or staging strategy). Each increment may be delivered to the customer.

Images provided by: Mark Fewster, Grove Software Testing Ltd

- **Iterative:** start with a rough product and refine it, iteratively (rework strategy). Final version only delivered to the customer (although in practice, intermediate versions may be delivered to selected customers to get feedback).

Images provided by: Mark Fewster, Grove Software Testing Ltd

In terms of developing software, a purely iterative model does not produce a working system until the final iteration. The incremental approach produces working versions of parts of the system early on and each of these can be released to the customer. The advantage of this is that the customer can gain early benefit from using the deliveries and, perhaps most importantly, the customers can give valuable feedback. This feedback will influence what is done in future increments. Most iterative approaches also incorporate this feedback loop by delivering some (if not all) of the (intermediate) products created by the iterations.

The painting analogy shown above is not a perfect representation for the iterative approach. If the final product were to comprise 1,000 source code modules, you could be forgiven for thinking that an iterative approach would have people starting the first iteration by writing one line of code in each module and then have the second and subsequent iterations each adding another line of code to each module until they were completed. This is not the case.

In both iterative and incremental models, the features to be implemented are grouped together (for example according to business priority or risk). In this way, the focus is always on the most important of the outstanding features. The various project phases, including their work products and activities, then occur for each group of features. The phases may be done either sequentially or overlapping, and the iterations or increments themselves may be sequential or overlapping.

An iterative development model is shown in Figure 2.3.

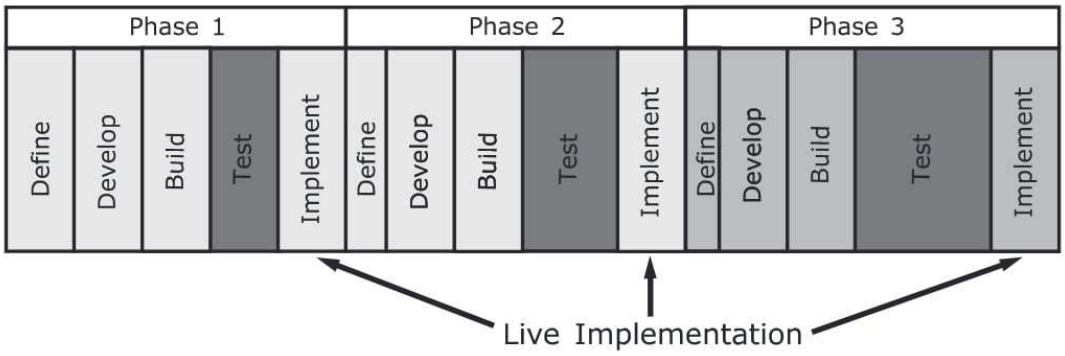


**FIGURE 2.3** Iterative development model

### *Testing in incremental and iterative development*

During project initiation, high-level test planning and test analysis occurs in parallel with the project planning and business/requirements analysis. Any detailed test planning, test analysis, test design and test implementation occurs at the beginning of each iteration.

Test execution often involves overlapping test levels. Each test level begins as early as possible and may continue after subsequent, higher test levels have started.

In an iterative or incremental life cycle, many of the same tasks will be performed but their timing and extent may vary. For example, rather than being able to implement the entire test environment at the beginning of the project, it may be more efficient to implement only the part needed for the current iteration. The testing tasks may be undertaken in a different order and not necessarily sequentially. There are likely to be fewer entry and exit criteria between activities compared with sequential models. Also, much of the test planning and completion reporting are more likely to occur at the start and end of the project, respectively, rather than at the start and end of each iteration.

With any of the iterative or incremental life cycle models, the farther ahead the planning occurs, the farther ahead the scope of the test process can extend.

Common issues with iterative and incremental models include:

- More regression testing.
- Defects outside the scope of the iteration or increment.
- Less thorough testing.

Because the system is being produced a bit at a time, at any given point there will some part which is completed in some sense, either an increment or the work that was done iteratively. This part will be tested and may be used by the customer or user to give feedback. When the next increment or iteration is developed, this will also be tested, but it is also important to do regression testing of the parts which have already been developed. The more iterations or increments there are, the more regression testing will be needed throughout development. (This type of testing is a good candidate for automation.)

Defects that are found in the part that you are currently testing are dealt with in the usual way, but what about defects found either by regression testing of previously developed parts, or discovered by accident when testing a new part? These defects do need to be logged and dealt with, but because they are outside the scope of the current iteration/increment, they can sometimes fall between the cracks and be forgotten, neglected or argued about.

There is a danger that the testing may be less thorough in incremental and iterative life cycles, particularly regression testing of previously developed parts, and especially if the regression testing is manual rather than automated. There is also a danger that the testing is less formal, because we are dealing with smaller parts of the system, and formality of the testing may seem like overkill for such a small thing.

Examples of iterative and incremental development models are Rational Unified Process (RUP), Scrum, Kanban and Spiral (or prototyping).

### *Rational Unified Process (RUP)*

RUP is a software development process framework from Rational Software Development, a division of IBM. It consists of four steps:

- Inception: the initial idea, planning (for example what resources would be needed) and go/no-go decision for development, done with stakeholders.
- Elaboration: further detailed investigation into resources, architecture and costs.
- Construction: the software product is developed, including testing.
- Transition: the software product is released to customers, with modifications based on user feedback.

This development process is tailorable for different contexts, and there are tools (and services) to support it. One of the basic principles is that development is iterative, with risk being the primary driver for decisions about the development. Another principle (relevant to us) is that the evaluation of quality (including testing) is continuous throughout development.

In RUP, the increments that are produced, although significantly smaller than what is produced by sequential models, are larger than the increments produced by Agile development (see Scrum below), and would typically take months rather than days or weeks to complete. They might contain groups of related features, for example.

### *Scrum*

Scrum is an iterative and incremental framework for effective team collaboration, which is typically used in Agile development, the most well-known iterative method. It (and Agile) is based on recognizing that change is inevitable and taking a practical empirical approach to changing priorities. Work is broken down into small units that can be completed in a fairly short time (days, a week or two or even a month). The delivery of a unit of work is called a sprint. For software development, a sprint includes all aspects of development for a particular feature or set of small features, everything from requirements (typically user stories) through to testing and (ideally) test automation.

The development teams are small (3 to 9 people) and cross-functional, that is, they include people who perform various roles, and often individuals take on different tasks (such as testing) within the team. The key roles are:

- The Product Owner represents the business, that is, stakeholders and end users.
- The Development team (which includes testers) makes its own decisions about development, that is, they are self-organizing with a high level of autonomy.
- The Scrum Master helps the development team to do their work as efficiently as possible, by interacting with other parts of the organization and dealing with problems. The Scrum Master is not a manager, but a facilitator for the team.

A stand-up meeting of typically around 15 minutes is held each day, for example first thing in the morning, to update everyone with progress from the previous day and plan the work ahead. (This is where the term 'scrum' came from, as in the gathering of a rugby team.)

At the start of a sprint, in the sprint planning meeting, some features are selected to be implemented, with other features being put on a backlog. Acceptance criteria apply to user stories, and are similar to test conditions, saying what needs to work for the user story to be considered working. A definition of done can apply to a user story (which includes but goes beyond satisfaction of the acceptance criteria), but also to unit testing, system testing, iterations and releases. After a sprint completes, a retrospective should be held to assess what went well and what could be improved for the next sprint.

Because development is limited to the sprint duration which is time-boxed, flexibility is in choosing what can be developed in the time. Compare that to sequential models, where all the features are selected first and the time taken to develop all of them is based on that. Thus Scrum (and Agile) enable us to deliver the greatest value soonest, an approach first proposed in the 1980s.

Because the iterations are short, the increments are small, such as a few small features or even a few enhancements or bug fixes.

### *Kanban*

Kanban came from an approach to work in manufacturing at Toyota. It is a way of visualizing work and workflow. A Kanban board has columns for different stages of work, from initial idea through development and testing stages to final delivery to users. The tasks are put on sticky notes which are moved from left to right through the columns (like an assembly line for cars).

A key principle of Kanban is to have a limit for work-in-progress activities. If we concentrate on one task, we are much more efficient at doing it, so this approach is less wasteful than trying to do little bits of lots of different tasks. This focus on eliminating waste makes this a lean approach.

There is also a strong focus on user and customer needs. Iterations can be a fixed length to deliver a single feature or enhancement, or features can be grouped together for delivery. Kanban can span more than one team's work (as opposed to Scrum). If user stories are grouped by feature, work may span more than one column on the Kanban board, sometimes referred to as swim lanes.

### *Spiral (or prototyping)*

The Spiral model, initially proposed by Boehm [1996] is based on risk. There are four steps: determine objectives, identify risks and alternatives, develop and test, and plan the next iteration. Prototypes may be developed as a way of addressing risks; these prototypes may be kept and incorporated into later cycles (an incremental approach), they might be discarded (a throw-away prototype), or they may be re-worked as part of the next cycle.

The diagram of the Spiral model shows development starting small from a centre, and moving in a circular way clockwise through the four stages. Each succeeding cycle builds outwards in a spiral through the phases, developing more functionality each time. The key driver for the Spiral model is that it is risk-driven.

### *Agile development*

In this section, we will describe what Agile development is and then cover the changes that this way of working brings to testing. This is additional to what you

need to know for the exam, as the Syllabus does not specifically cover Agile development (the ISTQB Foundation Level Agile Tester Extension Syllabus covers this), but we hope this will give you useful background, especially if you are not familiar with it.

Agile software development is a group of software development methodologies based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. Most Agile teams use Scrum, as described above. Typical Agile teams are 5 to 9 people, and the Agile manifesto describes ways of working that are ideal for small teams, and that counteract problems prevalent in the late 1990s, with its emphasis on process and documentation. The Agile manifesto consists of four statements describing what is valued in this way of working:

- individuals and interactions over processes and tools
- working software over comprehensive documentation
- customer collaboration over contract negotiation
- responding to change over following a plan.

