Rex Black
Erik Van Veenendaal
Dorothy Graham

THIRD EDITION

Updated for ISTQB Foundation Syllabus 2011 and Glossary 2.1

FOUNDATIONS OF SOFTWARE TESTING

ISTQB CERTIFICATION

Foundations of Software Testing: ISTQB Certification, 3rd Edition
Rex Black, Erik van Veenendaal, and Dorothy Graham

Publishing Director: Linden Harris
Publisher: Brendan George
Development Editor: Annabel Ainscow
Content Project Editor: Kate Daniel
Production Controller: Eyvett Davis
Typesetter: Integra Software Services
Cover Design: Design Deluxe, Bath

This work is adapted from *Foundations of Software Testing* by Dorothy Graham, Erik van Veenendaal, Isoleb Evans and Rex Black published by South-Western, a division of Cengage Learning, Inc. © 2009.

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

ISBN: 978-1-4080-4405-6

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

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

For your lifelong learning solutions, visit **www.cengage.co.uk**

Purchase your next print book, e-book or e-chapter at
www.cengagebrain.com

Printed in China by RR Donnelley
1 2 3 4 5 6 7 8 9 10 – 14 13 12

CHAPTER ONE

Fundamentals of testing

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

1.1 WHY IS TESTING NECESSARY?

SYLLABUS LEARNING OBJECTIVES FOR 1.1 WHY IS TESTING NECESSARY? (K2)

LO-1.1.1 **Describe, with examples, the way in which a defect in software can cause harm to a person, to the environment or to a company. (K2)**

LO-1.1.2 **Distinguish between the root cause of a defect and its effects. (K2)**

LO-1.1.3 **Give reasons why testing is necessary by giving examples. (K2)**

LO-1.1.4 **Describe why testing is part of quality assurance and give examples of how testing contributes to higher quality. (K2)**

LO-1.1.5 **Explain and compare the terms error, defect, fault, failure and the corresponding terms mistake and bug, using examples. (K2)**

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

As we go through this section, watch for the Syllabus terms **bug, defect, error, fails (false-fail result, false-positive result), failure, fault, mistake, passed (false-negative result, false-pass result), quality,** and **risk**. You'll find these terms defined in the glossary.

1.1.1 Software systems context

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 young 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 were 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. Billions of mobile phones exist, many of which are hand-held computers that get smarter and smarter with every new model.

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.

False-fail result: A test result in which a defect is reported although no such defect actually exists in the test object.

False-positive result: See false-fail result.

1.1.2 The human (and other) causes of software defects

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.

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

Defect (bug, fault) A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g. an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system.

Failure Deviation of the component or system from its expected delivery, service or result.

The precise chain of events goes something like this. A programmer makes a **mistake** (or **error**), such as forgetting about the possibility of inputting an excessively long string into a field on a screen. The programmer thus puts a **defect** (or **fault** or **bug**) into the program, such as forgetting to check 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 will result in unexpected behaviour; i.e. the system exhibits a **failure**, such as accepting an over-long input that it should reject.

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 1000 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's important to point out that the introduction of defects is not a purely random accident, though some defects are 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 mistakes 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 lifecycle.

Ideally, defects are removed in the same phase of the lifecycle in which they are introduced. (Well, ideally defects aren't introduced at all, but this is not possible, for, 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 lifecycle phase. Multiplicative increases in cost, of the sort seen in

