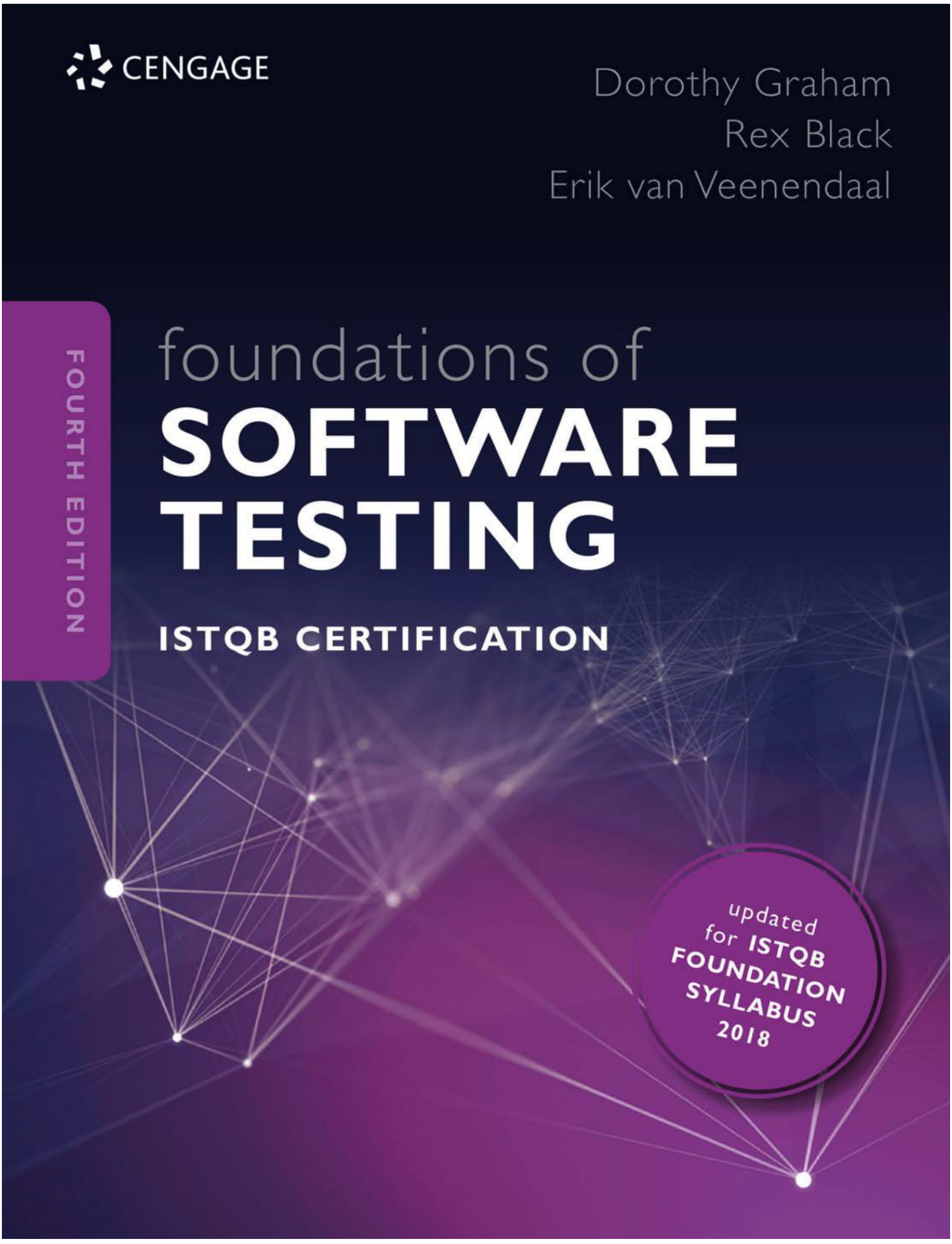
CENGAGE
Dorothy Graham
Rex Black
Erik van Veenendaal
FOURTH EDITION
foundations of
SOFTWARE
TESTING
ISTQB CERTIFICATION
updated
for ISTQB
FOUNDATION
SYLLABUS
2018

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'.

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'.

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?'

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

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

1.1.1 Typical objectives of testing

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

- 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.

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

- 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).