While there are several Agile methodologies in practice, the industry seems to have settled on the use of Scrum as an Agile management approach, and Extreme Programming (XP) as the main source of Agile development ideas. Some characteristics of project teams using Scrum and XP are:

- The generation of business stories (a form of lightweight use cases) to define the functionality, rather than highly detailed requirements specifications.
- The incorporation of business representatives into the development process, as part of each iteration (called a sprint and typically lasting 2 to 4 weeks), providing continual feedback and to define and carry out functional acceptance testing.
- The recognition that we cannot know the future, so changes to requirements are welcomed throughout the development process, as this approach can produce a product that better meets the stakeholders' needs as their knowledge grows over time.
- The concept of shared code ownership among the developers, and the close inclusion of testers in the sprint teams.
- The writing of tests as the first step in the development of a component, and the automation of those tests before any code is written. The component is complete when it then passes the automated tests. This is known as test-driven development.
- Simplicity: building only what is necessary, not everything you can think of.
- The continuous integration and testing of the code throughout the sprint, at least once a day.

Proponents of the Scrum and XP approaches emphasize testing throughout the process. Each iteration (sprint) culminates in a short period of testing, often with an independent tester as well as a business representative. Developers are to write and run test cases for their code, and leading practitioners use tools to automate those tests and to measure structural coverage of the tests (see Chapters 4 and 6). Every time a change is made in the code, the component is tested and then integrated with the existing code, which is then tested using the full set of automated component

test cases. This gives continuous integration, by which we mean that changes are incorporated continuously into the software build.

Agile development provides both benefits and challenges for testers. Some of the benefits are:

- The focus on working software and good quality code.
- The inclusion of testing as part of and the starting point of software development (test-driven development).
- Accessibility of business stakeholders to help testers resolve questions about expected behaviour of the system.
- Self-organizing teams, where the whole team is responsible for quality and gives testers more autonomy in their work.
- Simplicity of design that should be easier to test.

There are also some significant challenges for testers when moving to an Agile development approach:

- Testers who are used to working with well-documented requirements will be designing tests from a different kind of test basis: less formal and subject to change. The manifesto does not say that documentation is no longer necessary or that it has no value, but it is often interpreted that way.
- Because developers are doing more component testing, there may be a perception that testers are not needed. But component testing and confirmation-based acceptance testing by only business representatives may miss major problems. System testing, with its wider perspective and emphasis on non-functional testing as well as end-to-end functional testing is needed, even if it does not fit comfortably into a sprint.
- The tester's role is different: since there is less documentation and more personal interaction within an Agile team, testers need to adapt to this style of working, and this can be difficult for some testers. Testers may be acting more as coaches in testing to both stakeholders and developers, who may not have a lot of testing knowledge.
- Although there is less to test in one iteration than a whole system, there is also a constant time pressure and less time to think about the testing for the new features.
- Because each increment is adding to an existing working system, regression testing becomes extremely important, and automation becomes more beneficial. However, simply taking existing automated component or component integration tests may not make an adequate regression suite.

Software engineering teams are still learning how to apply Agile approaches. Agile approaches cannot be applied to all projects or products, and some testing challenges remain to be surmounted with respect to Agile development. However, Agile methodologies are showing promising results in terms of both development efficiency and quality of the delivered code.

More information about testing in Agile development and iterative incremental models can be found in books by Black [2017], Crispin and Gregory [2008] and Gregory and Crispin [2015]. There is also an ISTQB certificate for the Foundation Level Agile Tester Extension.

## 2.1.2 Software development life cycle models in context

As with many aspects of development and testing, there is no one correct or best life cycle model for every situation. Every project and product is different from others, so it is important to choose a development model that is suitable for your own situation or context. The most suitable development model for you may be based on the following:

- the project goal
- the type of product being developed
- business priorities (for example time to market)
- identified product and project risks.

The development of an internal admin system is not the same as a safety-critical system such as flight control software for aircraft or braking systems for cars. These types of development need different life cycle models in order to succeed. The internal admin system may be developed very informally, with different features delivered incrementally. The development (and testing) of safety-critical systems needs to be far more rigorous and may be subject to legal contracts and regulatory requirements, so a sequential life cycle model may be more appropriate.

It is also important to consider organizational and cultural context. If you want to use Scrum for example, good communication between team members is critical.

The context also determines the test levels and/or test activities that are appropriate for a project, and they may be combined or reorganized. For the integration of a **commercial off-the-shelf (COTS)** software product into a system, for example, a purchaser may perform acceptance testing focused on functionality and other attributes (for example integration to the infrastructure and other systems), followed by a system integration test. The acceptance testing can include testing of system functions, but also testing of quality attributes such as performance and other non-functional tests. The testing may be done from the perspective of the end user and may also be done from an operations point of view.

**Commercial off-the-shelf (COTS)** (off-the-shelf software) A software product that is developed for the general market, i.e. for a large number of customers, and that is delivered to many customers in identical format.

The context within an organization also determines the most suitable life cycle model. For example, the V-model may be used to develop back office systems, so that all new features are integrated and tested before everyone updates to the new system. At the same time, Agile development may be used for the UI (User Interface) to the website, and a Spiral model may be used to develop a new app.

Internet of Things (IoT) systems present special challenges for testing (as well as for development). Many different objects need to be integrated together and tested in realistic conditions, but each object or device may be developed in a different way with a different life cycle model. There is also more emphasis on the later stages of the development life cycle, after objects are actually in use. There may be extensive updates needed to different devices and supporting software systems once users begin using the systems for real. There may also be unforeseen security issues that might require updates. There may even be issues when trying to decommission devices or software for IoT systems. Changing contexts always have an influence on testing, and in our technological world, constant change is definitely the norm, so testing (and development) always needs to adapt to its context.

## 2.2 TEST LEVELS

### SYLLABUS LEARNING OBJECTIVES FOR 2.2 TEST LEVELS (K2)

**FL-2.2.1 Compare the different test levels from the perspective of objectives, test basis, test objects, typical defects and failures, and approaches and responsibilities (K2)**

We have mentioned (and given the definition for) test levels in Section 2.1. The definition of 'test level' is 'an instance of the test process', which is not necessarily the most helpful. The Syllabus here describes test levels as:

groups of test activities that are organized and managed together

The test activities were described in Chapter 1, Section 1.4 (test planning through to test completion). When we talk about test levels, we are looking at those activities performed with reference to development levels (such as those described in the V-model) from components to systems, or even systems of systems.

In this section, we'll look in more detail at the various test levels and show how they are related to other activities within the software development life cycle. 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, leading to more effective and efficient testing. We will look at four test levels in this section.

As we go through this section, watch for the Syllabus terms **acceptance testing**, **alpha testing**, **beta testing**, **component integration testing**, **component testing**, **contractual acceptance testing**, **integration testing**, **operational acceptance testing**, **regulatory acceptance testing**, **system integration testing**, **system testing**, **test basis**, **test case**, **test environment**, **test object**, **test objective** and **user acceptance testing**. These terms are also defined in the Glossary.

While the specific test levels required for – and planned for – a particular project can vary, good practice in testing suggests that each test level has the following clearly identified:

- Specific **test objectives** for the test level.
- The **test basis**, the work product(s) used to derive the test conditions and **test cases**.
- The **test object** (that is, what is being tested such as an item, build, feature or system under test).
- The typical defects and failures that we are looking for at this test level.
- Specific approaches and responsibilities for this test level.

One additional aspect is that each test level needs a **test environment**. Sometimes an environment can be shared by more than one test level: in other situations, a particular environment is needed. For example, acceptance testing should have a test environment that is as similar to production as is possible or feasible.

**Test objective** A reason or purpose for designing and executing a test.

**Test basis** The body of knowledge used as the basis for test analysis and design.

**Test case** A set of preconditions, inputs, actions (where applicable), expected results and postconditions, developed based on test conditions.

**Test object** The component or system to be tested. See also: test item.

**Test environment** (test bed, test rig) An environment containing hardware, instrumentation, simulators, software tools and other support elements needed to conduct a test.

In component testing, developers often just use their development environment. In system testing, an environment may be needed with particular external connections, for example.

When these topics are clearly understood and defined for the entire project team, this contributes to the success of the project. In addition, during test planning, the managers responsible for the test levels should consider how they intend to test a system's configuration, if such data is part of a system.

## 2.2.1 Component testing

**Component testing** (module testing, unit testing) The testing of individual hardware or software components.

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

Component tests are typically based on the requirements and detailed design specifications applicable to the component under test, as well as the code itself (which we'll discuss in Chapter 4 when we talk about white-box testing).

The component under test, the test object, includes the individual components, the data conversion and migration programs used to enable the new release, and database tables, joins, views, modules, procedures, referential integrity and field constraints, and even whole databases.

### *Component testing: objectives*

The different test levels have different objectives. The objectives of component testing include:

- Reducing risk (for example by testing high-risk components more extensively).
- Verifying whether or not functional and non-functional behaviours of the component are as they should be (as designed and specified).
- Building confidence in the quality of the component: this may include measuring structural coverage of the tests, giving confidence that the component has been tested as thoroughly as was planned.
- Finding defects in the component.
- Preventing defects from escaping to later testing.

In incremental and iterative development (for example Agile), automated component regression tests are run frequently, to give confidence that new additions or changes to a component have not caused existing components or links to break.

Component testing may be done in isolation from the rest of the system depending on the context of the development life cycle and the system. Most often, mock objects or stubs and drivers are used to replace the missing software and simulate the interface between the software components in a simple manner. A stub or mock object is called from the software component to be tested; a driver calls a component to be tested (see Figure 2.4). Test harnesses may also be used to provide similar functionality, and service virtualization can give cloud-based functionality to test components in realistic environments.

Component testing may include testing of functionality (for example are the calculations correct) and specific non-functional characteristics such as resource-behaviour (for example memory leaks), performance testing (for example do calculations complete quickly enough), as well as structural testing. Test cases are derived from work products such as the software design or the data model.

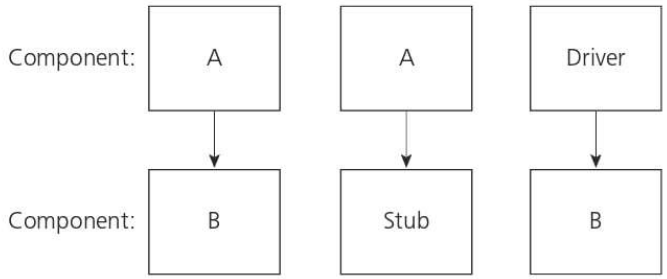


**FIGURE 2.4** Stubs and drivers

### *Component testing: test basis*

What is this particular component supposed to do? Examples of work products that can be used as a test basis for component testing include:

- detailed design
- code
- data model
- component specifications (if available).

### *Component testing: test objects*

What are we actually testing at this level? We could say the smallest thing that can be sensibly tested on its own. Typical test objects for component testing include:

- components themselves, units or modules
- code and data structures
- classes
- database models.