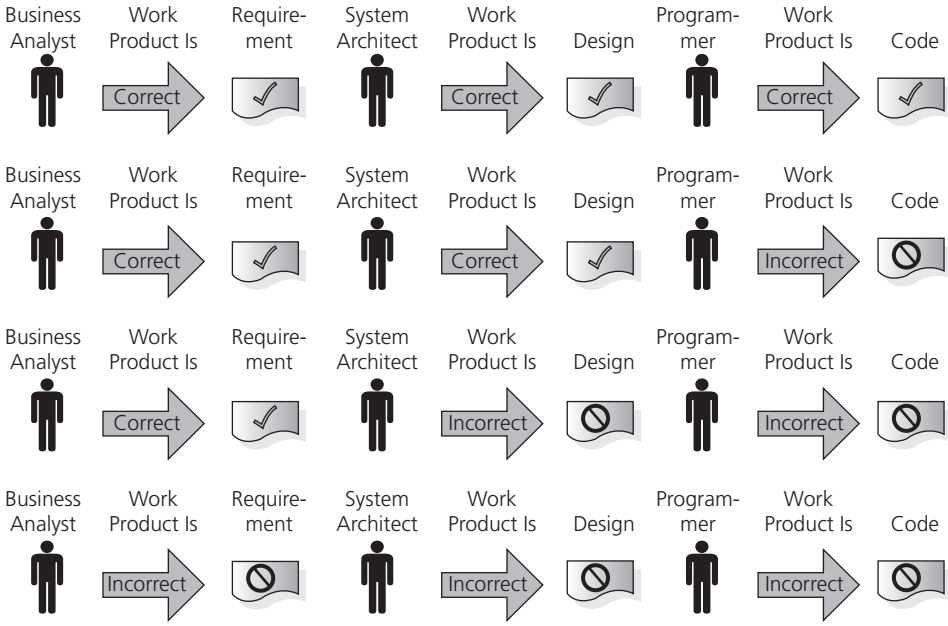

FIGURE 1.1 Four typical scenarios

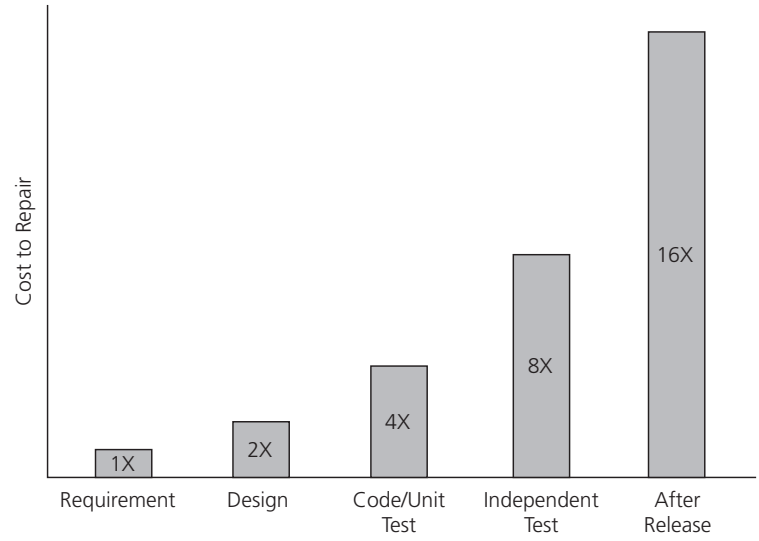

FIGURE 1.2 Multiplicative increases in cost

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. If you are curious about the economics of software testing and other **quality**-related activities, you can see [Gilb and Graham 1993], [Black 2009] or [Black 2004].

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

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).

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.

1.1.3 The role of testing, and its effect on quality

Risk A factor that could result in future negative consequences; usually expressed as impact and likelihood.

Since software and the associated work products are written by fallible humans, all of these work products will always have defects. Where there are defects, there are **risks** of failure. While nothing can reduce the level of risk to zero, we certainly can – and should – try to reduce risk to an acceptable level prior to releasing the software to customers and users.

Testing is part of how these risks of failure can be reduced. If the software and its associated work products (including requirements specifications, design specifications, documentation, etc.) are subjected to the kind of rigorous, systematic testing described in this book and in the Foundation syllabus, the testers are more likely to find defects in some areas, while reducing the risk in other areas by determining that

the software works properly under the tested conditions. Developers and others on the project teams can then debug the system, removing the defects found.

Testing does not change the quality of the system under test directly. When testing finds defects and those defects are repaired, the quality of the system is increased. However, debugging, not testing, is the activity that changed the quality of the system (see Section 1.2 below).

Testing does provide a way of measuring the system's quality. The rate of defect discovery, the number of known defects, the extent of test coverage, and the percentage of tests which have **passed** all reflect on the quality of the system. If the tests find few or no defects (assuming we followed proper testing approaches such as those described in this book), or if all of the defects found are resolved, then we can have confidence in the system. When a properly designed test is run, an unknown situation is changed into a known situation, which reduces the level of risk to the quality of the product.

False-pass result: A test result which fails to identify the presence of a defect that is actually present in the test object.

False-negative result: See false-pass result.

At this juncture, let us point out that a properly designed set of tests should measure the quality of the system in terms of both functional and non-functional characteristics. The specific characteristics tested depend on what matters in terms of quality for the particular system under test. We'll discuss functional and non-functional test types further in Chapter 2. You can find more information on functional and non-functional quality characteristics in 'Software Engineering – Software Product Quality' [ISO 9126].

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.

Testing thus 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 software 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 assure acceptable levels of quality and quality risk upon release.

1.1.4 How much testing is enough?

Selecting which test conditions to cover is a fundamental problem of testing. As will be discussed later in this chapter and often in this book, the number of possible test cases is infinite for all but the simplest systems. Given immortality and unlimited resources, we *could* conceivably test forever. However, in the real world, we can only select a finite number of test cases for design, implementation, and execution.

Instead of trying to find every possible test case, an impossible ideal, we should focus on what tests give us the greatest value by covering the most important parts of the system. Coverage can be measured in a number of ways. In this book, we will talk about coverage of requirements, coverage of code structures, and coverage of risks to the quality of the product (including risks derived from technical, safety, and business considerations).

We could say that sufficient coverage is achieved when we balance what we should cover against project constraints such as time and budget. We should also ensure that testing provides sufficient information to the project and product stakeholders, so that

they can decide whether to proceed with the project or with release. All three of these areas – our analysis of what should be covered, the limitations on what can be covered, and the stakeholders' specific needs for information – are to some extent in tension, in that we may have to give up something in one area in order to increase what we achieve in one or both of the other areas.

Testing always involves trade-offs. At some point, the various risks associated with the project will have reached some acceptable level, one understood by the various project stakeholders. We will discuss this issue of risk and how it influences testing in Chapter 5.

In some cases, software testing is not just a good way to reduce risks to the quality of the system to an acceptable level; testing can be required to meet contractual or legal requirements, or industry-specific standards. The authors have clients in the medical systems business, and those clients are subject to regulations that require a certain degree and rigor of testing. For companies writing software used on commercial aviation, the US Federal Aviation Administration requires a certain level of structural test coverage, a topic which we'll discuss in Chapter 4.

