2 LIFE CYCLES

Angelina Samaroo

INTRODUCTION

In the previous chapter, we looked at testing as a concept – what it is and why we should do it. In this chapter, we will look at testing as part of the overall software development process. Clearly, testing does not take place in isolation; there must be a product first.

We will refer to work products and products. A work product is an intermediate deliverable required to create the final product. Work products can be documentation or code. The code and associated documentation will become the product when the system is declared ready for release. In software development, work products are generally created in a series of defined stages, from capturing a customer requirement, to creating the system, to delivering the system. These stages are usually shown as steps within a Software Development Life Cycle.

In this chapter, we will look at two life cycle models – sequential and iterative. For each one, the testing process will be described and the objectives at each stage of testing explained.

Finally, we will look at the different types of testing that can take place throughout the development life cycle.

Learning objectives

The learning objectives for this chapter are listed below. You can confirm that you have achieved these by using the self-assessment questions at the start of the chapter, the 'Check of understanding' boxes distributed throughout the text and the example examination questions provided at the end of the chapter. The chapter summary will remind you of the key ideas.

The sections are allocated a K number to represent the level of understanding required for that section; where an individual topic has a lower K number than the section as a whole, this is indicated for that topic; for an explanation of the K numbers, see the **Introduction**.

Software development lifecycle models (K2)

- FL-2.1.1 Explain the relationships between software development activities and test activities in the software development lifecycle.
- FL-2.1.2 Identify reasons why software development lifecycle models must be adapted to the context of project and product characteristics (K1).

Test levels (K2)

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

Test types (K2)

- FL-2.3.1 Compare functional, non-functional, and white-box testing.
- FL-2.3.2 Recognize that functional, non-functional, and white-box tests occur at any test level. (K1)
- FL-2.3.3 Compare the purposes of confirmation testing and regression testing.

Maintenance testing (K2)

- FL-2.4.1 Summarize triggers for maintenance testing.
- FL-2.4.2 Describe the role of impact analysis in maintenance testing.

Self-assessment questions

The following questions have been designed to enable you to check your current level of understanding for the topics in this chapter. The answers are at the end of the chapter.

Question SA1 (K2)
Which of the following is true of the V model?

a. Coding starts as soon as each function in a system has been defined.
b. The test activities occur after all development activities have been completed.
c. It enables the production of a working version of the system as early as possible.
d. It enables test planning to start as early as possible.

Question SA2 (K2)
Which of the following is true of white-box testing?

a. It is carried out only by developers.
b. It can be used to test data file structures.
c. It is used only at unit and integration test levels.
d. Coverage achieved using white-box test techniques is not measurable.

Question SA3
Which of the following is a test object for integration testing?

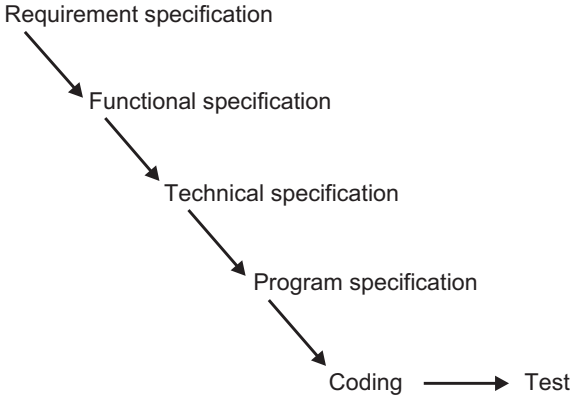
a. A sub-system.
b. An epic.
c. A risk analysis report.
d. A sequence diagram.

SOFTWARE DEVELOPMENT MODELS

A development life cycle for a software product involves capturing the initial requirements from the customer, expanding on these to provide the detail required for code production, writing the code and testing the product, ready for release.

A simple development model is shown in Figure 2.1. This is known traditionally as the waterfall model.

Figure 2.1 Waterfall model



The waterfall model in Figure 2.1 shows the steps in sequence, where the customer requirements are progressively refined to the point where coding can take place. This type of model is often referred to as a linear or sequential model. Each work product or activity is completed before moving on to the next.

In the waterfall model, testing is carried out once the code has been fully developed. Once this is completed, a decision can be made on whether the product can be released into the live environment.

This model for development shows how a fully tested product can be created, but it has a significant drawback: what happens if the product fails the tests? Let us look at a simple case study.

CASE STUDY – DEVELOPMENT PROCESS

Let us consider the manufacture of a smartphone. Smartphones have become an essential part of daily life for many. They must be robust enough to withstand the rigours of being thrown into bags or on floors and must be able to respond quickly to commands.

Many phones now have touchscreens. This means that the apps on the phone must be accessible via a tap on the screen. This is done via a touchscreen driver. The driver is a piece of software that sits between the screen (hardware) and the apps (software), allowing the app to be accessed from a tap on an icon on the screen.

If a waterfall model were to be used to manufacture and ship a touchscreen phone, then all functionality would be tested at the very end, just prior to shipping.

If it is found that the phone can be dropped from a reasonable height without breaking, but that the touchscreen driver is defective, then the phone will have failed in its core required functionality. This is a very late stage in the life cycle to uncover such a fault.

In the waterfall model, the testing at the end serves as a quality check. The product can be accepted or rejected at this point. In the smartphone manufacturing example, this model could be adopted to check that the phone casings after manufacture are crack free, rejecting those that have failed.

In software development, however, it is unlikely that we can simply reject the parts of the system found to be defective and release the rest. The nature of software functionality is such that removal of software is often not a clear-cut activity – this action could cause other areas to function incorrectly. It might even cause the system to become unusable. If the touchscreen driver is not functioning correctly, then some of the apps might not be accessible via a tap on the icon. On a touchscreen phone, this would be an intolerable fault in the live environment.

