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).

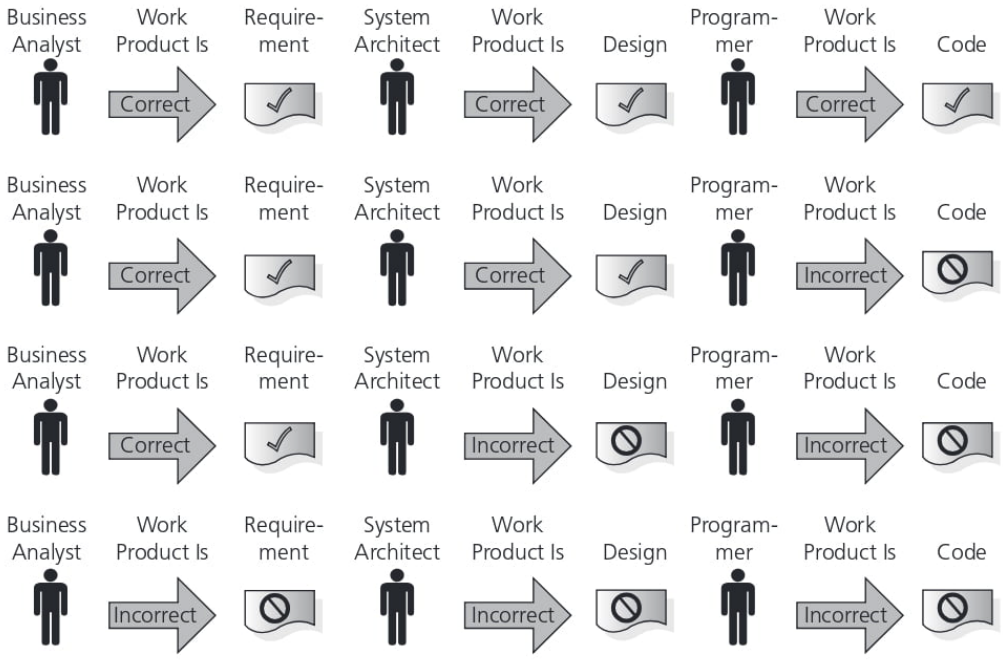

FIGURE 1.1 Four typical scenarios

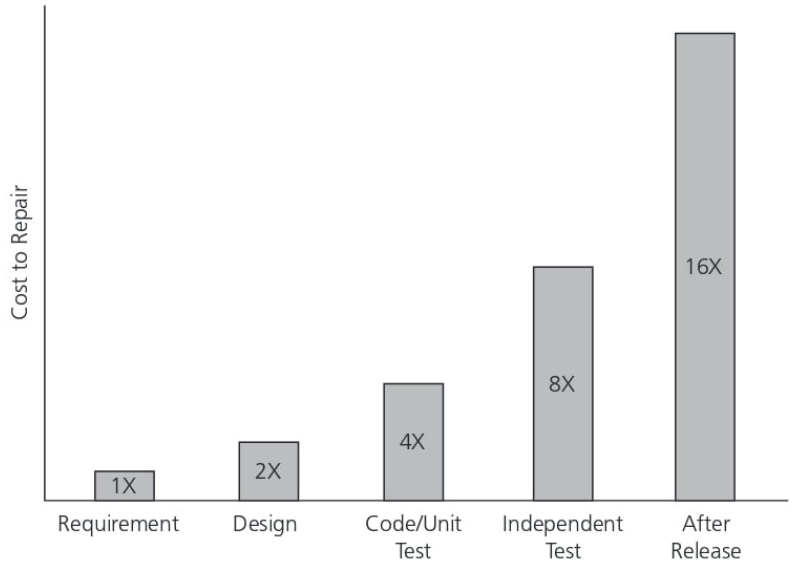

FIGURE 1.2 Multiplicative increases in cost

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.