### *Component testing: typical defects and failures*

Examples of defects and failures that can typically be revealed by component testing include:

- incorrect functionality (for example not as described in a design specification)
- data flow problems
- incorrect code or logic.

At the end of the description of all the test levels, see Table 2.1 which summarizes the characteristics of each test level.

### *Component testing: specific approaches and responsibilities*

Typically, component testing occurs with access to the code being tested and with the support of the development environment, such as a unit test framework or debugging tool. In practice it usually involves the developer who wrote the code. The developer may change between writing code and testing it. Sometimes, depending on the applicable level of risk, component testing is carried out by a different developer, introducing independence. Defects are typically fixed as soon as they are found, without formally recording them in a defect management tool. Of course, if such defects are recorded, this can provide useful information for root cause analysis.

One approach in component testing, initially developed in Extreme Programming (XP), is to prepare and automate test cases before coding. This is called a test-first approach or test-driven development (TDD). This approach is highly iterative and is based on cycles of developing automated tests, then building and integrating small pieces of code, and executing the component tests until they pass, and is typically done in Agile development. The idea is that the first thing the developer does is to write some automated tests for the component. Of course if these are run now, they will fail because no code is there! Then just enough code is written until those tests pass. This may involve fixing defects now found by the tests and re-factoring the code. (This approach also helps to build only what is needed rather than a lot of functionality that is not really wanted.)

## 2.2.2 Integration testing

**Integration testing** Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems.

**Component integration testing** (link testing) Testing performed to expose defects in the interfaces and interactions between integrated components.

**System integration testing** Testing the combination and interaction of systems.

**Integration testing** tests interfaces between components and interactions of different parts of a system such as an operating system, file system and hardware or interfaces between systems. Integration tests are typically based on the software and system design (both high-level and low-level), the system architecture (especially the relationships between components or objects) and the workflows or use cases by which the stakeholders will employ the system.

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 components and is done after component testing. It is a good candidate for automation. In iterative and incremental development, both component tests and integration tests are usually part of continuous integration, which may involve automated build, test and release to end users or to a next level. At least, this is the theory. In practice, component integration testing may not be done at all, or misunderstood and as a consequence not done well.
- **System integration testing** tests the interactions between different systems, packages and microservices, and may be done after system testing. System integration testing may also test interfaces to and provided by external organizations (such as web services). In this case, the developing organization may control only one side of the interface, resulting in a number of problems: changes may be destabilizing, defects in the external organization's software may block progress in the testing, or special test environments may be needed. Business processes implemented as workflows may involve a series of systems that can even run on different platforms. System integration testing may be done in parallel with other testing activities.

### *Integration testing: objectives*

The objectives of integration testing include:

- Reducing risk, for example by testing high-risk integrations first.
- Verifying whether or not functional and non-functional behaviours of the interfaces are as they should be, as designed and specified.
- Building confidence in the quality of the interfaces.

- Finding defects in the interfaces themselves or in the components or systems being tested together.
- Preventing defects from escaping to later testing.

Automated integration regression tests (such as in continuous integration) provide confidence that changes have not broken existing interfaces, components or systems.

### *Integration testing: test basis*

How are these components or systems supposed to work together and communicate? Examples of work products that can be used as a test basis for integration testing include:

- software and system design
- sequence diagrams
- interface and communication protocol specifications
- use cases
- architecture at component or system level
- workflows
- external interface definitions.

### *Integration testing: test objects*

What are we actually testing at this level? The emphasis here is in testing things with others which have already been tested individually. We are interested in how things work together and how they interact. Typical test objects for integration testing include:

- subsystems
- databases
- infrastructure
- interfaces
- APIs (Application Programming Interfaces)
- microservices.

### *Integration testing: typical defects and failures*

Examples of defects and failures that can typically be revealed by component integration testing include:

- Incorrect data, missing data or incorrect data encoding.
- Incorrect sequencing or timing of interface calls.
- Interface mismatch, for example where one side sends a parameter where the value exceeds 1,000, but the other side only expects values up to 1,000.
- Failures in communication between components.
- Unhandled or improperly handled communication failures between components.
- Incorrect assumptions about the meaning, units or boundaries of the data being passed between components.

Examples of defects and failures that can typically be revealed by system integration testing include:

- inconsistent message structures between systems
- incorrect data, missing data or incorrect data encoding
- interface mismatch
- failures in communication between systems
- unhandled or improperly handled communication failures between systems
- incorrect assumptions about the meaning, units or boundaries of the data being passed between systems
- failure to comply with mandatory security regulations.

At the end of the description of all the test levels, see Table 2.1 which summarizes the characteristics of each test level.

### *Integration testing: specific approaches and responsibilities*

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 components or systems are integrated simultaneously, after which everything is tested as a whole. This is called big-bang integration. Big-bang integration 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 planning 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 down the integration test process.

Another extreme is that all programs are integrated one by one, and tests are 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 mock objects or stubs and drivers may have to be developed and used in the test. Within incremental integration testing a range of possibilities exist, partly depending on the system architecture:

- Top-down: testing starts from the top and works to the bottom, following the control flow or architectural structure (for example starting from the GUI or main menu). Components or systems are substituted by stubs.
- Bottom-up: testing reverses this approach, starting 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 the most problems. Doing so prevents major defects at the end of the integration test stage. In order to reduce the risk of late defect discovery, integration should normally be incremental rather than big bang. Ideally, testers should understand the architecture

and influence integration planning. If integration tests are planned before components or systems are built, they can be developed in the order required for most efficient testing. A risk analysis of the most complex interfaces can help to focus integration testing. In iterative and incremental development, integration is also incremental. Existing integration tests should be part of the regression tests used in continuous integration. Continuous integration has major benefits because of its iterative nature.

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. In integrating system X with system Y, again the focus is on the communication between the systems and what can be done by both systems together, rather than defects in the individual systems. Both functional and structural approaches may be used. Testing of specific non-functional characteristics (for example performance) may also be included in integration testing.

Component integration testing is often carried out by developers; system integration testing is generally the responsibility of the testers. Either type of integration testing could be done by a separate team of specialist integration testers, or by a specialist group of developers/integrators, including non-functional specialists. The testers performing the system integration testing need to understand the system architecture. Ideally, they should have had an influence on the development, integration planning and integration testing.

## 2.2.3 System testing

**System testing** is concerned with the behaviour of the whole system/product as defined by the scope of a development project or product. It may include tests based on risk analysis reports, system, functional or software requirements specifications, business processes, use cases or other high-level descriptions of system behaviour, interactions with the operating system and system resources. The focus is on end-to-end tasks that the system should perform, including non-functional aspects, such as performance.

**System testing** Testing an integrated system to verify that it meets specified requirements. (Note that the ISTQB definition implies that system testing is only about verification of specified requirements. In practice, system testing is often also about validation that the system is suitable for its intended users, as well as verifying against any type of requirement.)

In some systems, the quality of the data may be of critical importance, so there would be a focus on data quality. System level tests may be automated to provide a regression suite to ensure that changes have not adversely affected existing system functionality. Stakeholders may use the information from system testing to decide whether the system is ready for user acceptance testing, for example. System testing is also where conformance to legal or regulatory requirements or to external standards is tested.

The test environment is important for system testing; it should correspond to the final production environment as much as possible.

### *System testing: objectives*

The objectives of system testing include:

- reducing risk
- verifying whether or not functional and non-functional behaviours of the system are as they should be (as specified)
- validating that the system is complete and will work as it should and as expected
- building confidence in the quality of the system as a whole
- finding defects
- preventing defects from escaping to later testing or to production.

### System testing: test basis

What should the system as a whole be able to do? Examples of work products that can be used as a test basis for system testing include:

- software and system requirement specifications (functional and non-functional)
- risk analysis reports
- use cases
- epics and user stories
- models of system behaviour
- state diagrams
- system and user manuals.

### System testing: test objects

What are we actually testing at this level? The emphasis here is in testing the whole system, from end to end, encompassing everything that the system needs to do (and how well it should do it, so non-functional aspects are also tested here). Typical test objects for integration testing include:

- applications
- hardware/software systems
- operating systems
- system under test (SUT)
- system configuration and configuration data.

### System testing: typical defects and failures

Examples of defects and failures that can typically be revealed by system testing include:

- incorrect calculations
- incorrect or unexpected system functional or non-functional behaviour
- incorrect control and/or data flows within the system
- failure to properly and completely carry out end-to-end functional tasks
- failure of the system to work properly in the production environment(s)
- failure of the system to work as described in system and user manuals.

At the end of the description of all the test levels, see Table 2.1 which summarizes the characteristics of each test level.

### System testing: specific approaches and responsibilities

System testing is most often the final test on behalf of development to verify that the system to be delivered meets the specification and to validate that it meets expectations; one of its purposes is to find as many defects as possible. Most often it is carried out by specialist testers that form a dedicated, and sometimes independent, 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 independence 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 end-to-end behaviour of both functional and non-functional aspects of the system. An end-to-end test may include all of the steps in a typical transaction, from logging on, accessing data, placing an order, etc. through to logging off and checking order status in a database. Typical non-functional tests include performance, security and reliability. Testers may also need to deal with incomplete or undocumented requirements. System testing of functional requirements starts by using the most appropriate black-box techniques for the aspect of the system to be tested. For example, a decision table may be created for combinations of effects described in business rules. White-box techniques may also be used to assess the thoroughness of testing elements such as menu dialogue structure or web page navigation (see Chapter 4 for more on test techniques).

System testing requires a controlled test environment with regard to, among 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) environment. 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.

System testing is often carried out by independent testers, for example an internal test team or external testing specialists. However, if testers are only brought in when system test execution is about to start, you will miss a lot of opportunities to save time and money, as well as aggravation. If there are defects in specifications, such as missing functions or incorrect descriptions of business processes, these may not be picked up before the system is built. Because many defects result from misunderstandings, the discussions (indeed arguments) about them tend to be worse the later they are discovered. The developers will defend their understanding because that is what they have built. The independent testers or end-users may realize that what was built was not what was wanted. This situation can lead to defects being missed in testing (if they are based on wrong specifications) or things being reported as defects that actually are not (due to misunderstandings). These are known as false negative and false positive results respectively. Referring back to testing Principle 3, early test involvement saves time and money, so have testers involved in user story refinement and static testing such as reviews.