What is needed is a process that assures quality throughout the development life cycle. At every stage, a check should be made that the work product for that stage meets its objectives. This is a key point: work product evaluation taking place at the point where the product has been declared complete by its creator. If the work product passes its evaluation (test), we can progress to the next stage in confidence. In addition, finding problems at the point of creation should make fixing any problems cheaper than fixing them at a later stage. This is the cost escalation model, described in **Chapter 1**.

The checks throughout the life cycle include verification and validation.

Verification – checks that the work product meets the requirements set out for it. An example of this is to ensure that a website being built follows the guidelines for making websites usable by as many people as possible. Verification helps to ensure that we are building the product in the right way.

Validation – changes the focus of work product evaluation to evaluation against user needs. This means ensuring that the behaviour of the work product matches the customer needs as defined for the project. For example, for the same website above, the guidelines may have been written with people familiar with websites in

mind. It may be that this website is also intended for novice users. Validation would include these users checking that they too can use the website easily. Validation helps to ensure that we are building the right product as far as the users are concerned.

There are two types of development model that facilitate early work product evaluation.

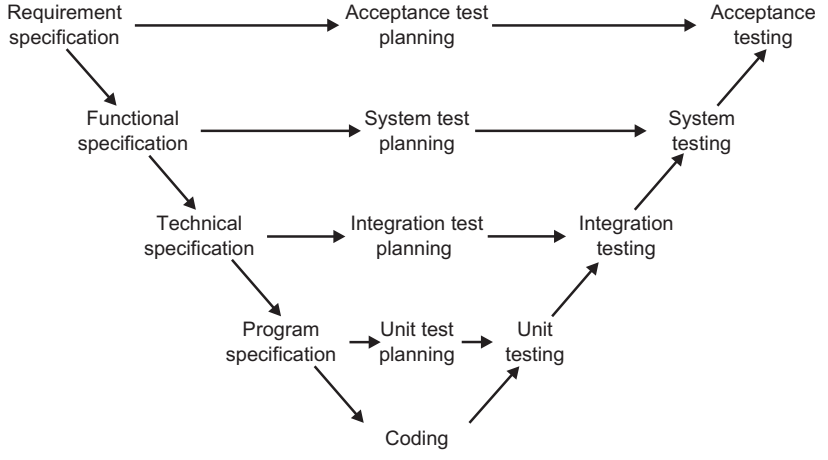
The first is an extension to the waterfall model, known as the V model. The second is a cyclical model, where the coding stage often begins once the initial user needs have been captured. Cyclical models are often referred to as iterative models.

We will consider first the V model.

V model (sequential development model)

There are many variants of the V model. One of these is shown in Figure 2.2.

Figure 2.2 V model for software development



As for the waterfall model, the left-hand side of the model focuses on elaborating the initial requirements, providing successively more technical detail as the development progresses. In the model shown, these are:

- Requirement specification – capturing of user needs.
- Functional specification – definition of functions required to meet user needs.

- Technical specification – technical design of functions identified in the functional specification.
- Program specification – detailed design of each module or unit to be built to meet required functionality.

These specifications could be reviewed to check for the following:

- Conformance to the previous work product (so in the case of the functional specification, verification would include a check against the requirement specification).
- That there is sufficient detail for the subsequent work product to be built correctly (again, for the functional specification, this would include a check that there is sufficient information in order to create the technical specification).
- That it is testable – is the detail provided sufficient for testing the work product?

Formal methods for reviewing documents are discussed in **Chapter 3**.

The middle of the V model shows that planning for testing should start with each work product. Thus, using the requirement specification as an example, acceptance testing is planned for, right at the start of the development. Test planning is discussed in more detail in **Chapter 5**.

The right-hand side focuses on the testing activities. For each work product, a testing activity is identified. These are shown in Figure 2.2:

- Testing against the requirement specification takes place at the acceptance testing stage.
- Testing against the functional specification takes place at the system testing stage.
- Testing against the technical specification takes place at the integration testing stage.
- Testing against the program specification takes place at the unit testing stage.

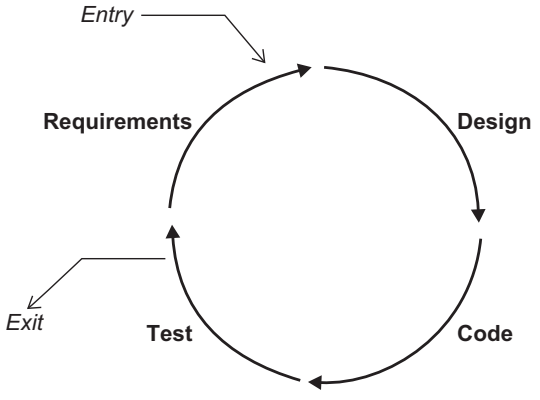
This allows testing to be concentrated on the detail provided in each work product, so that defects can be identified as early as possible in the life cycle, when the work product has been created. The different stages of testing are discussed later.

Remembering that each stage must be completed before the next one can be started; this approach to software development pushes validation of the system by the user representatives right to the end of the life cycle. If the customer needs were not captured accurately in the requirement specification, or if they change, then these issues may not be uncovered until the user testing is carried out. As we saw in **Chapter 1**, fixing problems at this stage could be very costly; in addition, it is possible that the project could be cancelled altogether.

Iterative–incremental development models

Let us now look at a different model for software development – iterative development. This is one where the requirements do not need to be fully defined before coding can start. Instead, a working version of the product is built, in a series of stages, or iterations – hence the name iterative/incremental development. Each stage encompasses requirements definition, design, code and test. This is shown diagrammatically in Figure 2.3.

