CHAPTER 2

Testing throughout the software life cycle

Testing is not a stand-alone activity. It has its place within a software development life cycle model and therefore the life cycle applied will largely determine how testing is organized. There are many different forms of testing. Because several disciplines, often with different interests, are involved in the development life cycle, it is important to clearly understand and define the various test levels and types. This chapter discusses the most commonly applied software development models, test levels and test types. Maintenance can be seen as a specific instance of a development process. The way maintenance influences the test process, levels and types and how testing can be organized is described in the last section of this chapter.

2.1 SOFTWARE DEVELOPMENT MODELS

1 **Understand the relationship between development, test activities and work products in the development life cycle and give examples based on project and product characteristics and context. (K2)**
2 **Recognize the fact that software development models must be adapted to the context of project and product characteristics. (Kl)**
3 **Recall reasons for different levels of testing and characteristics of good testing in any life cycle model. (Kl)**

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

The life cycle model that is adopted for a project will have a big impact on the testing that is carried out. Testing does not exist in isolation; test activities

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

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

2.1.1 V-model

Before discussing the **V-model,** we will look at the model which came before it. The waterfall model was one of the earliest models to be designed. It has a natural timeline where tasks are executed in a sequential fashion. We start at the top of the waterfall with a feasibility study and flow down through the various project tasks finishing with implementation into the live environment. Design flows through into development, which in turn flows into build, and finally on into test. Testing tends to happen towards the end of the project life cycle so defects are detected close to the live implementation date. With this model it has been difficult to get feedback passed backwards up the waterfall and there are difficulties if we need to carry out numerous iterations for a particular phase.

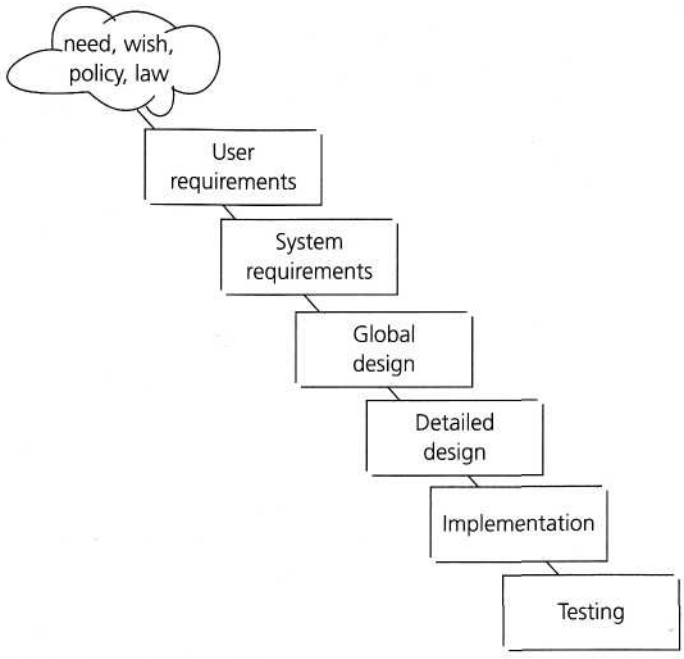

FIGURE 2.1 Waterfall model

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

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

- component testing: searches for defects in and verifies the functioning of software components (e.g. modules, programs, objects, classes etc.) that are separately testable;
- integration testing: tests interfaces between components, interactions to dif ferent parts of a system such as an operating system, file system and hard ware or interfaces between systems;
- system testing: concerned with the behavior of the whole system/product as defined by the scope of a development project or product. The main focus of system testing is verification against specified requirements;
- acceptance testing: validation testing with respect to user needs, require ments, and business processes conducted to determine whether or not to accept the system.

The various test levels are explained and discussed in detail in Section 2.2.