## 2.2.4 Acceptance testing

When the development organization has performed its system test (and possibly also system integration tests) and has corrected all or most defects, the system may be delivered for **acceptance testing**. Acceptance tests typically produce information to assess the system's readiness for release or deployment to end-users or customers. Although defects are found at this level, that is not the main aim of acceptance testing. (If lots of defects are found at this late stage, there are serious problems with the whole system, and major project risks.) The focus is on validation, the use of the system for real, how suitable the system is to be put into production or actual use by its intended users. Regulatory and legal requirements, and conformance to standards may also be checked in acceptance testing, although they should also have been addressed in an earlier level of testing, so that the acceptance test is confirming compliance to the standards.

**Acceptance testing**
Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.

### Acceptance testing: objectives

The objectives of acceptance testing include:

- establishing confidence in the quality of the system as a whole
- validating that the system is complete and will work as expected
- verifying that functional and non-functional behaviours of the system are as specified.

### Different forms of acceptance testing

Acceptance testing is quite a broad category, and it comes in several different flavours or forms. We will look at four of these.

**User acceptance testing** Acceptance testing conducted in a real or simulated operational environment by intended users focusing on their needs, requirements and business processes.

**Operational acceptance testing** (production acceptance testing) Operational testing in the acceptance test phase, typically performed in a (simulated) operational environment by operations and/or systems administration staff focusing on operational aspects, for example recoverability, resource-behaviour, installability and technical compliance.

**Contractual acceptance testing** Acceptance testing conducted to verify whether a system satisfies its contractual requirements.

**Regulatory acceptance testing** Acceptance testing conducted to verify whether a system conforms to relevant laws, policies and regulations.

#### *User acceptance testing (UAT)*

**User acceptance testing** is exactly what it says. It is acceptance testing done by (or on behalf of) users, that is, end-users. The focus is on making sure that the system is really fit for purpose and ready to be used by real intended users of the system. The UAT can be done in the real environment or in a simulated operational environment (but as realistic as possible). The aim of testing here is to build confidence that the system will indeed enable the users to do what they need to do in an efficient way. The system needs to fulfil the requirements and meet their needs. The users focus on their business processes, which they should be able to perform with a minimum of difficulty, cost and risk.

#### *Operational acceptance testing (OAT)*

**Operational acceptance testing** focuses on operations and may be performed by system administrators. The main purpose is to give confidence to the system administrators or operators that they will be able to keep the system running, and recover from adverse events quickly and without additional risk. It is normally performed in a simulated production environment and is looking at operational aspects, such as:

- testing of backups and restoration of backups
- installing, uninstalling and upgrading
- disaster recovery
- user management
- maintenance tasks
- data loading and migration tasks
- checking for security vulnerabilities (for example ethical hacking)
- performance and load testing.

#### *Contractual and regulatory acceptance testing*

If a system has been custom-developed for another company, there is normally a legal contract describing the responsibilities, schedule and costs of the project. The contract should also include or refer to acceptance criteria for the system, which should have been defined and agreed when the contract was first taken out. Having agreed the acceptance criteria in advance, **contractual acceptance testing** is focused on whether or not the system meets those criteria. This form of testing is often performed by users or independent testers.

**Regulatory acceptance testing** is focused on ensuring that the system conforms to government, legal or safety regulations. This type of testing is also often performed by

independent testers. It may be a requirement to have a representative of the regulatory body present to witness or to audit the tests.

For both of these forms of acceptance testing, the aim is to build confidence that the system is in conformance with the contract or regulations.

### *Alpha and beta testing*

Alpha and beta testing are typically used for COTS software, such as software packages that can be bought or downloaded by consumers. Feedback is needed from potential or existing users in their market before the software product is put out for sale commercially. The testing here is looking for feedback (and defects) from real users and customers or potential customers. Sometimes free software is offered to those who volunteer to do beta testing.

The difference between alpha and beta testing is only in where the testing takes place. **Alpha testing** is at the company that developed the software, and beta testing is done in the users' own offices or homes. In alpha testing, a cross-section of potential users 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. Alpha testing is normally mentioned first, but these two forms can be done in any order, or only one could be done (or none).

**Alpha testing**
Simulated or actual operational testing conducted in the developer's test environment, by roles outside the development organization.

**Beta testing** (field testing) Simulated or actual operational testing conducted at an external site, by roles outside the development organization.

**Beta testing** sends the system or software package out to a cross-section of users who install it and use it under real-world working conditions. The users send records of defects with the system to the development organization, where the defects are repaired. Beta testing is more visible, and is increasingly popular to be done remotely. For example, crowd testing, where people or potential users from all over the world remotely test an application, can be a form of beta testing. One of the advantages of beta testing is that different users will have a great variety of different environments (browsers, other software, hardware configurations, etc.), so the testing can cover many more combinations of factors.

## *Acceptance testing: test basis*

How do we know that the system is ready to be used for real? Examples of work products that can be used as a test basis for the various forms of acceptance testing include:

- business processes
- user or business requirements
- regulations, legal contracts and standards
- use cases
- system requirements
- system or user documentation
- installation procedures
- risk analysis reports.

For operational acceptance testing (OAT), there are some additional aspects with specific work products that can be a test basis:

- backup and restore/recovery procedures
- disaster recovery procedures
- non-functional requirements
- operations documentation

- deployment and installation instructions
- performance targets
- database packages
- security standards or regulations.

Note that it is particularly important to have very clear, well-tested and frequently rehearsed procedures for disaster recovery and restoring backups. If you are in the situation of having to perform these procedures, then you may be in a state of panic, since something serious will have already gone wrong. In that psychological state, it is very easy to make mistakes, and here mistakes could be disastrous. There are stories of organizations who compounded one disaster by accidentally deleting their backups, or who find that their backups are unusable or incomplete! This is why restoring from backups is an important test to do regularly.

### *Acceptance testing: test objects*

What are we actually testing at this level? The emphasis here is in gaining confidence, based on the particular form of acceptance testing: user confidence, confidence in operations, confidence that we have met legal or regulatory requirements, and confidence that real users will like and be happy with the software we are selling. Some of the things we are testing are similar to the test objects of system testing. Typical test objects for acceptance testing include:

- system under test (SUT)
- system configuration and configuration data
- business processes for a fully integrated system
- recovery systems and hot sites (for business continuity and disaster recovery testing)
- operational and maintenance processes
- forms
- reports
- existing and converted production data.

### *Acceptance testing: typical defects and failures*

Examples of defects and failures that can typically be revealed by acceptance testing include:

- system workflows do not meet business or user requirements
- business rules are not implemented correctly
- system does not satisfy contractual or regulatory requirements
- non-functional failures such as security vulnerabilities, inadequate performance efficiency under high load, or improper operation on a supported platform.

At the end of the description of all the test levels, see Table 2.1 which summarizes the characteristics of each test level.

### *Acceptance testing: specific approaches and responsibilities*

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 stakeholders may be involved as well. The execution of the acceptance test requires a test environment that is, for most aspects, representative of the production environment ('as-if production').

The goal of acceptance testing is to establish confidence in the system, part of the system or specific non-functional characteristics, for example usability of the system. Acceptance testing is most often focused on a validation type of testing, where 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 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 component testing.
- Acceptance testing of a new functional enhancement may come before system testing.

User acceptance testing 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 organized 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 meets its exit criteria from the system test can start while another subsystem may still be in the system test phase. In most organizations, system administration will perform the operational acceptance test shortly before the system is released. The operational acceptance test may include testing of backup/restore, data load and migration tasks, disaster recovery, user management, maintenance tasks and periodic check of security vulnerabilities.

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.

In iterative development, different forms of acceptance testing may be done at various times, and often in parallel. At the end of an iteration, a new feature may be tested to validate that it meets stakeholder and user needs. This is user acceptance testing. If software for general release (COTS) is being developed, alpha and beta testing may be used at or near the end of an iteration or set of iterations. Operational and regulatory acceptance testing may also occur at the end of an iteration or set of iterations.

### *Test level characteristics: summary*

Table 2.1 summarizes the characteristics of the different test levels: the test basis, test objects and typical defects and failures. We have covered these in the various sections, but it is useful to contrast them in order to distinguish them from each other. We have omitted some of the detail to make the table easier to take in at a glance. Note that some of the typical defects for integration testing are only for system integration testing (SIT).

**TABLE 2.1** Test level characteristics

| | Component testing | Integration testing | System testing | Acceptance testing |
|---|---|---|---|---|
| **Objectives** | reduce risk<br>verify functional and non-functional behaviour<br>build confidence in components<br>find defects<br>prevent defects to higher levels | reduce risk<br>verify functional and non-functional behaviour<br>build confidence in interfaces<br>find defects<br>prevent defects to higher levels | reduce risk<br>verify functional and non-functional behaviour<br>validate completeness, works as expected<br>build confidence in whole system<br>find defects<br>prevent defects to higher levels | establish confidence in whole system and its use<br>validate completeness, works as expected<br>verify functional and non-functional behaviour |
| **Test basis** | detailed design<br>code<br>data models<br>component specifications | software/system design<br>sequence diagrams<br>interface and communication protocol specs<br>use cases<br>architecture (component or system)<br>workflows<br>external interface definitions | requirement specs (functional and non-functional)<br>risk analysis reports<br>use cases<br>epics and user stories<br>models of system behaviour<br>state diagrams<br>system and user manuals | business processes<br>user, business, system requirements<br>regulations, legal contracts and standards<br>use cases<br>documentation<br>installation procedures<br>risk analysis |

| | Component testing | Integration testing | System testing | Acceptance testing |
|---|---|---|---|---|
| **Test objects** | components, units, modules<br>code<br>data structures<br>classes<br>database models | subsystems<br>databases<br>infrastructure<br>interfaces<br>APIs<br>microservices | applications<br>hardware/software<br>operating systems<br>system under test<br>system configuration and data | system under test (SUT)<br>system configuration and data<br>business processes<br>recovery systems<br>operation and maintenance processes<br>forms<br>reports<br>existing and converted production data |
| **Typical defects and failures** | wrong functionality<br>data flow problems<br>incorrect code/logic | data problems<br>inconsistent message structure (SIT)<br>timing problems<br>interface mismatch<br>communication failures<br>incorrect assumptions<br>not complying with regulations (SIT) | incorrect calculations<br>incorrect or unexpected behaviour<br>incorrect data/control flows<br>cannot complete end-to-end tasks<br>does not work in production environment(s)<br>not as described in manuals/ documentation | system workflows do not meet business or user needs<br>business rules not correct<br>contractual or regulatory problems<br>non-functional failures (performance, security) |

## 2.3 TEST TYPES

### SYLLABUS LEARNING OBJECTIVES FOR 2.3 TEST TYPES (K2)

**FL-2.3.1** **Compare functional, non-functional, and white-box testing (K2)**

**FL-2.3.2** **Recognize that functional, non-functional and white-box tests occur at any test level (K1)**

**FL-2.3.3** **Compare the purposes of confirmation testing and regression testing (K2)**

In this section, we'll look at different test types. We'll discuss tests that focus on the functionality of a system, which informally is *testing what the system does*. We'll also discuss tests that focus on non-functional attributes of a system, which informally is *testing how well the system does what it does*. We'll introduce testing based on the system's structure. Finally, we'll look at testing of changes to the system, both confirmation testing (testing that the changes succeeded) and regression testing (testing that the changes did not affect anything unintentionally).