Figure 2.3 Iterative development



This type of development is often referred to as cyclical – we go 'round the development cycle a number of times', within the project. The project will have a defined timescale and cost. Within this, the cycles will be defined. Each cycle will also have a defined timescale and cost. The cycles are commonly referred to as time-boxes. For each time-box, a requirement is defined and a version of the code is produced, which will allow testing by the user representatives. At the end of each time-box, a decision is made on what extra functionality needs to be created for the next iteration. This process is then repeated until a fully working system has been produced.

Some models incorporate the idea of 'self-organising' teams. This does not mean that the team is leaderless, rather that the team decides how to best manage and execute the tasks amongst themselves. This will of course include the relationship between testers and developers, and how defects are reported.

A key feature of this type of development is the involvement of user representatives in the testing. Having the users represented throughout minimises the risk of developing an unsatisfactory product. The user representatives are empowered to request changes to the software, to meet their needs.

Components or systems developed using these methods often involve overlapping and iterating test levels throughout development. Ideally, each feature is tested at several

test levels before delivery. This is often facilitated by continuous delivery or deployment, enabled by making use of significant automation.

This approach to software development can pose problems, however.

The lack of formal documentation can make it difficult to test. To counter this, developers may use test-driven development (TDD). This is where functional tests are written first, and code is then created and tested. It is reworked until it passes the tests.

In addition, the working environment may be such that developers make any changes required, without formally recording them. This approach could mean that changes cannot be traced back to the requirements, nor to the parts of the software that have changed. Thus, traceability as the project progresses is reduced. To mitigate this, a robust process must be put in place at the start of the project to manage these changes (often part of a configuration management process – this is discussed further in **Chapter 5**).

Another issue associated with changes is the amount of testing required to ensure that implementation of the changes does not cause unintended changes to other parts of the software (this is called regression testing, discussed later in this chapter).

Forms of iterative development include Scrum, Kanban, Spiral and the Rational Unified Process (RUP). Agile is an umbrella term incorporating these and other methods.

- Scrum – here the focus is on short iterations spanning just hours, days or a few weeks. The increments developed are thus correspondingly small. This term may already be familiar to those of you already working in an Agile environment.
- Kanban – as for Scrum, you may already be familiar with this term. It allows for easy visualisation of a workflow, via the usual task board used in Agile development projects. It is not a time-boxing tool; it can be used to show progress of a single enhancement or group of features, from a 'to-do' state, to a 'done' state.
- Rational Unified Process – iterations here tend to be longer than in Scrum, with correspondingly larger feature sets. Those of you working in this environment may recall the Inception – Elaboration – Construction – Transition phases.
- Spiral – Dr Barry Boehm (whom you came across in **Chapter 1** when discussing the cost escalation model) created this model. Here, risk is used as the driver for determining the levels of documentation and effort required for a given project. This can include a prototyping model, where increments created may be reworked significantly or even abandoned if the risks are too high.

Agile methods of developing software have gained significant ground in recent years. Organisations across business sectors have embraced this collaborative way of working and many qualifications focusing on Agile methodologies now exist. The syllabus for this qualification does not dwell on Agile; however, for completeness of learning, a summary will now be provided.

The Agile development methodology is supported through the Agile Alliance, www.agilealliance.org. The Alliance has created an Agile manifesto with four points, supported by 12 principles. The essence of these is to espouse the value of adopting a can-do and collaborative approach to creating a product. The idea is that the development teams work closely with the business, responding to their needs at the time, rather than attempting to adhere to a contract for requirements that might well need to be changed prior to the launch date. Many examples can be provided to suggest that this is a suitable way of working. Going back to our smartphone example, there are many well-known phone manufacturers who failed to move with consumer demands, costing them significant market share.

A popular framework for Agile is Scrum. Scrum is not an acronym; it was taken from the game of rugby. In rugby the team huddles to agree tactics; the ball is then passed back and forth until a sprint to the touchline is attempted. In Scrum, there is a daily stand-up meeting to agree tactics for the day; an agreed set of functions to be delivered at the end of a time-box (Sprint); periodic reviews of functionality by the customer representatives; and a team retrospective to reflect on the previous Sprint in order to improve on the next. In Agile, the term 'user story' is common when referring to requirements, as is the term 'backlog' when referring to a set of requirements or tasks for a particular Sprint.

The ISTQB now offers a qualification in Agile testing as an extension to this Foundation in software testing. Further information can be found at www.istqb.org

CHECK OF UNDERSTANDING

1. What is meant by verification?
2. What is meant by validation?
3. Name three work products typically shown in the V model.
4. Name three activities typically shown in the V model.
5. Identify a benefit of the V model.
6. Identify a drawback of the V model.
7. Name three activities typically associated with an iterative model.
8. Identify a significant benefit of an iterative model.
9. List three challenges of an iterative development.
10. List three types of iterative development.
11. Compare the work products in the V model with those in an iterative model.

For both types of development, testing plays a significant role. Testing helps to ensure that the work products are being developed in the right way (verification) and that the product will meet the user needs (validation).

Characteristics of good testing across the development life cycle include:

- Early test design – in the V model, we saw that test planning begins with the specification documents. This activity is part of the test process, discussed in **Chapter 1**. After test planning, the documents are analysed and test cases designed. This approach ensures that testing starts with the development of the requirements; that is, a proactive approach to testing is undertaken. Proactive approaches to test design are discussed further in **Chapter 5**. As we saw in iterative development, test-driven development may be adopted, pushing testing to the front of the development activity.
- Each work product is tested – in the V model, each document on the left is tested by an activity on the right. Each specification document is called the test basis, that is, it is the basis on which tests are created. In iterative development, the functionality for each iteration is tested before moving on to the next.
- Each test level has objectives specific to that level. Thus, at unit level the focus is on individual pieces of code; at integration level the focus is on the interfaces and so on.
- Testers are involved in reviewing requirements before they are released – in the V model, testers are invited to review associated documents from a testing perspective. Techniques for reviewing documents are outlined in **Chapter 3**.