1.2 WHAT IS TESTING?

SYLLABUS LEARNING OBJECTIVES FOR 1.2 WHAT IS TESTING? (K2)

LO-1.2.1 Recall the common objectives of testing. (K1)

LO-1.2.2 Provide examples for the objectives of testing in different phases of the software life cycle. (K2)

LO-1.2.3 Differentiate testing from debugging. (K2)

In this section, we will review the common objectives of testing and the activities that allow us to achieve those objectives. We'll explain how testing helps us to find defects, provide confidence and information, and prevent defects.

As you read this section, you'll encounter the terms **confirmation testing, debugging, requirement, re-testing, review, test case, test control, test design specification, testing,** and **test objective.**

An ongoing misperception 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 other activities involved in the test process.

While we'll discuss the test process in more detail later in this chapter (in section 1.4), let's look briefly at some of the major activities in the test process:

- **Test planning**: In test planning, we establish (and update) the scope, approach, resources, schedule, and specific tasks in the intended test activities that comprise the rest of the test process. Test planning should identify test items, the features to be tested and not tested, the roles and responsibilities of the participants and stakeholders, the relationship between the testers and the developers of the test items, the extent to which testing is independent of development of these work

items (see section 1.5 below), the test environments required, the appropriate **test design** techniques, entry, and exit criteria, and how we'll handle project risks related to testing. Test planning often produces, as a deliverable work product, a test plan. Test planning is discussed in Chapter 5.

- **Test control**: While test planning is essential, things don't always go according to plan. In **test control**, we develop and carry out corrective actions to get a test project back on track when we deviate from the plan. Test control is discussed in Chapter 5.
- **Test analysis**: In test analysis, we identify what to test, choosing the test conditions we need to cover. These conditions are any item or event that we can and should verify using one or more **test cases**. Test conditions can be functions, transactions, features, quality attributes, quality risks, or structural elements. Test analysis is discussed in Chapter 4.
- **Test design**: In test design, we determine how we will test what we decided to test during test analysis. Using test design techniques, we transform these general test conditions and the general **testing objectives** from the test plan into tangible test cases at the appropriate level of detail. Test design generally, and specific test design techniques, are discussed in Chapter 4.
- **Test implementation**: In test implementation, we carry out the remaining activities required to be ready for test execution, such as developing and prioritizing our test procedures, creating test data, and setting up test environments. These elements of test implementation are covered in Chapter 4. In many cases, we want to use automated test execution as part of our test process. In such cases, test implementation includes preparing test harnesses and writing automated test scripts. These elements of test implementation are covered in Chapter 6.
- **Test execution**: In test execution, we run our tests against the test object (also called the system under test).
- **Checking results**: As part test execution, we see the actual results of the test case, the consequences and outcomes. These include outputs to screens, changes to data, reports, and communication messages sent out. We must compare these actual results against expected results to determine the pass/fail status of the test. Defining expected results is discussed in Chapter 4, while managing test execution, including checking of results, is discussed in Chapter 5.
- **Evaluating exit criteria**: At a high level, exit criteria are a set of conditions that would allow some part of a process to complete. Exit criteria are usually defined during test planning by working with the project and product stakeholders to balance quality needs against other priorities and various project constraints. Such criteria ensure that everything that should be done has been done before we declare some set of activities finished. Specifically, test exit criteria help us report our results (as we can report progress against the criteria) as well as helping us plan when to stop testing. Establishing and evaluating exit criteria are discussed in Chapter 5.
- **Test results reporting**: In test results reporting, we want to report our progress against exit criteria, as described above. This often involves details related to the status of the test project, the test process, and quality of the system under test. We'll discuss test results reporting in Chapter 5.
- **Test closure**: Test closure involves collecting test process data related to the various completed test activities in order to consolidate our experience, re-useable testware, important facts, and relevant metrics. Test closure is discussed in section 1.4 below.

Test design specification A document specifying the test conditions (coverage items) for a test item, the detailed test approach and identifying the associated high level test cases.

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 case A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.

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

Testing The process consisting of all lifecycle 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.

Requirement A condition or capability needed by a user to solve a problem or achieve an objective that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.

Review An evaluation of a product or project status to ascertain discrepancies from planned results and to recommend improvements. Examples include management review, informal review, technical review, inspection, and walkthrough.

Notice that there are major test activities both before and after test execution. In addition, in the ISTQB definition of software **testing**, you'll 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 (e.g. requirements, **reviews** and static analysis of code) that non-testers (and even sometime 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.

The following are some objectives for testing given in the Foundation Syllabus:

- Finding defects, such as the identification of failures during test execution that lead to the discovery of the underlying defects.
- Gaining confidence in the level of quality, such as when those tests considered highest risk pass and when the failures that are observed in the other tests are considered acceptable.
- Providing information for decision-making, such as satisfaction of entry or exit criteria.
- Preventing defects, such as when early test activities such as 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 of these test basis documents that will reveal problems that otherwise would not surface until test execution, potentially much later in the project.

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) along with building confidence that the system meets its requirements.

When done to evaluate 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.

Testing can 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 can be an objective of a separate level of testing, as when testing a safety critical system.

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

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** or **re-testing**. This activity ensures that the fix does indeed resolve the failure. In terms of roles, dynamic and static testing are testing roles, debugging is a development role, and confirmation testing is again a testing role.

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

Confirmation testing (re-testing) Testing that runs test cases that failed the last time they were run, in order to verify the success of corrective actions. [Note: While the Glossary uses re-testing as the preferred term, this preference is out of step with ordinary usage and with the actual English-language definition of 're-testing'.]