The test types discussed here can involve the development and use of a model of the software or its behaviours. Such models can occur in structural testing when we use control flow models or menu structure models. Such models in non-functional testing can involve performance models, usability models and security threat models. They can also arise in functional testing, such as the use of process flow models, state transition models or plain language specifications. Examples of such models will be found in Chapter 4.

As we go through this section, watch for the Syllabus terms **functional testing**, **non-functional testing**, **test type** and **white-box testing**. You will find these terms defined in the Glossary as well.

Test types are introduced as a means of clearly defining the objective of a certain test level for a program or project. We need to think about different 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 make it easier to make and communicate decisions about test objectives. Typical objectives may include:

- Evaluating functional quality, for example whether a function or feature is complete, correct and appropriate.
- Evaluating non-functional quality characteristics, for example reliability, performance efficiency, security, compatibility and usability.
- Evaluating whether the structure or architecture of the component or system is correct, complete and as specified.
- Evaluating the effects of changes, looking at both the changes themselves (for example defect fixes) and also the remaining system to check for any unintended side-effects of the change. These are confirmation testing and regression testing, respectively, and are discussed in Section 2.3.4.

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, that is, confirming that defects have been fixed (confirmation testing, or re-testing) and looking for unintended changes (regression testing). Depending on its objectives, testing will be organized differently. For example, component testing aimed at performance would be quite different from component testing aimed at achieving decision coverage.

**Test type** A group of test activities based on specific test objectives aimed at specific characteristics of a component or system.

## 2.3.1 Functional testing

The function of a system (or component) is what it does. This is typically described in work products such as business requirements specifications, functional specifications, use cases, epics or user stories. There may be some functions that are assumed to be provided that are not documented. They are also part of the requirements 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 (for example tests for components may be based on a component specification).

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

**Functional testing** Testing conducted to evaluate the compliance of a component or system with functional requirements.

Functional testing can also be done focusing on suitability, interoperability testing, security, accuracy and compliance. Security testing, for example, investigates the functions (for example a firewall) relating to detection of threats, such as viruses, from malicious outsiders.

Testing of functionality could be done from different perspectives, the two main ones being requirements-based or business-process-based.

Requirements-based testing uses a specification of the functional requirements 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 specification) and use this to prioritize the tests. This will ensure that the most important 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 business 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. User scenarios originate from object-oriented development but are nowadays popular in many development 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 experience-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 design, a model may be developed, such as a process model, state transition model or a plain-language specification.

The thoroughness of functional testing can be measured by a coverage measure based on elements of the function that we can list. For example, we can list all of the options available from every pull-down menu. If our set of tests has at least one test for each option, then we have 100% coverage of these menu options. Of course, that does not mean that the system or component is 100% tested, but it does mean that we have at least touched every one of the things we identified. When we have traceability between our tests and functional requirements, we can identify which requirements we have not yet covered, that is, have not yet tested (coverage gaps). For example, if we covered only 90% of the menu options, we could add tests so that the untested 10% are then covered.

Special skills or knowledge may be needed for functional testing, particularly for specialized application domains. For example, medical device software may need medical knowledge both for the design and testing of such systems. The worst thing that a heart pacemaker can do is not to stop giving the electrical stimulant to the heart (the heart may still limp along less efficiently). The worst thing is to speed up, giving the signal much too frequently; this can be fatal. Other specialized application areas include gaming or interactive entertainment systems, geological modelling for oil and gas exploration, or automotive systems.

## 2.3.2 Non-functional testing

This test type 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** Testing conducted to evaluate the compliance of a component or system with non-functional requirements.

**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, portability testing and security 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 characteristics keep on reappearing, although their place in the hierarchy can differ. The International Organization for Standardization (ISO) has defined a set of quality characteristics in ISE/IEC 25010 [2011].

A common misconception is that non-functional testing occurs only during higher levels of testing such as system test, system integration test and acceptance test. In fact, non-functional testing may be performed at all test levels; the higher the level of risk associated with each type of non-functional testing, the earlier in the life cycle it should occur. Ideally, non-functional testing involves tests that quantifiably measure characteristics of the systems and software. For example, in performance testing we can measure transaction throughput, resource utilization and response times. Generally, non-functional testing defines expected results in terms of the external behaviour of the software. This means that we typically use black-box test techniques. For example, we could use boundary value analysis to

define the stress conditions for performance tests, and equivalence partitioning to identify types of devices for compatibility testing, or to identify user groups for usability testing (novice, experienced, age range, geographical location, educational background).

The thoroughness of non-functional testing can be measured by the coverage of non-functional elements. If we had at least one test for each major group of users, then we would have 100% coverage of those user groups that we had identified. Of course, we may have forgotten an important user group, such as those with disabilities, so we have only covered the groups we have identified.

If we have traceability between non-functional tests and non-functional requirements, we may be able to identify coverage gaps. For example, an implicit requirement is for accessibility for disabled users.

Special skills or knowledge may be needed for non-functional testing, such as for performance testing, usability testing or security testing (for example for specific development languages).

More about non-functional testing is found in other ISTQB qualification Syllabuses, including the Advanced Test Analyst, the Advanced Technical Test Analyst, and the Advanced Security Tester, the Foundation Performance Testing, and the Foundation Usability Testing Syllabus.

## 2.3.3 White-box testing

The third test type looks at the internal structure or implementation of the system or component. If we are talking about the structure of a system, we may call it the system architecture. Structural elements also include the code itself, control flows, business processes and data flows. **White-box testing** is also referred to as structural testing or glass-box because we are interested in what is happening inside the box.

**White-box testing** (clear-box testing, code-based testing, glass-box testing, logic-coverage testing, logic-driven testing, structural testing, structure-based testing) Testing based on an analysis of the internal structure-based of the component or system.

White-box 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 it is true to say that it tends to be mostly applied at component testing and component integration testing, 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 hierarchy or the interfaces between components (the interfaces themselves can be listed as coverage items). The test basis for system, system integration or acceptance testing could be a business model, for example business rules.

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 (for example statements or decision outcomes) that have been exercised (that is, they have been 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 and white-box test techniques are covered in Chapter 4.)

Special skills or knowledge may be needed for white-box testing, such as knowledge of the code (to interpret coverage tool results) or how data is stored (for database queries).

## 2.3.4 Change-related testing

The final test type is the testing of changes. This category is slightly different to the others because if you have made a change to the software, you will have changed the way it functions, how well it functions (or both) and its structure. 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. There are two things to be particularly aware of when changes are made: the change itself and any other effects of the change.

### *Confirmation testing (re-testing)*

**Confirmation testing** (re-testing) Dynamic testing conducted after fixing defects with the objective to confirm that failures caused by those defects do not occur anymore.

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 steps leading up to the failure are carried out in exactly the same way as described in the defect report, using the same inputs, data and environment, and possibly extending beyond the test to ensure that the change has indeed fixed all of the problems due to the defect. 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 introduced 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*

**Regression testing** Testing of a previously tested component or system following modification to ensure that defects have not been introduced or have been uncovered in unchanged areas of the software as a result of the changes made.

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 make sure (as far as is practical) 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 (certainly 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.). In some cases, all of the test cases in a regression test suite would be executed every time a new version of software is produced; this makes them ideal candidates for automation. However, it is much better to be able to select subsets for execution, especially if the regression test suite is very large. In Agile development, a selection of regression tests would be run to meet the objectives of a particular iteration. Automation of regression tests should start as early as possible in the project. See Chapter 6 for more on test automation.

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 the host operating system is introduced or the production environment has a new version of the Java Virtual Machine or anti-malware software.

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. 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 considering 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 possible and desirable to eliminate some test cases from a large regression test suite, for example if they are repetitive (tests which exercise the same conditions) or can be combined (if they are always run together). Another approach is to eliminate test cases when the risk associated with that test is so low that it is not worth running it anymore.

Both confirmation testing and regression testing are done at all test levels.

In iterative and incremental development, changes are more frequent, even continuous and the software is refactored frequently. This makes confirmation testing and regression testing even more important. But iterative development such as Agile should also include continuous testing, and this testing is mainly regression testing. For IoT systems, change-related testing covers not only software systems but the changes made to individual objects or devices, which may be frequently updated or replaced.

## 2.3.5 Test types and test levels

We mentioned as we went through the test types that each test type is applicable at every test level. The testing is different, depending on the test level and test type, of course, but the Syllabus gives examples of each test type at each test level to illustrate the point.

### *Functional tests at each test level*

Let's use a banking example to look at the different levels of testing. There are many features in a financial application. Some of them are visible to users and others are behind the scenes but equally important for the whole application to work well. The more technical and more detailed aspects should be tested at the lower levels, and the customer-facing aspects at higher levels. We will also look at examples of the different test types showing the different testing for functional, non-functional, white-box and change-related testing.

- Component testing: how the component should calculate compound interest.
- Component integration testing: how account information from the user interface is passed to the business logic.
- System testing: how account holders can apply for a line of credit.

- System integration testing: how the system uses an external microservice to check an account holder's credit score.
- Acceptance testing: how the banker handles approving or declining a credit application.

### *Non-functional tests at each test level*

- Component testing: the time or number of CPU cycles to perform a complex interest calculation.
- Component integration testing: checking for buffer overflow (a security flaw) from data passed from the user interface to the business logic.
- System testing: portability tests on whether the presentation layer works on supported browsers and mobile devices.
- System integration testing: reliability tests to evaluate robustness if the credit score microservice does not respond.
- Acceptance testing: usability tests for accessibility of the banker's credit processing interface for people with disabilities.