TEST LEVELS

In Figure 2.2, the test stages of the V model are shown. They are often called test levels. The term test level provides an indication of the focus of the testing, and the types of problems it is likely to uncover. The typical levels of testing are:

- component (unit) testing;
- integration testing;
- system testing;
- acceptance testing.

Each of these test levels will include tests designed to uncover problems specifically at that stage of development. These levels of testing can also be applied to iterative development. In addition, the levels may change depending on the system. For instance, if the system includes some software developed by external parties, or bought off the shelf (commercial off-the shelf (COTS) based), acceptance testing on these may be conducted before testing the system as a whole.

Each test level will have a test basis (a description of the item) and a test object (the item under test). A test basis is some form of definition of what the code is intended to do and is used as a reference for deriving the tests. It can include the requirements; user stories; the source code; or the knowledge of the tester, based on experience. The higher

the level of documentation, the more precise the test design can be. Typically, in V model development, there is more documentation than in iterative development. Techniques for test design will be covered in **Chapter 4**.

Test levels are characterised by the following attributes:

- specific objectives;
- test basis, referenced to derive test cases;
- test object (i.e. what is being tested);
- typical defects and failures;
- specific approaches and responsibilities.

Let us now look at these levels of testing in more detail.

Component (unit) testing

Before testing of the code can start, clearly the code has to be written. This is shown at the bottom of the V model. Generally, the code is written in component parts, or units. The components are usually constructed in isolation, for integration at a later stage. Components are also called programs, modules or units.

Component (unit) testing is often done in isolation from the rest of the system, depending on the Software Development Life Cycle model and the system, which may require mock objects, service virtualisation, harnesses, stubs and drivers.

Component (unit) testing may cover:

- Functional requirements (such as the ability to remove items from a shopping cart).
- Non-functional characteristics (such as checking for memory leaks – this is where the program holds on to memory it is no longer using, which may cause the system to slow down when in use).
- Structural testing – this is checking the percentage of code exercised through testing. Testing based on code (white-box testing) is discussed in **Chapter 4**.

Component (unit) testing is intended to check the quality of the individual piece of code prior to its integration with other units.

- Specific objectives:
  - reducing risk;
  - verifying whether the functional and non-functional behaviours of the component are as designed and specified;
  - building confidence in the component's quality;
  - finding defects in the component;
  - preventing defects from escaping to higher test levels.

- Test bases include:
  - a detailed design;
  - a component specification;
  - a data model or other document describing the expected functionality of the unit;
  - the code itself can be used as a basis for component testing.
- Test objects include:
  - the components;
  - the programs;
  - code and data structures;
  - classes;
  - database modules and other pieces of code.
- Typical defects and failures include:
  - incorrect functionality, perhaps due to incorrect logic – for example introducing too short a time for displaying an error message to a user before removing it, causing them not to see it properly;
  - data flow problems – for instance, a part of the code requests an input, but provides no output;
  - code that contains overly complicated constructs, reducing maintainability of the code.
- Specific approaches and responsibilities:
  - Developers tend to fix defects are soon as they are found.
  - One approach to unit testing is called test-driven development. This originated in eXtreme Programming. As its name suggests, test cases are written first, and then the code is built, tested and changed until the unit passes its tests. This is an iterative approach to unit testing.
  - Unit testing is often supported by a unit test framework, as well as debugging tools. These assist the developer in finding and fixing defects, without the need for a formal defect management process at this stage. If defects are logged and analysed however, they can provide opportunities for root cause analysis to improve the test process for future releases.
  - Regression testing is automated so that issues with a software build can be detected quickly. This is now typical in iterative development models.

Integration testing

Once the units have been written, the next stage is to put them together to create the system. This is called integration. It involves building something larger from a number of smaller pieces.

The purpose of integration testing is to expose defects in the interfaces and in the interactions between integrated components or systems.

There are two different levels of integration testing described in the ISTQB syllabus, which may be carried out on test objects of varying size as follows:

Component integration testing, which focuses on the interactions and interfaces between integrated components. It is performed after component testing and is generally automated. In iterative and incremental development, component integration tests are usually part of the continuous integration process.

System integration testing, which focuses on the interactions and interfaces between systems, packages and microservices (where an application is decomposed into fine-grained services, loosely coupled to make up the system). It can also cover interactions and interfaces with external organisations. For example, a trading system in an investment bank may interact with the stock exchange to get the latest prices for its stocks and shares on the international market. Where external organisations are involved, extra challenges for testing present themselves, since the developing organisation will not have control over the interfaces. This can include creating the test environment, defect resolution and so on.

- Specific objectives:
  - reducing risk;
  - verifying whether the functional and non-functional behaviours of the interfaces are as designed and specified;
  - building confidence in the quality of the interfaces;
  - finding defects (which may be in the interfaces themselves or within the components or systems);
  - preventing defects from escaping to higher test levels.
- Test bases include:
  - software and system design;
  - sequence diagrams;
  - interface and communication protocol specifications;
  - use cases;
  - architecture at component or system level;
  - workflows;
  - external interface definitions.
- Test objects include:
  - sub-systems;
  - databases;
  - infrastructure;