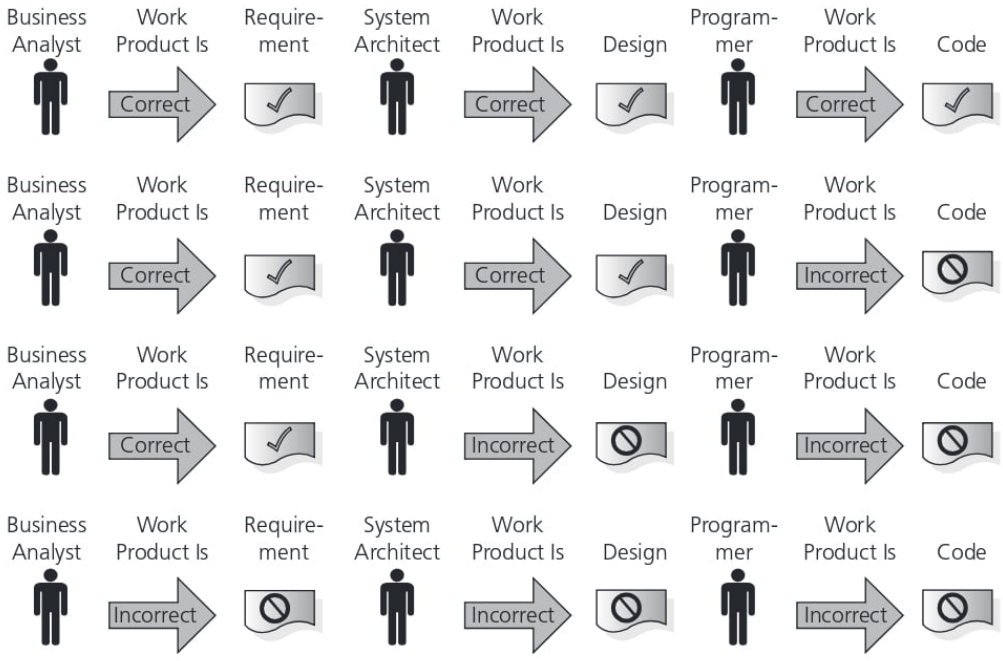

FIGURE 1.1 Four typical scenarios

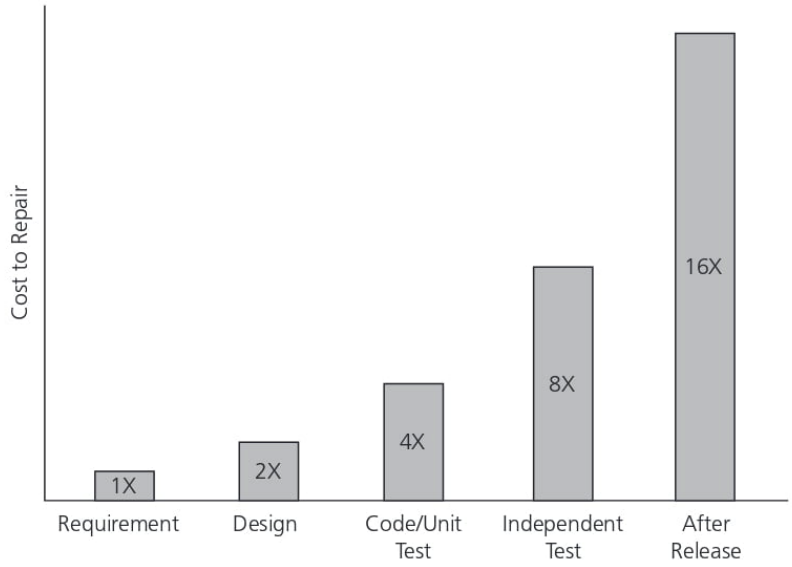

FIGURE 1.2 Multiplicative increases in cost

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.

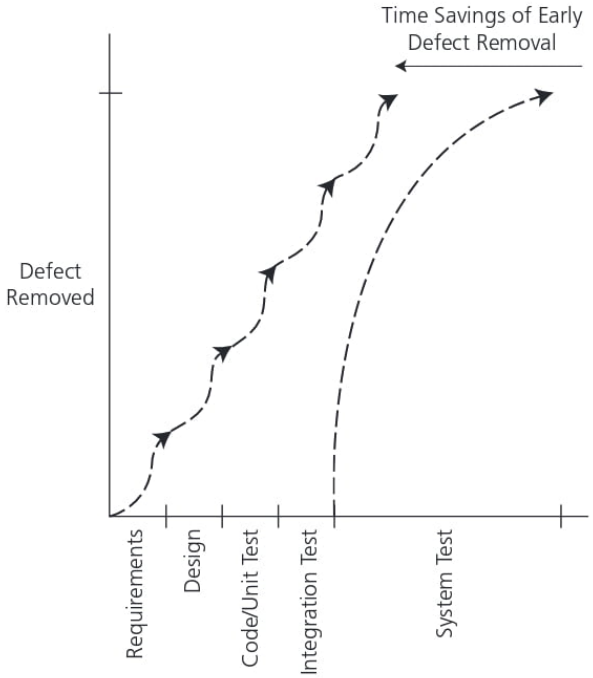

FIGURE 1.3 Time savings of early defect removal

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).