1.3 SEVEN TESTING PRINCIPLES

SYLLABUS LEARNING OBJECTIVES FOR 1.3 SEVEN TESTING PRINCIPLES (K2)

LO-1.3.1 Explain the seven principles in testing. (K2)

In this section, we will review seven fundamental principles of testing which 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.

As you read this section, you'll encounter the terms **complete testing**, **exhaustive testing** and **test strategy**.

The following subsections will review each principle, using the principle name as the subsection title for easy correlation to the syllabus. In addition, you can refer to Table 1.1 for a quick reference of the principles and their text, as written in the syllabus.

1.3.1 Testing shows the presence of defects

As mentioned in the previous section, one 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 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.

However, no test team, test technique, or **test strategy** can achieve 100% defect detection effectiveness. 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 with such a mathematical series.

Test strategy A high-level description of the test levels to be performed and the testing within those levels for an organization or program (one or more projects).

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

Exhaustive testing (complete testing) A test approach in which the test suite comprises all combinations of input values and preconditions.

1.3.2 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 a couple of examples. 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, as mentioned in section 1.1, 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 to identify test conditions. These test conditions inherit the priority of the requirement they derive from. We focus the test effort based on the priority to determine the appropriate number of test cases for each requirement, and also to sequence the test cases.

1.3.3 Early testing

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 should minimize the cost of removing defects.

So, the first principle tells us that we can't find all the bugs, but rather can only find some percentage of them. The second principle tells us that we can't 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 1000 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 lifecycle. 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, less than 30 defects remain. Even though each test activity was only 45% effective at finding defects, the overall sequence of activities was 97% effective.

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 won't achieve the cumulative effect, for reasons we'll discuss in a moment.

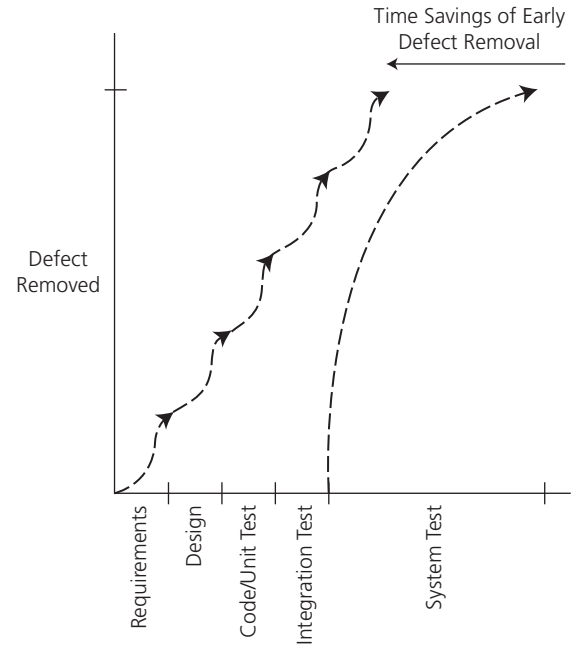

FIGURE 1.3 Time savings of early defect removal

1.3.4 Defect clustering

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 less) 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]. 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.

1.3.5 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's used, so too a given set of tests will eventually stop finding new defects when it's re-run against a system under development or maintenance. If the tests don't provide adequate coverage, this slowdown in defect finding will result in a level of false confidence and

excessive optimism among the project team. The air will be let out of the balloon once the system is released to customers and users, though.

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 test coverage set is adequate, and that the remaining level of risk is acceptable.

1.3.6 Testing is context dependent

Our safety-critical clients test with a great deal of rigor 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, 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 an e-commerce site, 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.

1.3.7 Absence-of-errors fallacy

Throughout this section, we've expounded the idea that a sequence of test activities, started early, 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's all which is required to achieve project success?

Sadly, it is not. Many systems have been built which failed in user acceptance testing or in the marketplace. Some of these systems failed due to a high level of defects, such as the Apple Newton. However, some systems had very low levels of defects, yet still failed.

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, 20 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.

Perhaps you are wondering where this leaves Apple. If usability is important, why is the Mac's share of the desktop marketplace so low, given the MacOS reputation for ease-of-use? The problem lies in Apple's failure to optimize MacOS for enterprise and large-company use. Certain features treasured by system administrators are not present in MacOS.

Macs also suffer from a major problem, which, while not related to software quality, creates an insurmountable obstacle for many CIOs and CFOs: unit price. With organizations forced to refresh their desktop systems anywhere from two to three times a decade, and with many large enterprises owning thousands upon thousands of desktop systems, spending even $100 more than necessary on commoditized computer hardware is simply a non-starter, no matter how well the system works.

1.4 FUNDAMENTAL TEST PROCESS

SYLLABUS LEARNING OBJECTIVES FOR 1.4 FUNDAMENTAL TEST PROCESS (K1)

LO-1.4.1 Recall the five fundamental test activities and respective tasks from planning to closure (K1)

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

In this section, you'll also encounter the glossary terms **coverage (test coverage), exit criteria, incident, regression testing, test approach, test basis, test condition, test data, test execution, test log, test monitoring, test plan, test procedure, test suite, test summary report,** and **testware.**

In section 1.2, we had an overview of testing activities throughout the lifecycle, going beyond the usual understanding of testing as consisting simply of **test execution**. Certainly, test execution is the most visible testing activity. However, as we saw in section 1.3, effective and efficient testing requires properly planned and executed **test approaches**, with tests designed and implemented to cover the proper areas of the system, executed in the right sequence, and with their results reviewed regularly.