### *White-box tests at each test level*

- Component testing: achieve 100% statement and decision coverage for all components performing financial calculations.
- Component integration testing: coverage of how each screen in the browser interface passes data to the next screen and to the business logic.
- System testing: coverage of sequences of web pages that can occur during a credit line application.
- System integration testing: coverage of all possible inquiry types sent to the credit score microservice.
- Acceptance testing: coverage of all supported financial data file structures and value ranges for bank-to-bank transfers.

### *Change-related tests at each test level*

- Component testing: automated regression tests for each component are included in the continuous integration framework and pipeline.
- Component integration testing: confirmation tests for interface-related defects are activated as the fixes are checked into the code repository.
- System testing: all tests for a given workflow are re-executed if any screen changes.
- System integration testing: as part of continuous deployment of the credit scoring microservice, automated tests of the interactions of the application with the microservice are re-executed.
- Acceptance testing: all previously failed tests are re-executed after defects found in acceptance testing are fixed.

Note that not every test type will occur at every test level in every system! However, it is a good idea to think about how every test type might apply at each test level, and try to implement those tests at the earliest opportunity within the development life cycle.

## 2.4 MAINTENANCE TESTING

### SYLLABUS LEARNING OBJECTIVES FOR 2.4 MAINTENANCE TESTING (K2)

**FL-2.4.1 Summarize triggers for maintenance testing (K2)**

**FL-2.4.2 Describe the role of impact analysis in maintenance testing (K2)**

Once deployed, a system is often in service for years or even decades. During this time, the system and its operational environment are often corrected, changed or extended. As we go through this section, watch for the Syllabus terms **impact analysis** and **maintenance testing**. You will find these terms also defined in the Glossary.

**Maintenance testing**
Testing the changes to an operational system or the impact of a changed environment to an operational system.

Testing that is executed during this life cycle phase is called **maintenance testing**. Maintenance testing, along with the entire process of maintenance releases, should be carefully planned. Not only must planned maintenance releases be considered, but the process for developing and testing hot fixes must be as well. Maintenance testing includes any type of testing of changes to an existing, operational system, whether the changes result from modifications, migration or retirement of the software or system.

Modifications can result from planned enhancement changes such as those referred to as minor releases, that include new features and accumulated (non-emergency) bug fixes. Modifications can also result from corrective and more urgent emergency changes. Modifications can also involve changes of environment, such as planned operating system or database upgrades, planned upgrade of COTS software, or patches to correct newly exposed or discovered vulnerabilities of the operating system.

Migration involves moving from one platform to another. This can involve abandoning a platform no longer supported or adding a new supported platform. Either way, testing must include operational tests of the new environment as well as of the changed software. Migration testing can also include conversion testing, where data from another application will be migrated into the system being maintained.

Note that maintenance testing is different from testing for maintainability (which is the degree to which a component or system can be modified by the intended maintainers). In this section, we'll discuss maintenance testing.

The same test process steps will apply as for testing during development 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. If testing is done more formally, an application for a change may be used to produce a test plan for testing the change, with test cases changed or created as needed. In less formal testing, thought needs to be given to how the change should be tested, even if this planning, updating of test cases and execution of the tests is part of a continuous process.

The scope of maintenance testing depends on several factors, which influence the test types and test levels. The factors are:

- Degree of risk of the change, for example a self-contained change is a lower risk than a change to a part of the system that communicates with other systems.

- The size of the existing system, for example a small system would need less regression testing than a larger system.
- The size of the change, which affects the amount of testing of the changes that would be needed. The amount of regression testing is more related to the size of the system than the size of the change.

## 2.4.1 Triggers for maintenance

As stated, maintenance testing is done on an existing operational system. There are three possible triggers for maintenance testing:

- modifications
- migration
- retirement.

Modifications include planned enhancement changes (for example release-based), corrective and emergency changes and changes of environment, such as planned operating system or database upgrades, or patches to newly exposed or discovered vulnerabilities of the operating system and upgrades of COTS software.

Modifications may also be of hardware or devices, not just software components or systems. For example, in IoT systems, new or significantly modified hardware devices may be introduced to a working system. The emphasis in maintenance testing would likely focus on different types of integration testing and security testing at all test levels.

Maintenance testing for migration (for example from one platform to another) should also include operational testing of the new environment, as well as the changed software. It is important to know that the platform you will be transferring to is sound before you start migrating your own files and applications.

Maintenance testing for the retirement of a system may include the testing of data migration or archiving, if long data-retention periods are required. Testing of restore or retrieve procedures after archiving may also be needed. There is no point in trying to save and preserve something that you can no longer access. These procedures should be regularly tested and action taken to migrate away from technology that is reaching the end of its life. You may remember seeing magnetic tape on old movies, which was thought to be a good long-term archiving solution at the time.

## 2.4.2 Impact analysis and regression testing

As mentioned earlier, maintenance testing usually consists of two parts:

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

**Impact analysis** The identification of all work products affected by a change, including an estimate of the resources needed to accomplish the change.

In addition to testing what has been changed, maintenance testing includes extensive regression testing to parts of the system that have not been changed. Some systems will have extensive regression suites (automated or not) where the costs of executing all of the tests would be significant. A major and important activity within maintenance testing is **impact analysis**. During impact analysis, together with stakeholders, a decision is made on what parts of the system may be unintentionally

affected and therefore need more extensive 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, so this gives us the best value for the time and effort we can spend in regression testing.

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 specified and tests would be drawn up just as for the development. Do not forget that automated regression tests will also need to be updated in line with the changes; this can take significant effort, depending on the architecture of your automation (see Chapter 6).

Impact analysis can also be used to help make a decision about whether or not a particular change should be made. If the change has the potential to cause high-risk vulnerabilities throughout the system, it may be a better decision not to make that change.

There are a number of factors that make impact analysis more difficult:

- Specifications are out of date or missing (for example business requirements, user stories, architecture diagrams).
- Test cases are not documented or are out of date.
- Bi-directional traceability between tests and the test basis has not been maintained.
- Tool support is weak or non-existent.
- The people involved do not have domain and/or system knowledge.
- The maintainability of the software has not been taken into enough consideration during development.

Impact analysis can be very useful in making maintenance testing more efficient, but if it is not, or cannot be, done well, then the risks of making the change are greatly increased.

## CHAPTER REVIEW

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

From Section 2.1, you should now understand the relationship between development activities and test activities within a development life cycle and be familiar with sequential life cycle models (waterfall and V-model) and iterative/incremental life cycle models (RUP, Scrum, Kanban and Spiral). 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 be able to give reasons why software development life cycle models need to be adapted to the context of the project and product being developed. You should know the Glossary terms **commercial off-the-shelf (COTS), sequential development model** and **test level**.

From Section 2.2, you should know the typical levels of testing (component, integration, system and acceptance testing). You should be able to compare the different levels of testing with respect to their major objectives, the test basis, typical objects of testing, typical defects and failures, and approaches and responsibilities for each test level. You should know the Glossary terms **acceptance testing**, **alpha testing**, **beta testing**, **component integration testing**, **component testing**, **contractual acceptance testing**, **integration testing**, **operational acceptance testing**, **regulatory acceptance testing**, **system integration testing**, **system testing**, **test basis**, **test case**, **test environment**, **test object**, **test objective** 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 **functional testing**, **non-functional testing**, **test type** 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** Which of the following statements is true?

a. Overlapping test levels and test activities are more common in sequential life cycle models than in iterative incremental models.

b. The V-model is an iterative incremental life cycle model because each development activity has a corresponding test activity.

c. When completed, iterative incremental life cycle models are more likely to deliver the full set of features originally envisioned by stakeholders than sequential models.

d. In iterative and incremental life cycle models, delivery of usable software to end-users is much more frequent than in sequential models.

**Question 2** What level of testing is typically performed by system administration staff?

a. Regulatory acceptance testing.

b. System testing.

c. System integration testing.

d. Operational acceptance testing.

**Question 3** Which of the following is a test type?

a. Component testing.

b. Functional testing.

c. System testing.

d. Acceptance testing.

**Question 4** Consider the three triggers for maintenance, and match the event with the correct trigger:

1. Data conversion from one system to another.
2. Upgrade of COTS software.
3. Test of data archiving.
4. System now runs on a different platform and operating system.
5. Testing restore or retrieve procedures.
6. Patches for security vulnerabilities.

a. Modification: 2 and 3, Migration: 1 and 5, Retirement: 4 and 6.

b. Modification: 2 and 6, Migration: 1 and 4, Retirement: 3 and 5.

c. Modification: 2 and 4, Migration: 1 and 3, Retirement: 5 and 6.

d. Modification: 1 and 5, Migration: 2 and 3, Retirement: 4 and 6.

**Question 5** Which of these is a functional test?

a. Measuring response time on an online booking system.

b. Checking the effect of high volumes of traffic in a call centre system.

c. Checking the online bookings screen information and the database contents against the information on the letter to the customers.

d. Checking how easy the system is to use, particularly for users with disabilities such as impaired vision.

**Question 6** Which of the following is true, regarding the process of testing emergency fixes?

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 function 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 developer'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

Static 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, for example on code.

## 3.1 STATIC TECHNIQUES AND THE TEST PROCESS

### SYLLABUS LEARNING OBJECTIVES FOR 3.1 STATIC TECHNIQUES AND THE TEST PROCESS (K2)

**FL-3.1.1** **Recognize types of software work product that can be examined by the different static testing techniques (K1)**

**FL-3.1.2** **Use examples to describe the value of static testing (K2)**

**FL-3.1.3** **Explain the difference between static and dynamic techniques, considering objectives, types of defects to be identified, and the role of these techniques within the software life cycle (K2)**

In this section, we consider how static testing techniques fit into the overall test process. Dynamic testing requires that we run the item or system under test, but static testing techniques allow us to find defects directly in work products, without the execution of the code and without the need to isolate the failure to locate the underlying defect. Static techniques include both reviews and static analysis, each of which we'll discuss in this chapter. Static techniques are efficient ways to find and remove defects, and can find certain defects that are hard to find with dynamic testing. As we go through this section, watch for the Syllabus terms **dynamic testing**,

**static analysis** and **static testing**. You will find these terms also defined in the Glossary.

In Chapter 1, we saw that testing is defined as all life cycle activities, both static and dynamic, to do with planning, preparing and evaluating software and related work products. As indicated in that definition, two approaches can be used to achieve these objectives, static testing and dynamic testing. Static analysis is a form of automated static testing.