  - interfaces;
  - Application Program Interfaces (APIs);
  - microservices.
- Typical defects and failures for component integration testing include:
  - incorrect or missing data;
  - incorrect data encoding;
  - incorrect sequencing or timing of interface calls;
  - interface mismatches;
  - failures in communication between components;
  - ignored or improperly handled communication failures between components;
  - incorrect assumptions about the meaning, units or boundaries of the data being passed between components.
- Typical defects and failures for system integration testing include:
  - inconsistent message structures between systems;
  - incorrect or missing data;
  - incorrect data encoding (as above) for component integration testing;
  - interface mismatch (as above) for component integration testing;
  - failures in communication between systems;
  - ignored or improperly handled communication failures between systems;
  - incorrect assumptions about the meaning, units or boundaries of the data being passed between systems;
  - failure to comply with mandatory security regulations.
- Specific approaches and responsibilities.
  - Component integration testing is usually carried out by developers.
  - System integration testing may be done after system testing or in parallel with ongoing system test activities (in both sequential development and iterative and incremental development). It is usually carried out by testers.
  - Continuous integration, where software is integrated on a component-by-component basis (i.e. functional integration), is now commonplace. This allows integration defects to be found as soon as they are introduced.
  - Regression testing is often automated, as we saw for component testing.

Before integration testing can be planned, an integration strategy is required. This involves making decisions on how the system will be put together in a systematic way prior to testing.

Systematic integration strategies may be based on the system architecture (e.g. top-down and bottom-up), functional tasks, transaction processing sequences or some other aspect of the system or components.

There are three commonly quoted integration strategies, as follows.

Big-bang integration

This is where all units are linked at once, resulting in a complete system. When the testing of this system is conducted, it is difficult to isolate any errors found because attention is not paid to verifying the interfaces across individual units.

This type of integration is generally regarded as a poor choice of integration strategy. It introduces the risk that problems may be discovered late in the project, when they are more expensive to fix.

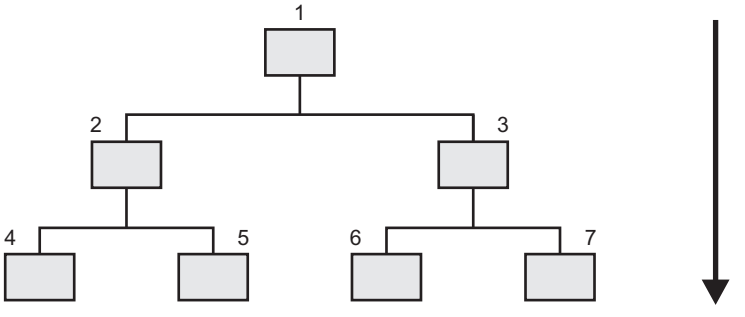
Top-down integration

This is where the system is built in stages, starting with components that, when activated, cause other components to become active. These are called 'calling' components. Components that call others are usually placed above those that are called. Top-down integration testing permits the tester to evaluate component interfaces, starting with those at the 'top'.

Let us look at the diagram in Figure 2.4 to explain this further.

The control structure of a program can be represented in a chart. In Figure 2.4, component 1 can call components 2 and 3. Thus in the structure, component 1 is placed above components 2 and 3. Component 2 can call components 4 and 5. Component 3 can call components 6 and 7. Thus in the structure, components 2 and 3 are placed above components 4 and 5 and components 6 and 7, respectively.

Figure 2.4 Top-down control structure



In this chart, the order of integration might be:

- 1,2
- 1,3
- 2,4
- 2,5
- 3,6
- 3,7

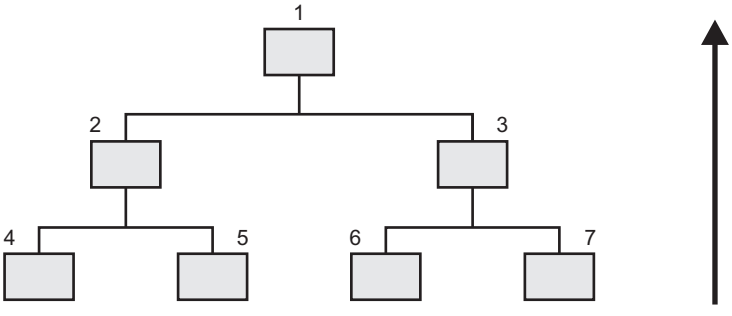
Top-down integration testing requires that the interactions of each component must be tested when they are built. Those lower down in the hierarchy may not have been built or integrated yet. In Figure 2.4, in order to test component 1's interaction with component 2, it may be necessary to replace component 2 with a substitute since component 2 may not have been integrated yet. This is done by creating a skeletal implementation of the component, called a stub. A stub is a passive component, called by other components. In this example, stubs may be used to replace components 4 and 5 when testing component 2.

The use of stubs is commonplace in top-down integration, replacing components not yet integrated.

Bottom-up integration

This is the opposite of top-down integration and the components are integrated in a bottom-up order. This is shown in Figure 2.5.

Figure 2.5 Bottom-up integration



The integration order might be:

- 4,2
- 5,2

- 6,3
- 7,3
- 2,1
- 3,1

So, in bottom-up integration, components 4–7 would be integrated before components 2 and 3. In this case, the components that may not be in place are those that actively call other components. As in top-down integration testing, they must be replaced by specially written components. When these special components call other components, they are called drivers. They are so called because, in the functioning program, they are active, controlling other components. Components 2 and 3 could be replaced by drivers when testing components 4–7. They are generally more complex than stubs.

System testing

Having checked that the components all work together at unit integration level, the next step is to consider the functionality from an end-to-end perspective. This activity is called system testing.

System testing is necessary because many of the criteria for test selection at unit and integration testing result in the production of a set of test cases that are unrepresentative of the operating conditions in the live environment. Thus, testing at these levels is unlikely to reveal errors due to interactions across the whole system, or those due to environmental issues.

System testing serves to correct this imbalance by focusing on the behaviour of the whole system/product as defined by the scope of a development project or programme, in a representative live environment. It is usually carried out by a team that is independent of the development process. The benefit of this independence is that an objective assessment of the system can be made, based on the specifications as written, and not the code.