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

Test approach The implementation of the test strategy for a specific project. It typically includes the decisions made that follow based on the (test) project's goal and the risk assessment carried out, starting points regarding the test process, the test design techniques to be applied, exit criteria and test types to be performed.

To help testers maximize the likelihood of such effective and efficient software testing, the ISTQB has defined and published a fundamental test process consisting of the following main activities:

- Planning and control
- Analysis and design
- Implementation and execution
- Evaluating exit criteria and reporting
- Test closure activities

These activities are logically sequential, in the sense that tasks within each activity often create the pre-conditions or precursor work products for tasks in subsequent activities. However, in many cases, the activities in the process may overlap or take place concurrently, provided that these dependencies are fulfilled.

You should not mistake the name given above, fundamental test process, to mean 'the one and only way to test which never alters from one project to another and always contains these and only these test tasks'. 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

the fundamental test process, its main activities, and the constituent tasks, based on the organizational, project, process, and product needs, constraints, and other contextual realities.

1.4.1 Test planning and control

As mentioned in section 1.2, planning involves defining the overall strategic and tactical objectives of testing, as well as discovering and specifying the specific test activities required to satisfy those objectives and the general mission of testing. Metaphorically, you can think of **test planning** as similar to figuring out how to get from one place to another. 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.

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 control we continuously compare actual progress against the plan, adjust the plan, report the test status and any necessary deviations from the plan, monitor test activities, and take whatever actions are necessary to meet the mission and objectives of the project. To some extent, test control involves re-planning, which should take into account the feedback from these **test monitoring** and control activities.

We'll discuss test planning and control tasks in more detail in Chapter 5.

Test plan A document describing the scope, approach, resources and schedule of intended test activities. It identifies among others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice, and any risks requiring contingency planning. It is a record of the test planning process.

Test monitoring A test management task that deals with the activities related to periodically checking the status of a test project. Reports are prepared that compare the actuals to that which was planned.

1.4.2 Test analysis and design

In test analysis and design, we transform the more general testing objectives defined in the test plan into tangible **test conditions** and test cases. The way in which and degree to which the test conditions and test cases are made tangible – that is to say, specifically documented – depends on the needs of the testers, the expectations of the project team, any applicable regulations, and other considerations. Test analysis and design is discussed in more detail in Chapter 4.

During test analysis and design activities, we may have to perform the following major tasks:

- Review the **test basis**. We can say colloquially that the test bases are those documents upon which we base our tests. The test basis can include requirements and design specifications, risk analysis reports, the system design and architecture, and interface specifications. The Foundation syllabus also defines the software integrity level as a possible test basis. The software integrity level is the degree to which software must comply with a set of stakeholder-selected characteristics, such as software complexity, the risk assessment, the required safety and security levels, desired performance and reliability, or cost, which reflect the importance of the software to its stakeholders.
- Evaluate the testability of the test basis and test objects, which can result in reporting on any issues that might impede testing and adjusting the test plan to deal with these issues. This task often occurs as part of or in parallel with the

Test condition An item or event of a component or system that could be verified by one or more test cases, e.g. a function, transaction, feature, quality attribute, or structural element.

Test basis All documents from which the requirements of a component or system can be inferred. The documentation on which the test cases are based. If a document can be amended only by way of formal amendment procedure, then the test basis is called a frozen test basis.

review of the test basis, but it's important to see these as distinct so that this important task is not forgotten.

- Identify and prioritize specific test conditions based on analysis of the risks, test items, the requirements and design specifications, and the behaviour and structure of the software.
- Design and prioritize high level (i.e. abstract or logical) test cases. Such test cases give general guidance on inputs and expected results, but do not include the specific inputs or expected results. For example, a high level test case might specify testing the checkout function of an e-commerce application, but would not include the particular inputs for specific fields or the exact appearance of screens after the inputs are submitted.

Test data Data that exists (for example, in a database) before a test is executed, and that affects or is affected by the component or system under test.

- Identify the necessary **test data** to support the test conditions and test cases as they are identified and designed.
- Design the test environment, including the set-up and any required infrastructure and tools.
- Create traceability between the test basis documents and the test cases. This traceability should be bi-directional so that we can check which test basis elements go with which test cases (and vice versa) and determine the degree of **coverage** of the test basis by the test cases. Traceability is also very important for maintenance testing, as we'll discuss in Chapter 2.

Coverage (test coverage) The degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite.

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

1.4.3 Test implementation and execution

In test implementation and execution, 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. This involves transforming the high level (or abstract or logical) test cases into low level (concrete) test cases, which may be done more formally and written into a test script, or it may be done 'on the fly' as a tester is executing tests from a list of high level test conditions. Test implementation also involves setting up the test environment. During test execution, of course, we run the tests.

Test procedure specification (test procedure, test script, manual test script) A document specifying a sequence of actions for the execution of a test.

During test implementation, we may have to perform the following major tasks:

- Finalize, implement and prioritize the test cases.
- Identify and create specific test data, which can take the form of inputs, data resident in databases and other data repositories, and system configuration data.
- Develop and prioritize test procedures, often using the test basis and test project constraints to achieve the optimal order of test execution. Once we have prioritized test procedures, we may need to create **test suites** from the test procedures for efficient test execution.

Test suite A set of several test cases for a component or system under test, where the post condition of one test is often used as the precondition for the next one.