**Static analysis** The process of evaluating a component or system without executing it, based on its form, structure, content or documentation.

**Static testing** Testing a work product without code being executed.

**Dynamic testing** Testing that involves the execution of the software of a component or system.

The definitions of **static analysis** and **static testing** are very similar, and to be honest, are somewhat confusing! The definition of static analysis would apply equally well to reviews, which are a form of static testing but are not part of static analysis. Generally, static analysis involves the use of tools to do the analysis; in fact, the Syllabus describes them as 'tool-driven evaluation' of code or work products. The key difference is: considering the code we want to evaluate, **dynamic testing** actually executes that code; static testing (including static analysis) does NOT execute the code we are evaluating.

One area where static analysis is often used (and is critical for) is in safety-critical systems such as flight control software, medical devices or nuclear power control software. However static analysis is also very important in security testing, as it can identify malicious code (which does not make itself visible in execution). In continuous delivery and continuous deployment, the automated build systems also frequently make use of static analysis as part of the build process.

### 3.1.1 Work products that can be examined by static testing

Static analysis is most often used to evaluate code against various criteria, such as adherence to coding standards, thresholds for complexity, spelling and grammar correctness and reading difficulty (the last few for work products other than code). However, static testing is broader than automated evaluations; reviews can be applied to any type of work product, including:

- Any type of specification: business requirements, functional requirements, security requirements.
- Epics, user stories and acceptance criteria.
- Code.
- Testware, that is, any type of work product to do with testing, for example test plans, test conditions, test cases, test procedures and automated test scripts.
- User guides, help text, wizards and other things designed to help the user to more effectively use the system.
- Web pages (there are also static analysis tools to analyze whether any links are broken for example).
- Contracts (a particularly important work product to review, as a lot of money may be riding on the specific wording), project plans, schedules and budgets.
- Models such as activity diagrams or other models used in model-based testing (MBT).

(Note that there is an ISTQB Foundation Level Model-Based Tester Extension Syllabus.)

Reviews apply to any work product; the reviewers need to be able to read and understand it, but can then provide feedback about it. See Section 3.2 for more on reviews.

Static analysis of software code is done using a tool to analyze the code with respect to the criteria of interest. For example, static analysis tools can identify dead code, a section of code that can never be reached from anywhere else in the code. This dead code can never be executed, so should be removed, as it could be confusing to leave it in. Static analysis can also identify a variable whose value is used before it has been defined. This type of defect can cause failures which are difficult to find by dynamic testing.

There are also tools that can analyze natural language text, for example in requirements, to catch some typos, assess readability level (ease of understandability), etc., so these are also a form of static analysis.

## 3.1.2 Benefits of static testing

Studies have shown that as a result of reviews, a significant increase in productivity and product quality can be achieved Gilb and Graham [1993] and van Veenendaal [1999]. Reducing the number of defects early in the product life cycle also means that less time would be spent on testing and maintenance. The use of static testing, such as reviews on software work products, has two major advantages:

- Since static testing can start early in the life cycle, early feedback on quality issues can be established, for example an early validation of user requirements rather than late in the life cycle during acceptance testing. Feedback during design review or backlog refinement is more useful than after a feature has been built.
- By detecting defects at an early stage, rework costs are most often relatively low, and thus relatively cheap improvements to the quality of software products can be achieved, as many of the follow-on costs of late updates are avoided, for example additional regression tests, confirmation tests, etc.

Additional benefits of static testing may include:

- Defects are more efficiently detected and corrected, particularly since this is done before dynamic test execution.
- Defects that are not easily found by dynamic testing, such as security vulnerabilities, are identified.
- Defects in future design and code are prevented by uncovering inconsistencies, ambiguities, contradictions, omissions, inaccuracies and redundancies in requirements.
- Since rework effort is substantially reduced, development productivity figures are likely to increase.
- Reduced development cost and time.
- Reduced testing cost and time. If defects are found and fixed before test execution starts, there are fewer to find in testing, so more tests pass, and there are fewer defect reports to write and fewer confirmation tests to run after fixes. This saves both time and money.
- Reduced total cost of quality over the software's lifetime. If defects are found and fixed early, then there should be fewer that get through to later testing or operation. The defects that are not there do not need to be investigated or fixed, saving time and money.

- Improved communication within the team, since there is an exchange of information between the participants during reviews, which can lead 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 work products themselves. It is also important that the quality improvement is not achieved just once, but has a more permanent nature. 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. This is particularly useful for sprint or project retrospectives.

### 3.1.3 Differences between static and dynamic testing

Static and dynamic testing have the same objectives: to assess the quality of work products and identify defects as early as possible. But static and dynamic testing are not the same. They find different types of defect, so they are complementary and are best used together. It does not make sense to ask if one is better than the other; both are useful and needed.

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. Dynamic testing can be started early by identifying test conditions and test cases as early as possible in the life cycle (as we discussed in Chapter 1 Section 1.4), but dynamic test execution 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 dynamic testing option is not applicable for the majority of the software work products. Among the questions that arise are:

- How can we evaluate or analyze a work product, such as a requirement specification, a user story, a design document, a test plan or a user manual?
- How can we effectively examine the source code before execution?

As discussed above, one powerful technique that can be used is static testing, for example reviews. In principle, all human-readable software work products can be tested using review techniques.

Types of defects that are easier to find during static testing include:

- Requirements defects, such as inconsistencies, ambiguities, contradictions, omissions, inaccuracies, redundancies.
- Design defects, such as inefficient algorithms or database structures, high coupling or low cohesion.
- Coding defects, such as variables with undefined values, variables that are declared but never used, unreachable code, duplicate code. All of these can be found by static analysis tools.
- Deviations from standards, for example lack of adherence to coding standards.
- Incorrect interface specifications, such as different units of measurement used by the calling system than by the called system. In 1999, a Mars Orbiter burned up in the atmosphere due to one team using metric units of measurement and the other using imperial units of measurement, as described in Nasa [1999].

- Security vulnerabilities, for example buffer overflow susceptibility.
- Traceability problems, such as gaps or inaccuracies or lack of coverage (for example missing tests for an acceptance criterion).
- Maintainability defects, such as improper modularization, poor reusability, code that is difficult to analyze and modify (often referred to as code smells). These defects do not show up in dynamic testing but can be critical for long-term costs of the system.

Compared to dynamic testing, static testing finds defects rather than failures. You may recall that in Chapter 1 Section 1.2.3 we made a distinction between errors, defects and failures. An error is a mistake made by a human being (for example in writing code), a defect is something that is wrong (for example in the code itself) and a failure is when the system or component does not perform as it should (for example returns the wrong balance). Static testing does not cause the system or component to do anything, so it cannot find failures; only dynamic testing can do that. However, static testing does find defects directly. Dynamic testing has to investigate the failure to find a defect.

In addition to finding defects, the objectives of reviews are often also informational, communicational and educational. 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 or product owners attend the review meeting and provide feedback to the development team, so reviews are also a means of customer/user communication.

## 3.2 REVIEW PROCESS

### SYLLABUS LEARNING OBJECTIVES FOR 3.2 REVIEW PROCESS (K3)

**FL-3.2.1** **Summarize the activities of the work product review process (K2)**

**FL-3.2.2** **Recognize the different roles and responsibilities in a formal review (K1)**

**FL-3.2.3** **Explain the differences between different review types: informal review, walkthrough, technical review and inspection (K2)**

**FL-3.2.4** **Apply a review technique to a work product to find defects (K3)**

**FL-3.2.5** **Explain the factors that contribute to a successful review (K2)**

In this section, we will focus on reviews as a distinct – and distinctly useful – form of static testing. We'll discuss the process for carrying out reviews. We'll talk about who does what in a review meeting and as part of the review process. We'll cover types of reviews that you can use. We'll look at different variations for reviews based on different ways to prepare for and perform reviews, and finally we will look at success factors to enable the most effective and efficient reviews possible. As we go through this section, watch for the Syllabus terms **ad hoc reviewing**, **checklist-based reviewing**, **formal review**, **informal review**, **inspection**, **perspective-based reading**, **review**, **role-based reviewing**, **scenario-based reviewing**, **technical review** and **walkthrough**. You will find these terms also defined in the Glossary.

One reason why reviews are so useful is that having a different person look at a work product is a way to overcome cognitive bias, the tendency to see what we intended rather than what we actually wrote.

**Review** A type of static testing during which a work product or process is evaluated by one or more individuals to detect issues and to provide improvements.

**Informal review**
A type of review without a formal (documented) procedure.

**Formal review**
A form of review that follows a defined process with a formally documented output.

**Reviews** vary from very informal to formal (that is, well-structured and regulated). Although inspection is perhaps the most documented and formal review technique, it is certainly not the only one. The formality of a review process is related by 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 work product. A two-person team can conduct an informal review, as the author can ask a colleague to review a work product or code. Pair working (pair programming, pair testing or a tester and developer pairing) is also an informal way to review the work products that both are working on. In later stages, reviews often involve more people and a meeting. This normally involves peers of the author, who try to find defects in the work product under review and discuss these defects in a review meeting. The goal is to help the author and to improve the quality of the work product. Informal reviews come in various shapes and forms, but all have one characteristic in common: they are not documented.

**Formal reviews** generally have team participation, documented results of the review and specified procedures to follow in carrying out the review.

Different reviews may have a different focus or objective for the review. For example, one objective may be to find defects. This is often at least one of the objectives of most types of review, formal or informal. Sometimes the objective is for all participants to gain knowledge and understanding; although a walkthrough is normally used for this purpose, an informal review could also meet this goal, and it is often a by-product (if not an explicit objective) for technical reviews or inspections. Another objective may be to hold discussions and come to a consensus about technical issues; this is normally the focus of a technical review.

The standard that covers review processes is ISO/IEC 20246 [2017].

## 3.2.1 Work product review process

In contrast to informal reviews, formal reviews follow a formal process. Informal reviews may perform at least some of the same activities to some extent. The review process consists of the following main activities:

### *Planning*

The Foundation Syllabus specifies the following elements of the planning activity:

- Defining the scope of the review: the purpose of the review, what work products (for example documents) or parts of work products to review and the quality characteristics to be evaluated in the review.

- Estimating effort and the timeframe for the review.
- Identifying review characteristics such as the type of review with roles, activities and checklists.
- Selecting the people to participate in the review and allocating roles to each reviewer.
- Defining the entry and exit criteria for more formal review types (for example inspections).
- Checking that entry criteria are met before the review starts (for more formal review types).