System testing often produces information that is used by stakeholders to make release decisions, which may include checking that legal or regulatory requirements and standards have been met.

The behaviour required of the system may be documented in functional specifications, use cases or user stories. These should include the functional and non-functional requirements of the system or feature.

A functional requirement is a requirement that specifies a function that a system or system component must perform. Functional requirements can be specific to a system. For instance, you expect to be able to search for flights on a travel agent's website, whereas you visit your online bank to check that you have sufficient funds to pay for the flight. Thus, functional requirements provide detail on what the application being developed will do.

Non-functional system testing looks at those aspects that are important but not directly related to what functions the system performs. These tend to be generic requirements, which can be applied to many different systems. In the example above, you can expect that both systems will respond to your inputs in a reasonable time frame, for instance. Typically, these requirements will consider both normal operations and behaviour under exceptional circumstances. Thus, non-functional requirements detail how the application will perform in use.

Examples of non-functional requirements include:

- installability – installation procedures;
- maintainability – ability to introduce changes to the system;
- performance – expected normal behaviour;
- load handling – behaviour of the system under increasing load;
- stress handling – behaviour at the upper limits of system capability;
- portability – use on different operating platforms;
- recovery – recovery procedures on failure;
- reliability – ability of the software to perform its required functions over time;
- usability – ease with which users can engage with the system.

The amount of testing required at system testing, however, can be influenced by the amount of testing carried out (if any) at the previous stages. In addition, the amount of testing advisable also depends on the amount of verification carried out on the requirements (this is discussed further in **Chapter 3**).

- Specific objectives:
  - reducing risk;
  - verifying whether the functional and non-functional behaviours of the system are as designed and specified;
  - validating that the system is complete and will work as expected;
  - building confidence in the quality of the system as a whole;
  - finding defects;
  - preventing defects from escaping to higher test levels or production.
- Test bases include:
  - system and software requirement specifications (functional and non-functional);
  - risk analysis reports;
  - use cases;
  - epics and user stories;

  - models of system behaviour;
  - state diagrams;
  - system and user manuals.
- Test objects include:
  - applications;
  - hardware/software systems;
  - operating systems;
  - system under test (SUT);
  - system configuration and configuration data (i.e. data that can be configured to suit a particular use or need).
- Typical defects and failures include:
  - incorrect calculations;
  - incorrect or unexpected system behaviour;
  - incorrect control and/or data flows within the system.
- Specific approaches and responsibilities:
  - Testers should be involved in the static review of documents to avoid ambiguities in, and lack of understanding of, requirements. These can lead to the false positives and negatives discussed in **Chapter 1**. Reviews will be explored further in **Chapter 3**.
  - System testing should use the most appropriate techniques (to be covered in **Chapter 4**) for the aspect(s) of the system to be tested.
  - The test environment should mimic the target or production environment as far as practicable.
  - System testing is typically carried out by independent testers.
  - As we saw earlier, automation is often used in regression testing to aid in providing confidence in the eventual system functionality.

Acceptance testing

The purpose of acceptance testing is to provide the end users with confidence that the system will function according to their expectations.

Unlike system testing, however, the testing conducted here should be independent of any other testing carried out. Its key purpose is to demonstrate system conformance to, for example, the customer requirements and operational and maintenance processes. For instance, acceptance testing may assess the system's readiness for deployment and use.

Typical forms of acceptance testing include the following:

- User acceptance testing – testing by user representatives to check that the system meets their business needs. This can include factory acceptance testing, where the system is tested by the users before moving it to their own site. Site acceptance testing could then be performed by the users at their own site.
- Operational acceptance testing – often called operational readiness testing. This involves checking that the processes and procedures are in place to allow the system to be used and maintained. This can include checking:
  - back-up facilities;
  - installing, uninstalling and upgrading;
  - performance testing;
  - procedures for disaster recovery;
  - user management;
  - maintenance procedures;
  - data load and migration tasks;
  - security vulnerabilities.
- Contract and regulatory acceptance testing:
  - Contractual acceptance testing – sometimes the criteria for accepting a system are documented in a contract. Testing is then conducted to check that these criteria have been met, before the system is accepted. This is typical of custom-developed software.
  - Regulatory acceptance testing – in some industries, systems must meet governmental, legal or safety standards. Examples of these are the defence, banking and pharmaceutical industries. The results of tests here may be witnessed or audited by regulatory bodies.
- Alpha and beta testing:
  - Alpha testing takes place at the developer's site – the operational system is tested while still at the developer's site by internal staff, before release to external customers. Note that testing here is still independent of the development team.
  - Beta testing takes place at the customer's site – the operational system is tested by a group of customers, who use the product at their own locations and provide feedback, before the system is released. This is often called 'field testing'.

Both alpha and beta testing are typically used by developers of COTS software in order to get feedback before final go-live.

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

It is worth noting that defect finding is not a main aim of acceptance testing, although they must of course be logged and resolved when found.

- Test bases include:
  - user, business, legal, regulatory and system requirements;
  - use cases and business processes;
  - installation procedures;
  - risk analysis reports.
- Test bases for operational acceptance testing include:
  - back-up, restore and disaster recovery procedures;
  - security standards or regulations;
  - non-functional requirements;
  - operations documentation;
  - deployment and installation instructions;
  - performance targets;
  - database packages.
- Test objects include:
  - system under test;
  - system configuration, configuration and production data;
  - business processes for a fully integrated system;
  - recovery systems and hot sites (for business continuity and disaster recovery testing);
  - operational and maintenance processes;
  - forms and reports.
- Typical defects and failures.
  - System workflows do not meet business or user requirements.
  - Business rules are not implemented correctly.
  - System does not satisfy contractual or regulatory requirements.
  - Non-functional failures such as security vulnerabilities, inadequate performance efficiency under high loads, or improper operation on a supported platform.
- Specific approaches and responsibilities.
  - Acceptance testing is often the responsibility of the customers or users of a system, although other project team members may be involved as well.
  - Acceptance testing is often thought of as the last test level in a sequential development life cycle, but it may also occur at other times; for example:

- When a COTS software product is installed or integrated.
- Before system testing for a new functional enhancement.
- At the end of each iteration in iterative development – for verification against the documented acceptance criteria and validation against the user needs. This can also include alpha and beta testing.