- If automation is to occur, prepare test harnesses and write automated test scripts. Test automation is discussed in Chapter 6.
- Verify that the test environment has been set up correctly, ideally before test execution starts so that test environment issues do not impede test progress.
- Verify and update the bi-directional traceability prepared previously, to make certain that adequate coverage exists.

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 be comprised almost entirely of test implementation activities that could have been completed before the software was delivered.

During test execution, we may have to perform the following major tasks:

- Execute the test procedures manually or with the use of automated test execution tools, according to the planned sequence.
- Compare actual results with expected results, observing where the actual and expected results differ. These anomalies should be logged as **incidents** for further investigation.
- Log the outcome of test execution. 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**.
- Analyze the incidents in order to establish their cause. Causes of incidents can include defects in the code, in which case we have a failure. Other incidents result from defects in specified test data, in the test document, or simply a mistake in the way the test was executed. Report the incidents as appropriate. Some organizations track test defects (i.e. defects in the tests themselves) as incidents, while others do not. Incident analysis and reporting is discussed in Chapter 5.
- 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 a corrected 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**).

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

1.4.4 Evaluating exit criteria and reporting

In evaluating **exit criteria** and reporting, we assess test execution against the objectives which we defined in the test plan. We should do so for each test level (as discussed in Chapter 2, Section 2).

During evaluation of exit criteria and reporting, we may have to perform the following major tasks:

- Check the **test logs** gathered during test execution against the exit criteria specified in test planning.
- Assess if more tests are needed or if the exit criteria specified should be changed.
- Write a **test summary report** for stakeholders, as discussed in Chapter 5.

The specific evaluation and reporting tasks and deliverables that apply to a particular project depends on various contextual issues relevant to the project, which are discussed further in Chapter 5.

Incident Any event occurring that requires investigation.

Testware Artifacts produced during the test process required to plan, design, and execute tests, such as documentation, scripts, inputs, expected results, set-up and clear-up procedures, files, databases, environment, and any additional software or utilities used in testing.

Regression testing Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made. It is performed when the software or its environment is changed.

Exit criteria The set of generic and specific conditions, agreed upon with the stakeholders, for permitting a process to be officially completed. The purpose of exit criteria is to prevent a task from being considered completed when there are still outstanding parts of the task which have not been finished. Exit criteria are used to report against and to plan when to stop testing.

Test log A chronological record of relevant details about the execution of tests.

Test summary report A document summarizing testing activities and results. It also contains an evaluation of the corresponding test items against exit criteria.

1.4.5 Test closure activities (K1)

In test closure, we collect data from completed test activities to consolidate experience, testware, facts and numbers. Test closure activities should occur at major project milestones. These can include when a software system is released, a test project is completed (or cancelled), a milestone has been achieved, or a maintenance release has been completed, though the specific milestones that involve closure activities should be specified in the test plan.

During test closure, we may have to perform the following major tasks:

- Check which planned deliverables have been delivered, and possibly create any deliverables we might have missed.
- Ensure that all incident reports are resolved, and possibly log change requests for any that remain open.
- Document the acceptance of the system, particularly as part of the closure of the user acceptance test level.
- Finalize and archive testware, the test environment and the test infrastructure for later reuse.
- Hand over the testware to the maintenance organization, if that is a different group.
- Analyze lessons learned to determine changes needed for future releases and projects (i.e. perform a retrospective).
- Use the information gathered to improve test maturity, especially as an input to test planning for future projects.

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

1.5 THE PSYCHOLOGY OF TESTING

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

LO-1.5.1 Recall the psychological factors that influence the success of testing (K1)

LO-1.5.2 Contrast the mindset of a tester and of a developer (K2)

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

You'll find a few syllabus terms in this section, **error guessing**, **independence of testing** and **test policy**.

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).

While these 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 case – of finding defects. Indeed many times finding defects is a primary objective of static and dynamic testing activities.

That mindset is quite different from the mindset that a business analyst, system design, architect, database administrator, or programmer 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.

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 what the Foundation syllabus refers to as the author bias. 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. As we saw earlier, because human beings are fallible, all software work products contain defects when initially created, and any one of those defects could render the work product unfit for use.

While some developers are aware of their author bias when they participate in reviews and perform unit testing of their own work products, that author bias acts 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 proof-reading our own work, for example, we see what we meant, not what we put!

In addition, many business analysts, system designers, architects, database administrators, and programmers 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.

A trained independent tester – or better yet a trained, certified, independent test team – can overcome both of these limitations. Independent test teams tend to be more effective at finding defects and failures. Capers Jones reports that unit testing by developers tops out at 40 to 50% defect detection effectiveness, while we regularly find independent test teams scoring 85% defect detection effectiveness. Indeed, some of our best clients, those with trained, certified, independent test teams, regularly score defect detection effectiveness of 95% and above.

To be most effective at finding defects, a tester needs the right mindset. Looking for defects and failures in a system calls for people with the following traits:

- 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 her personal slogan, 'If in doubt, it's 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.

Error guessing A test design technique where the experience of the tester is used to anticipate what defects might be present in the component or system under test as a result of errors made, and to design tests specifically to expose them.

A good independent tester has the skills, the training, the certification, and the mindset of a professional tester, and of these four the most important – and perhaps the most elusive – is the mindset. The best testers continuously strive to attain a more professional mindset, and it is a lifelong journey.