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

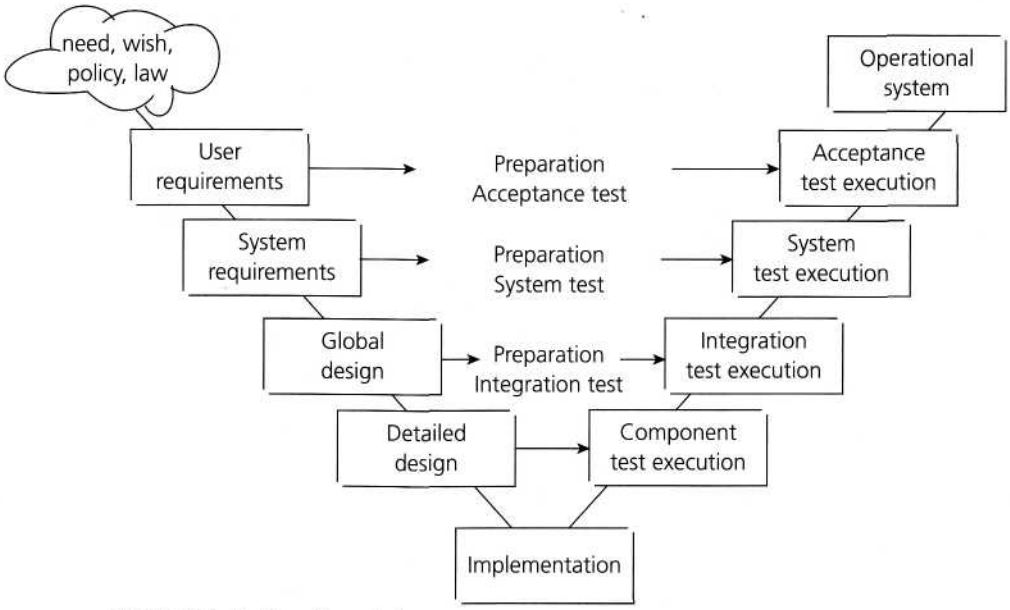

FIGURE 2.2 V-model

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

2.1.2 Iterative life cycles

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

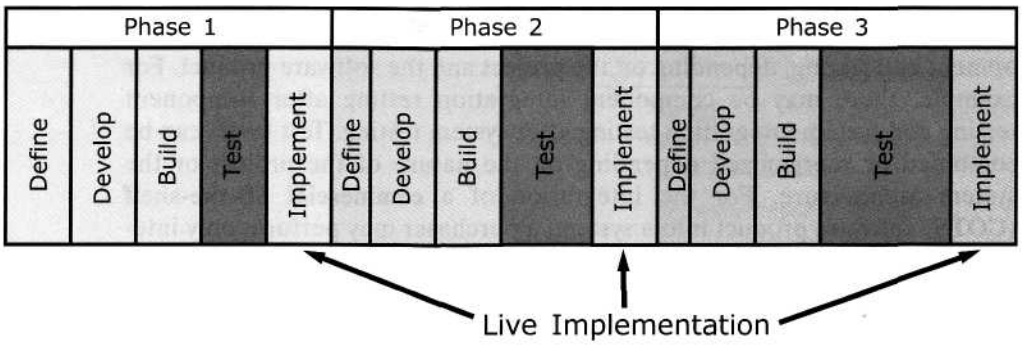

FIGURE 2.3 Iterative development model

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

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

Rapid Application Development

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

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

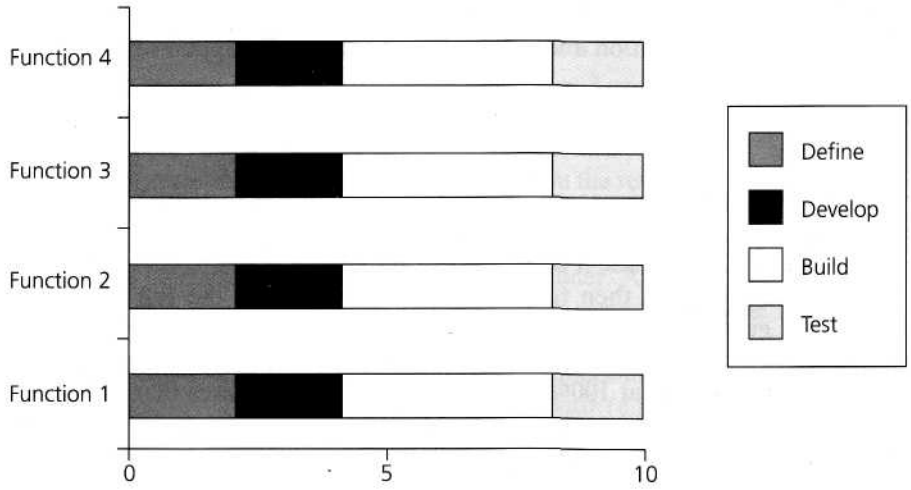

FIGURE 2.4 RAD model

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

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

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

Agile development

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

- It promotes the generation of business stories to define the functionality.
- It demands an on-site customer for continual feedback and to define and carry out functional acceptance testing.
- It promotes pair programming and shared code ownership amongst the developers.
- It states that component test scripts shall be written before the code is written and that those tests should be automated.
- It states that integration and testing of the code shall happen several times a day.
- It states that we always implement the simplest solution to meet today's problems.

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

2.1.3 Testing within a life cycle model

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

- for every development activity there is a corresponding testing activity;
- each test level has test objectives specific to that level;
- the analysis and design of tests for a given test level should begin during the corresponding development activity;
- testers should be involved in reviewing documents as soon as drafts are avail able in the development cycle.