In iterative development, project teams can employ various forms of acceptance testing during and at the end of each iteration, such as those focused on verifying a new feature against its acceptance criteria and those focused on validating that a new feature satisfies the users' needs. In addition, alpha tests and beta tests may occur, either at the end of each iteration, after the completion of each iteration, or after a series of iterations. User acceptance tests, operational acceptance tests, regulatory acceptance tests and contractual acceptance tests also may occur, either at the close of each iteration, after the completion of each iteration, or after a series of iterations.

CHECK OF UNDERSTANDING

1. List two documents that could be used as the test basis for unit testing.
2. Describe test driven development (TDD).
3. Identify two typical test objects used for integration testing.
4. List three documents used as the test basis for system testing.
5. Compare a functional requirement with a non-functional requirement.
6. What is the purpose of acceptance testing?
7. List three documents used as a test basis for acceptance testing.
8. Identify three types of acceptance testing.

TEST TYPES

In the previous section we saw that each test level has specific testing objectives. In this section we will look at the types of testing required to meet these objectives.

Test types fall into the following categories:

- functional testing;
- non-functional testing;
- white-box testing;
- testing after code has been changed.

Functional testing

As you saw in the section on system testing, functional testing looks at the specific functionality of a system, such as searching for flights on a website, or perhaps calculating employee pay correctly using a payroll system. This can include checks for completeness, correctness and appropriateness.

Functional testing is carried out at all levels of testing, from unit through to acceptance testing. In the example above, the testing of the employee pay may be done during unit testing; whereas searching for a flight is often done during system testing.

Functional testing is also called specification-based testing or black-box testing (covered in **Chapter 4**). It can be measured in terms of the percentage of requirements covered by the tests.

Designing tests at this level often requires specific domain skills. In today's world, testing the blockchain used in crypto-currencies requires different knowledge and skills to those used in designing tests for normal banking operations, for instance.

Non-functional testing

This is where the behavioural aspects of the system are tested. As you saw in the section on system testing, examples include usability, performance, efficiency and security testing among others.

As for functional testing, non-functional testing:

- should be performed at all levels so that potential defects are detected as early as possible.
- can make use of black-box testing techniques, such as checking that a flight can be booked within a specific time frame;
- often requires specialist knowledge (such as knowing the inherent weaknesses of specific technologies) and skills (such as having an understanding of how to carry out performance testing);
- can be measured – for instance checking the percentage of mobile devices tested for compatibility with an application.

These tests can be referenced against a quality model, such as the one defined in ISO/IEC 25010 Systems and software Quality Requirements and Evaluation (SQuaRE). Note that a detailed understanding of this standard is not required for the exam.

White-box testing

In white-box testing our focus is on the internal structure of the system. This could be the code itself, an architectural definition or data flows through the system.

White-box testing is commonly carried out at unit and component integration test levels. Here, common measures include code and interface coverage (percentage of code and

interfaces exercised by tests). Further detail on code coverage measures is provided in **Chapter 4**.

It can also be carried out at the higher levels of testing where a structural definition of the system exists. An example is a business flow (represented as a flow chart), which could be used to design tests at system or higher levels.

As before, this type of testing also requires specialised knowledge and skills, such as code creation, data storage on databases and use of the associated tools.

Testing related to changes

The previous sections detail the testing to be carried out at the different stages in the development life cycle. At any level of testing, it can be expected that defects will be discovered. When these are found and fixed, the quality of the system being delivered is improved.

After a defect is detected and fixed, the changed software should be retested to confirm that the problem has been successfully removed. This is called retesting or confirmation testing. Note that when the developer removes the defect this activity is called debugging, which is not a testing activity. Testing finds a defect, debugging fixes it.

The unchanged software should also be retested to ensure that no additional defects have been introduced as a result of changes to the software. This is called regression testing. Regression testing should also be carried out if the environment has changed.

Regression testing involves the creation of a set of tests, which serve to demonstrate that the system works as expected. These are run many times over a testing project, when changes are made, as discussed above. This repetition of tests makes regression testing suitable for automation in many cases. Test automation is covered in detail in **Chapter 6**.

In iterative development projects such as Agile development, the requirements churn introduces a great need for both confirmation and regression testing. There is also a concept of code refactoring (where a developer seeks to increase the quality of the code written), which also necessitates change-related testing.

CHECK OF UNDERSTANDING

Which of the following is correct?

a. Regression testing checks that a problem has been successfully addressed, while confirmation testing is done at the end of each release.
b. Regression testing checks that all problems have been successfully addressed, while confirmation testing refers to testing individual fixes.
c. Regression testing checks that fixes to errors do not introduce unexpected functionality into the system, while confirmation testing checks that fixes have been successful.
d. Regression testing checks that all required testing has been carried out, while confirmation testing checks that each test is complete.

MAINTENANCE TESTING

For many projects (though not all) the system is eventually released into the live environment. Hopefully, once deployed, it will be in service as long as intended, perhaps for years or decades.

During this deployment, it may become necessary to change the system.