All of this is not to say that software work product developers – business analysts, system designers, architects, database administrators, and programmers – should not review and test their own work. They certainly should. Quality is everyone's responsibility. The presence of an independent test team or single tester is not a signal to developers to abdicate their professional responsibilities to produce the best quality work products they can. Abdicating responsibility for quality and saying, 'Hurrah, we have an independent test team now, so we don't have to test our own work, we can just let the testers find all the bugs', is a known worst-practice of software development.

When highly-defective, barely functioning software is delivered to independent test teams in higher levels of testing such as system test or system integration test, such a level of bugginess often overwhelms the testing process and results in defect detection effectiveness much lower than 85%. Since more defects are delivered in

the first place, this situation results in a much higher number of defects delivered to users and customers.

So, developer self-testing has an important role to play. Business analysts have a deep understanding of the problem the software is to solve. Designers, system architects, and database administrators have a deep understanding of how the system fits together, and fits with other important systems such as databases. Programmers have a deep understanding of the implementation and low-level design of the software. For this reason, business analysts, system designers, architects, database administrators, and programmers should participate in reviews of their work products and carry out static analysis of their work products. In addition, unit and component integration testing is usually carried out by programmers, though independent testers can play a role in these test levels.

It's also important to understand that **independence of testing** exists on a spectrum running from lower to higher:

Independence of testing Separation of responsibilities, which encourages the accomplishment of objective testing.

- When tests are designed and possibly even executed by the software work product developer who wrote the work product, this is a low level of independence. As mentioned above, this kind of testing is necessary, but we should have no illusions about achieving a high level of defect detection effectiveness with such tests.
- When tests are designed by someone other than the author, but someone from within the development team, this does reduce the author bias to some extent. However, group-think can exist within such teams, limiting the independence of thought and the effectiveness of test design. In addition, as noted above, there may be a lack of knowledge about how to test well.
- When tests are designed by people from a different organizational group, such as an independent test team or tester who specialize in usability or performance testing, this is a high level of independence. As mentioned earlier, such test teams can be highly effective defect detectors.
- When tests are designed and executed by people from a different organization or company, such as an outsource testing service provider or system certification body (e.g. the FDA for pharmaceutical systems), this provides the highest level of independence.

We'll revisit this issue of tester independence in Chapter 5.

Independence by itself doesn't guarantee successful testing. Testing activities need clearly defined objectives, too, as people involved in testing will tend to look to such objectives for direction during planning and other activities in the test process. For example, if the main objective of user acceptance testing is to gain confidence that the most-used business processes are working smoothly, a tester who thinks the main objective is to find defects may concentrate on obscure areas of the system instead of the main user paths. The tester may think they are doing the right thing, but in fact they are not. This is why test objectives should be clearly defined in a **testing policy**, ideally with effectiveness and efficiency measures and goals for each objective, and these goals clearly communicated to the testers.

Test policy A high level document describing the principles, approach and major objectives of the organization regarding testing.

Now, as we mentioned earlier, independence from the developers doesn't mean an adversarial relationship with them. In fact, such a relationship is toxic – often fatally so – to a test team's effectiveness. For example, if testers are not careful about the way they communicate, developers may perceive the identification of defects

(in reviews) and failures (during test execution) as a criticism of their work and in fact of them personally. This can result in a perception that testing is a destructive activity.

It is a particular problem when testers revel 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 must see themselves as teammates. In their special role, testers provide essential services within 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 the testing team, and how well are we doing?'

While the specific services are not always defined, it is common that mature development team members see testing as a constructive activity that helps the organization manage its quality risks. In addition, 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. Working with stakeholders to define these services is an essential part of defining the objectives, measures and goals for the test policy document mentioned earlier.

Such clearly defined objectives and goals, combined with constructive styles of communication on the part of test professionals, will help to avoid any negative personal or group dynamics between testers and their colleagues in the development team. Whenever defects are found by independent testers, whether during reviews, static analysis, or dynamic testing, true testing professionals distinguish themselves by demonstrating good interpersonal skills. True testing professionals communicate facts about defects, progress and risks in a constructive way. While this is not necessary, we have noticed that many of consummate testing professionals have business analysts, system designers, architects, programmers, and other developers with whom they work as close personal friends.

Certainly, 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 teammates:

- First of all, 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 don't 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 must work together to accomplish that.

- Next, 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's not really any harder to communicate your findings, especially the potentially embarrassing findings, in a neutral, fact-focused way. In fact, you'll find that, if you avoid criticizing people and their work products, but instead keep your written and verbal communications objective and factual, you also will avoid a lot of unnecessary conflict and drama with your colleagues.
- In addition, 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?
- Finally, 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's quite possible that you are misunderstanding someone's reaction to your findings. So, before assuming the worst about someone and their motivations, instead confirm that the other person has understood what you have said and vice versa.

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.

1.6 CODE OF ETHICS

SYLLABUS LEARNING OBJECTIVES FOR 1.6 CODE OF ETHICS

Note: At this time, no learning objectives are provided for this section, nor is an overall section K-level defined. You should remember the code of ethics.

In this section, we'll briefly introduce you to the ISTQB code of ethics. There are no Syllabus terms for this section.

As a software tester, as in any other profession, you will from time to time encounter ethical challenges. On the one hand, you are likely to have access to confidential and privileged information, or to be in the position to harm someone's interests. On the other hand, you are likely to have opportunities to advance good causes. So, a code of ethics will help guide your decisions and choose the best-possible outcome.