2.2 TEST LEVELS

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

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

2.2.1 Component testing

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

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

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

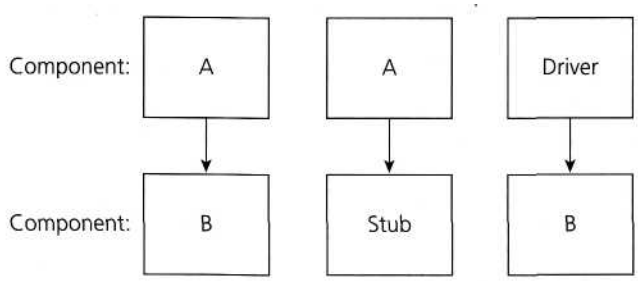

FIGURE 2.5 Stubs and drivers

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

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

2.2.2 Integration testing

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

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

- component integration testing tests the interactions between software com ponents and is done after component testing;
- system integration testing tests the interactions between different systems and may be done after system testing. In this case, the developing organiza tion may control only one side of the interface, so changes may be destabi lizing. Business processes implemented as workflows may involve a series of systems that can even run on different platforms.

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

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

- Top-down: testing takes place from top to bottom, following the control flow or architectural structure (e.g. starting from the GUI or main menu). Components or systems are substituted by stubs.
- Bottom-up: testing takes place from the bottom of the control flow upwards. Components or systems are substituted by drivers.
- Functional incremental: integration and testing takes place on the basis of the functions or functionality, as documented in the functional specification.

The preferred integration sequence and the number of integration steps required depend on the location in the architecture of the high-risk interfaces. The best choice is to start integration with those interfaces that are expected to cause most problems. Doing so prevents major defects at the end of the integration test stage. In order to reduce the risk of late defect discovery, integration should normally be incremental rather than 'big-bang'. Ideally testers should understand the architecture and influence integration planning. If integration tests are planned before components or systems are built, they can be developed in the order required for most efficient testing.

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

2.2.3 System testing

System testing is concerned with the behavior of the whole system/product as defined by the scope of a development project or product. It may include tests based on risks and/or requirements specification, business processes, use cases, or other high level descriptions of system behavior, interactions with the operating system, and system resources. System testing is most often the final test on behalf of development to verify that the system to be delivered meets the specification and its purpose may be to find as many defects as possible. Most often

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

System testing should investigate both **functional** and **non-functional requirements** of the system. Typical non-functional tests include performance and reliability. Testers may also need to deal with incomplete or undocumented requirements. System testing of functional requirements starts by using the most appropriate specification-based (black-box) techniques for the aspect of the system to be tested. For example, a decision table may be created for combinations of effects described in business rules. Structure-based (white-box) techniques may also be used to assess the thoroughness of testing elements such as menu dialog structure or web page navigation (see Chapter 4 for more on the various types of technique).

System testing requires a controlled **test environment** with regard to, amongst other things, control of the software versions, testware and the test data (see Chapter 5 for more on configuration management). A system test is executed by the development organization in a (properly controlled) environment. The test environment should correspond to the final target or production environment as much as possible in order to minimize the risk of environment-specific failures not being found by testing.

2.2.4 Acceptance testing

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

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

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

- A Commercial Off The Shelf (COTS) software product may be acceptance tested when it is installed or integrated.
- Acceptance testing of the usability of a component may be done during com ponent testing.

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

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

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

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

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

2.3 TEST TYPES: THE TARGETS OF TESTING

1 Compare four software test types (functional, non-functional, structural and change-related) by example. (K2)

2 Recognize that functional and structural tests occur at any test level. (Kl)

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

4 Identify and describe test types based on the analysis of a software system's structure or architecture. (K2)

5 Describe the purpose of confirmation testing and regression testing. (K2)

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

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

2.3.1 Testing of function (functional testing)

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

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

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

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

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

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

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

2.3.2 Testing of software product characteristics (non-functional testing)

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

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

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

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

The characteristics and their sub-characteristics are, respectively:

- functionality, which consists of five sub-characteristics: suitability, accuracy, security, interoperability and compliance; this characteristic deals with func tional testing as described in Section 2.3.1;
- **reliability,** which is defined further into the sub-characteristics maturity (robustness), fault-tolerance, recoverability and compliance;
- **usability,** which is divided into the sub-characteristics understandability, learnability, operability, attractiveness and compliance;
- **efficiency,** which is divided into time behavior (performance), resource uti lization and compliance;
- **maintainability,** which consists of five sub-characteristics: analyzability, changeability, stability, testability and compliance;
- **portability,** which also consists of five sub-characteristics: adaptability, installability, co-existence, replaceability and compliance.