Triggers for maintenance include:

- additional features being required;
- the system being migrated to a new operating platform;
- the system being retired – data may need to be migrated or archived;
- planned upgrade to COTS-based systems;
- new faults being found requiring fixing (these can be 'hot fixes').

Once changes have been made to the system, they will need to be tested (retesting), and it also will be necessary to conduct regression testing to ensure that the rest of the system has not been adversely affected by the changes. Testing that takes place on a system that is in operation in the live environment is called maintenance testing.

When changes are made to migrate from one platform to another, the system should also be tested in its new environment. When migration includes data being transferred in from another application, then conversion testing also becomes necessary.

As we have suggested, all changes must be tested, and, ideally, all of the system should be subject to regression testing. In practice, this may not be feasible or cost-effective. An understanding of the parts of the system that could be affected by the changes could

reduce the amount of regression testing required. Working this out is termed impact analysis; that is, analysing the impact of the changes on the system.

Impact analysis for maintenance

The purpose of impact analysis is to determine the likely impact of a change to a system. We need to understand the intentions of the change, any potential side effects of the change, and how existing tests may need to be changed.

This can be difficult for a system that has already been released and is in maintenance. This is because the specifications may be out of date (or non-existent); test cases may have not been documented; there is a lack of traceability of tests backs to requirements; there is weak or non-existent tool support; or the original development team may have moved on to other projects or left the organisation altogether.

CHECK OF UNDERSTANDING

1. How do functional requirements differ from non-functional requirements?
2. For which type of testing is code coverage measured?
3. What is the purpose of maintenance testing?
4. Give examples of when maintenance testing is necessary.
5. What is meant by the term impact analysis?

SUMMARY

In this chapter we have explored the role of testing within the Software Development Life Cycle. We have looked at the basic steps in any development model, from understanding customer needs to delivery of the final product. These were built up into formally recognisable models, using distinct approaches to software development.

The V model, as we have seen, is a stepwise approach to software development, meaning that each stage in the model must be completed before the next stage can be started, if a strict implementation of the model is required. This is often the case in safety-critical developments. The V model typically has the following work products and activities:

1. requirement specification;
2. functional specification;
3. technical specification;
4. program specification;
5. code;
6. unit testing;
7. integration testing;

8. system testing;
9. acceptance testing.

Work products 1–5 are subject to verification, to ensure that they have been created following the rules set out. For example, the program specification is assessed to ensure that it meets the requirements set out in the technical specification, and that it contains sufficient detail for the code to be produced.

In activities 6–9, the code is assessed progressively for compliance to user needs, as captured in the specifications for each level.

An iterative model for development has fewer steps but involves the user from the start. These steps are typically:

1. define iteration requirement;
2. build iteration;
3. test iteration.

This sequence is repeated for each iteration until an acceptable product has been developed.

An explanation of each of the test levels in the V model was given. For unit testing the focus is the code within the unit itself, for integration testing it is the interfacing between units, for system testing it is the end-to-end functionality, and for acceptance testing it is the user perspective.

An explanation of test types was then given and by combining test types with test levels we can construct a test approach that matches a given system and a given set of test objectives very closely. The techniques associated with test types are covered in detail in **Chapter 4** and the creation of a test approach is covered in **Chapter 5**.

Finally, we looked at the testing required when a system has been released, but a change has become necessary – maintenance testing. We discussed the need for impact analysis in deciding how much regression testing to do after the changes have been implemented. This can pose an added challenge if the requirements associated with the system are missing or have been poorly defined.

In the next chapter, techniques for improving requirements will be discussed.

Example examination questions with answers

E1. K1 question
Which of the following are test levels where white-box testing is applicable?

i. Unit testing.
ii. Acceptance testing.
iii. Regression testing.
iv. Performance testing.

a. i and ii.
b. i only.
c. ii and iii.
d. ii and iv.

E2. K2 question
Which of the following is true of non-functional testing?

a. Examples of non-functional testing are provided in ISO Standard 20246.
b. It is best carried out at system and acceptance test levels.
c. It cannot usually be measured.
d. It can make use of black-box test techniques.

E3. K2 question
Which of the following iterative development models tends to work with shorter iterations relative to the others?

a. Waterfall model.
b. Rational Unified Process.
c. Scrum.
d. V model.

E4. K2 question
Which of the following statements are true?

i. For every development activity there is a corresponding testing activity.
ii. Each test level has the same test objectives.
iii. The analysis and design of tests for a given test level should begin after the corresponding development activity.
iv. Testers should be involved in reviewing documents as soon as drafts are available in the development life cycle.

a. i and ii.
b. iii and iv.
c. ii and iii.
d. i and iv.

E5. K2 question
Which of the following is not true of regression testing?

a. It can be carried out at each stage of the life cycle.
b. It serves to demonstrate that the changed software works as intended.
c. It serves to demonstrate that software has not been unintentionally changed.
d. It is often automated.

Answers to questions in the chapter

SA1. The correct answer is d.
SA2. The correct answer is b.
SA3. The correct answer is a.

Answers to example examination questions

E1. The correct answer is a.
White-box testing is applicable at all test levels. Regression and performance testing are not test levels; they are test types.

E2. The correct answer is d.
Option a provides a standard for use in reviews. The standard used for non-functional testing is ISO 25010. Option b is incorrect – non-functional testing should be carried out all levels. Option c is incorrect, it can be measured in terms of percentage of non-functional requirements covered.

E3. The correct answer is c.
Options a and d are sequential models and do not use iterative development. Option b – the Rational Unified Process – tends to use longer iterations than Scrum.

E4. The correct answer is d.
Option ii is incorrect – each test level has a different objective. Option iii is also incorrect– test analysis and design should start once the documentation has been completed.

E5. The correct answer is b.
This is a definition of confirmation testing. The other three options are true of regression testing.