The ISTQB has derived its code of ethics from the ACM and IEEE code of ethics. The ISTQB code of ethics is shown in Table 1.2.

TABLE 1.2 ISTQB Code of Ethics

PUBLIC	Certified software testers shall act consistently with the public interest.
CLIENT AND EMPLOYER	Certified software testers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest.
PRODUCT	Certified software testers shall ensure that the deliverables they provide (on the products and systems they test) meet the highest professional standards possible.
JUDGMENT	Certified software testers shall maintain integrity and independence in their professional judgment.
MANAGEMENT	Certified software test managers and leaders shall subscribe to and promote an ethical approach to the management of software testing.
PROFESSION	Certified software testers shall advance the integrity and reputation of the profession consistent with the public interest.
COLLEAGUES	Certified software testers shall be fair to and supportive of their colleagues, and promote cooperation with software developers.
SELF	Certified software testers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

CHAPTER REVIEW

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

From Section 1.1, you should now be able to explain why testing is necessary and support that explanation with examples and evidence. You should be able to give examples of negative consequences of a software defect or bug for people, companies, and the environment. You should be able to contrast a defect with its symptoms, the anomalies caused by defects. You should be able to discuss the ways in which testing fits into and supports higher quality. You should know the glossary terms **bug, defect, error, fails (false-fail result, false-positive result), failure, fault, mistake, passed (false-negative result, false-pass result), quality** and **risk**.

From Section 1.2, you should now know what testing is. You should be able to remember the common objectives of testing. You should be able to describe how testing can find defects, provide confidence and information and prevent defects. You should know the glossary terms **confirmation testing, debugging, re-testing, requirement, review, test case, test control, test design specification, testing** and **test objective.**

You should be able to explain the fundamental principles of testing, discussed in Section 1.3. You should know the glossary terms **complete testing, exhaustive testing** and **test strategy**.

From Section 1.4, you should now recognize the fundamental test process. You should be able to recall the main testing activities related to test planning and control, analysis and design, implementation and execution, evaluating exit criteria and reporting, and test closure. You should know the glossary terms **coverage (test coverage), exit criteria, incident, regression testing, test approach, test basis, test condition, test data, test execution, test log, test monitoring, test plan, test procedure, test suite, test summary report** and **testware**.

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

Finally, from section 1.6, you should understand the ISTQB Code of Ethics.

SAMPLE EXAM QUESTIONS

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

a. To build confidence in the application.

b. To detect bugs in the application.

c. To gather evidence for a lawsuit.

d. To train the users.

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

a. Incident.

b. Defect.

c. Mistake.

d. Error.

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

a. Negative feedback to the tester.

b. Negative consequences that will occur.

c. Negative consequences that could occur.

d. Negative consequences for the test object.

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

a. Preventing defects in the system.

b. Finding defects through dynamic testing.

c. Gaining confidence in the system.

d. Finishing the project on time.

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

a. Exhaustive testing is impossible.

b. Defect clustering.

c. Pesticide paradox.

d. Absence-of-errors fallacy.

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

a. To verify the success of corrective actions.

b. To prevent a task from being incorrectly considered completed.

c. To ensure that defects have not been introduced by a modification.

d. To motivate better unit testing by the programmers.

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

a. Understanding what managers value about testing.

b. Explaining test results in a neutral fashion.

c. Identifying potential customer work-arounds for bugs.

d. Promoting better quality software whenever possible.

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

a. Test principles only affect the preparation for testing.

b. Test principles only affect test execution activities.

c. Test principles affect the early test activities such as review.

d. Test principles affect activities throughout the test life cycle.

ANSWERS TO SAMPLE EXAM QUESTIONS

This section contains the answers and the learning objectives for the sample questions in each chapter and for the full mock paper in Chapter 7.

If you get any of the questions wrong or if you weren't sure about the answer, then the learning objective tells you which part of the Syllabus to go back to in order to help you understand why the correct answer is the right one. The learning objectives are listed at the beginning of each section. For example, if you got Question 4 in Chapter 1 wrong, then go to Section 1.2 and read the first learning objective. Then re-read the part of the chapter that deals with that topic.

CHAPTER 1 FUNDAMENTALS OF TESTING

Question	Answer	Learning objective
1	A	1.1.3
2	B	1.1.5
3	C	1.1.5
4	A	1.2.1
5	A	1.2.3
6	C	1.4.1
7	B	1.5.1
8	D	1.2.3, 1.3.1 and 1.4.1

This page contains answers for this chapter only

REFERENCES

Key:
In syllabus
Extra to syllabus

PREFACE

ISTQB, www.istqb.org
ISTQB, *Certified Tester: Foundation Level Syllabus* (Version 2011)

CHAPTER 1 FUNDAMENTALS OF TESTING

Beizer, B. (1990) *Software Testing Techniques* (2nd edition), Van Nostrand Reinhold: Boston
Black, R. (2009) *Managing the Testing Process* (3rd edition), John Wiley & Sons: New York
Black, R. (2004) *Critical Testing Processes*, Addison Wesley: Reading MA
Gilb, T. (1993) *Software Inspection*, Addison Wesley: Reading, MA
ISO/IEC 9126–1: 2001, Software Engineering – Software Product Quality
Jones, C. (2008) *Estimating Software Costs, 3e*, McGraw Hill Education: New York

This page contains answers for this chapter only