2.3.3 Testing of software structure/architecture (structural testing)

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

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

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

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

2.3.4 Testing related to changes (confirmation and regression testing)

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

Confirmation testing (re-testing)

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

When doing confirmation testing, it is important to ensure that the test is executed in exactly the same way as it was the first time, using the same inputs, data and environment. If the test now passes does this mean that the software is now correct? Well, we now know that at least one part of the software is correct - where the defect was. But this is not enough. The fix may have introduced or uncovered a different defect elsewhere in the software. The way to detect these 'unexpected side-effects' of fixes is to do regression testing.

Regression testing

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

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

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

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

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

2.4 MAINTENANCE TESTING

1 **Compare maintenance testing (testing an operational system) to testing a new application with respect to test types, triggers for testing and amount of testing. (K2)**

2 **Identify reasons for maintenance testing (modifications, migration and retirement). (K2)**

3 **Describe the role of regression testing and impact analysis in maintenance. (K2)**

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

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

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

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

changes in emphasis. There are several areas where most differences occur, for example regarding the test basis. A 'catching-up' operation is frequently required when systems are maintained. Specifications are often 'missing', and a set of testware relating to the specifications simply does not exist. It may well be possible to carry out this catching-up operation along with testing a new maintenance release, which may reduce the cost. If it is impossible to compile any specifications from which test cases can be written, including expected results, an alternative test basis, e.g. a **test oracle,** should be sought by way of compromise. A search should be made for documentation which is closest to the specifications and which can be managed by developers as well as testers. In such cases it is advisable to draw the customer's attention to the lower test quality which may be achieved. Be aware of possible problems of 'daily production'. In the worst case nobody knows what is being tested, many test cases are executing the same scenario and if an incident is found it is often hard to trace it back to the actual defect since no traceability to test designs and/or requirements exists. Note that reproducibility of tests is also important for maintenance testing.

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

2.4.1 Impact analysis and regression testing

Usually maintenance testing will consist of two parts:

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

In addition to testing what has been changed, maintenance testing includes extensive regression testing to parts of the system that have not been changed. A major and important activity within maintenance testing is impact analysis. During **impact analysis,** together with stakeholders, a decision is made on what parts of the system may be unintentionally affected and therefore need careful regression testing. Risk analysis will help to decide where to focus regression testing - it is unlikely that the team will have time to repeat all the existing tests.

If the test specifications from the original development of the system are kept, one may be able to reuse them for regression testing and to adapt them for changes to the system. This may be as simple as changing the expected results for your existing tests. Sometimes additional tests may need to be built. Extension or enhancement to the system may mean new areas have been specified and tests would be drawn up just as for the development. It is also possible that updates are needed to an automated test set, which is often used to support regression testing.

2.4.2 Triggers for maintenance testing

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

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

Planned modifications

The following types of planned modification may be identified:

- perfective modifications (adapting software to the user's wishes, for instance by supplying new functions or enhancing performance);
- adaptive modifications (adapting software to environmental changes such as new hardware, new systems software or new legislation);
- corrective planned modifications (deferrable correction of defects).

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

Ad-hoc corrective modifications

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

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

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

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

CHAPTER REVIEW

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

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

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

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

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

SAMPLE EXAM QUESTIONS

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

a. Early test analysis and design.

b. Different test levels are defined with specific objectives.

c. Testers will start to get involved as soon as coding is done.

d. A and B above.

Question 2 Which option best describes objectives for test levels with a life cycle model?

a. Objectives should be generic for any test level.

b. Objectives are the same for each test level.

c. The objectives of a test level don't need to be defined in advance.

d. Each level has objectives specific to that level.

Question 3 Which of the following is a test type?

a. Component testing

b. Functional testing

c. System testing

d. Acceptance testing

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

a. Feasibility

b. Usability

c. Maintenance

d. Regression

Question 5 Which of these is a functional test?

a. Measuring response time on an on-line booking system.

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

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

d. Checking how easy the system is to use.

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

a. There is no time to test the change before it goes live, so only the best developers should do this work and should not involve testers as they slow down the process.

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

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

d. Retest the changed area and then use risk assessment to decide on a reasonable subset of the whole regression test to run in case other parts of the system have been adversely affected.

Question 7 A regression test:

a. Is only run once.

b. Will always be automated.

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

d. Will check changed areas of the software to see if they have been affected.

Question 8 Non-functional testing includes:

a. Testing to see where the system does not function correctly.

b. Testing the quality attributes of the system including reliability and usability.

c. Gaining user approval for the system.

d. Testing a system feature using only the software required for that function.

Question 9 Beta testing is:

a. Performed by customers at their own site.

b. Performed by customers at the software developer's site.

c. Performed by an independent test team.

d. Useful to test software developed for a specific customer or user.