Let's examine these in more detail.

The review process for a particular review may begin with a request for review by the author to the review leader, who takes overall responsibility for the review, for example 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, for example inspections, the facilitator 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 work product that is not ready for review. A work product 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 likely to be less effective because the numerous obvious and minor defects will conceal the major defects.

The following are possible entry criteria for a formal review:

- A short check of a work product sample by the review leader (or expert) does not reveal many major defects.
- The work product to be reviewed is available with line numbers (if relevant).
- The work product has been cleaned up by running any applicable automated checks, such as static analysis or spelling and grammar assessments.
- References needed for the review are stable and available.
- The work product author is prepared to join the review team and feels confident with the quality of the product.

If the work product passes the entry check, the review leader and author decide which part(s) of it to review. Because the human mind can comprehend 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 work product type. It should be derived from practical 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 work product size has been set and the pages to be checked have been selected, the review leader determines, in co-operation with the author, the composition of the review team. The team normally consists of four to six participants, including facilitator (moderator) and author. To improve the effectiveness of the review, different roles may be assigned to each of the participants. These roles help the reviewers focus on particular types of defects during individual review.

This reduces the chance of different reviewers finding the same defects. The review leader or facilitator assigns the roles to the reviewers. (See below on role-based reviewing.)

Quality characteristics may also be evaluated and documented in a review, for example, the testability of a design or the readability or understandability of user help or installation instructions. These may also be assigned roles.

### *Initiate review*

The Foundation Syllabus specifies the following activities for initiating a review:

- Distributing the work products (physically or electronically) and any other relevant material such as logging forms, checklists or related work products.
- Explaining the scope, objectives, process, roles and work products to the participants.
- Answering any questions that participants may have about the review.

Let's examine these in more detail.

The goal of this set of activities is to get everybody on the same wavelength regarding the work product under review and to commit to the time that will be spent on checking (that is, individual reviewing). The result of the entry check and defined exit criteria are discussed in case of a more formal review. This stage of the review process is important to increase the motivation of reviewers and thus the effectiveness of the review process. At customer sites, we have measured results of up to 70% more major defects found per page as a result of performing a kick-off meeting, a form of review initiation, as described in van Veenendaal and van der Zwan [2001].

The review initiation may be done remotely, or it may involve a meeting (in person or using video conferencing). If a meeting is held, the reviewers receive a short introduction to the objectives of the review and the work products. The relationships between the work product under review and the other work products (for example sources or predecessor work products) are explained, especially if the number of related work products is high.

Role assignments, checking rate, the pages to be reviewed, the roles to be taken by each person, process changes and possible other questions are also discussed during this meeting.

Whether or not a meeting is held, the facilitator (moderator) ensures that each reviewer is clear about their responsibilities, and answers any questions that they may have, either about the work products, or the review process itself.

### *Individual review (that is, individual preparation)*

The Foundation Syllabus specifies the following activities for the individual review:

- Reviewing all or part of the work documents(s).
- Noting potential defects, recommendations and questions.

Let us examine these in more detail.

In the individual review, the participants work alone on the work product under review using the related work products, procedures, rules and checklists provided. The individual participants identify defects, recommendations, questions and comments, according to their understanding of the work product and the particular role they have been given. All issues are recorded, preferably using a logging form. Spelling mistakes are recorded on the work product under review, but not mentioned during the meeting. The annotated work product may be given to the author at the end of the

logging meeting. Using checklists during individual reviewing 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. This may also be an assigned role.

A critical success factor for a thorough preparation is the number of pages individually reviewed per hour. This is called the checking rate. The optimum checking rate is the result of a mix of factors, including the type of work product, its complexity, the number of related work products 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, for example one page per hour. During preparation, participants should not exceed the checking rate they have been asked to use. By collecting data and measuring the review process, company-specific criteria for checking rate and work product size (see Planning) can be set, preferably specific to a work product type.

### *Issue communication and analysis*

The Foundation Syllabus specifies the following activities for issue communication and analysis:

- Communicating identified potential defects, for example in a review meeting.
- Analyzing potential defects, assigning ownership and status to them.
- Evaluating and documenting quality characteristics.
- Evaluating the review findings against the exit criteria to make a review decision (reject; major changes needed; accept, possibly with minor changes).

Let's examine these in more detail.

In a formal review, the things found by the individual reviewers are communicated to the author of the work product (or the person who will fix defects), and may also be communicated to the other reviewers. We may refer to these as issues at this point because we do not yet know if they are defects or not. The issues may be communicated electronically, or in a review meeting, which may consist of the following activities (partly depending on the review type): logging, discussion and decision making.

#### *Logging in a review meeting*

During logging, the issues, that is, potential defects, that have been identified during the individual review 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 inspection. To ensure progress and efficiency, no real discussion is allowed during logging. If an issue needs discussion, the item is noted as a discussion item and then handled in the discussion part of the meeting. 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 discussed 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 may propose the severity, or the facilitator may assign a severity. If reviewers assign severity, it is important that the meaning of each category is understood by all reviewers in the same way. Otherwise, one reviewer may regard everything as critical, for example, skewing the review results. Severity classes could be:

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

- *Major:* defects could cause a downstream effect (for example a fault in a design can result in an error in the implementation).
- *Minor:* defects are not likely to cause downstream damage (for example non-compliance 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 work product under review and given to the author at the end of the meeting, or could be dealt with in a separate proofreading exercise.

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

### *Discussion part of a review meeting*

For a more formal review, the issues classified as discussion items will be handled during a discussion part of the review meeting, which occurs after the logging has been completed. Less formal reviews will often not have separate logging and discussion parts and will start immediately with logging mixed with discussion (which may lead to fewer defects being found). Participants can take part in the discussion by bringing forward their comments and reasoning. In the discussion part of the meeting, the facilitator or moderator takes care of people issues. For example, they prevent discussions from getting too personal, rephrase remarks if necessary and call 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 facilitator 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 noted as an action point if the issue cannot be solved during the meeting. The outcome of discussions is documented for future reference.

Quality characteristics may also be evaluated and documented at this point, for example, the testability of a design or the readability or understandability of user help or installation instructions.

### *Decision-making part of a review meeting*

At the end of the meeting, a decision on the work product under review has to be made by the participants, sometimes based on formal exit criteria. The most important exit criterion may be the average number of critical and/or major defects found per page (for example no more than three critical/major defects per page). If the number of defects found per page exceeds a certain level, the work product may need to be reviewed again, after it has been reworked. If the work product complies with the exit criteria, it will be checked later by the review leader, facilitator or one or more participants. Subsequently, the work product can leave the review process.

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.

If a project is under pressure, the review leader or facilitator will sometimes be forced to skip re-reviews and exit with a defect-prone work product. Setting (and agreeing to) quantified exit criteria helps the review leader or facilitator to make firm decisions at all times. Even if a limited sample of a work product has been (formally)

reviewed, an estimate of remaining defects per page can give an indication of likely problems later on.

For informal reviews, some of these activities may be performed, but informally. For example, potential defects may be emailed to the author of the work product with a suggested severity classification. The author may then evaluate exit criteria, possibly checking back with the reviewer(s).

### *Fixing and reporting*

The Foundation Syllabus specifies the following activities for fixing and reporting:

- Creating defect reports for those findings that require changes.
- Fixing defects found (typically done by the author) in the work product reviewed.
- Communicating defects to the appropriate person or team (when found in a work product related to the work product reviewed).
- Recording updated status of defects (in formal reviews), potentially including the agreement of the comment originator.
- Gathering metrics (for more formal review types), for example of defects fixed, deferred, etc.
- Checking that exit criteria are met (for more formal review types).
- Accepting the work product when the exit criteria are reached.

Let's examine these in more detail.

Defect reports may have been recorded in a general defect logging tool (for example also used by testers) or may have been recorded in a review log.

During fixing, the author will improve the work product under review step by step, based on the issues or defects detected by the individual reviewers and/or those found in the review meeting. Not every issue that is reported is a defect that leads to a fix. It is often the author's responsibility to judge if an issue really is a defect which has to be fixed, though in some cases the review meeting participants may make those decisions. If nothing is done about an issue for a certain reason, it should still be reported to show that the author has considered it.

Changes that are made to the work product should be easy to identify by anyone who will be confirming that the fixes are correct. For example, the author may turn on 'track changes' in a document.

Sometimes an issue raised affects a work product that is not under the direct control of the author of the reviewed work product. For example, reviewing a feature to be implemented may reveal an inconsistency or omission in the user story or requirements specification that it is based on. If the author cannot update the user story or requirement specification directly, they need to communicate this to whoever can update the related work product.

As defects are fixed, the status of those defects should be updated wherever they are managed. In some reviews, the originator of the comment or defect would then confirm that the defect has been fixed adequately, that the author has correctly understood their comment and fixed the right problem in the right way.

In order to control and optimize the review process, a number of measurements are collected by the facilitator 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 responsibility of the facilitator or moderator to ensure that the information is correct and stored for future analysis.

The facilitator or moderator is responsible for ensuring that satisfactory actions have been taken on all logged defects, process improvement suggestions and change requests. Although the facilitator or moderator checks to make sure that the author has taken action on all known defects, it is not necessary for them to check all the corrections in detail. If it is decided that all participants will check the updated work product, the facilitator or moderator takes care of the distribution and collects the feedback. For more formal review types the review leader or facilitator checks for compliance to the exit criteria.

When all of the exit criteria have been met, including fixing, checking of fixes and confirming that the review process was properly carried out, then the work product can be formally accepted. This may be particularly important in safety-critical systems.

## 3.2.2 Roles and responsibilities in a formal review

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 reviewing. 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 successful without trained participants. This indeed is a critical success factor.

The best formal reviews come from well-organized teams, guided by trained facilitators (moderators). Within a review team, six types of roles can be distinguished: author, management, facilitator (or moderator), review leader, reviewers and scribe (or recorder).

### *The author*

The author has two main responsibilities:

- Creating the work product under review.
- Fixing defects in the work product (if necessary).

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

### *Management*

Management has a number of very important responsibilities in successful reviews, including:

- Ensuring that reviews are planned.
- Deciding on the execution of reviews.
- Assigning staff, budget and time.
- Monitoring ongoing cost effectiveness.
- Executing control decisions in the event of inadequate outcomes.

The role of management in reviews is often underestimated, but without adequate support from managers, reviews are seldom successful. The manager needs to believe in reviews, and to ensure that they will be carried out as part of development. This