includes allowing time in schedules for reviews to be done (and also any resulting rework!). Managers should also keep an eye on the effectiveness of reviews and encourage increasing effectiveness and efficiency, so that the greatest benefits are gained from reviews. The manager should have clear objectives for the review process(es) and should determine whether those objectives have been met. The manager will ensure that any review training requested by the participants takes place. Of course, a manager can also be involved in the review itself, depending on his or her background, playing the role of a review leader or reviewer if this would be helpful. In some review types, the manager should not be the moderator or facilitator of the review meeting (for example inspection).

Facilitator (often called moderator)

The responsibilities of the facilitator or moderator are:

- Ensuring the effective running of review meetings (when held).
- Mediating, if necessary, between the various points of view.
- Being the person upon whom the success of the review often depends.

The facilitator or moderator leads each individual review process. The facilitator performs the entry check and checks on the fixes, in order to control the quality of the input and output of the review process. The moderator also schedules the meeting, disseminates work products and other relevant materials, organizes the meeting, coaches other team members, paces the meeting, leads possible discussions and stores the data that is collected.

Review leader

The responsibilities of the review leader are:

- Taking overall responsibility for the review.
- Deciding who will be involved.

Depending on the size of the organization, the role of the review leader may be taken by a manager, or by the facilitator (moderator). If review leader is a separate role, they will be working closely with both of these roles.

In some organizations, there is no distinction made between the review leader and the facilitator in practice. The main difference is that the review leader is responsible for the review happening, and organizes the people involved, but may not be involved in the review meeting (if this is how issues are communicated). The facilitator's main role is in dealing with the people while the review is happening, ensuring that the review meetings are run well, and making sure that interpersonal issues do not disrupt the review process.

Reviewers

The responsibilities of the reviewers are:

- Being subject matter experts, persons working on the project, stakeholders with an interest in the work product, and/or individuals with specific technical or business backgrounds.
- Identifying potential defects in the work product under review.
- Representing different perspectives as requested, for example tester, developer, user, operator, business analyst, usability expert, etc.

The role of the reviewers (also called checkers or inspectors) is to be another set of eyes to look at the work products from a fresh point of view. The reviewers check any material for defects, mostly prior to the meeting. The level of thoroughness required depends on the type of review. The level of domain knowledge or technical expertise needed by the reviewers also depends on the type of review. Reviewers should be chosen to represent different perspectives in the review process. In addition to the work product under review, reviewers also receive other material, including source work products, standards, checklists, etc. In general, the fewer source and reference work products provided, the more domain expertise is neeeded regarding the content of the work product under review.

Scribe (or recorder)

The responsibilities of the scribe or recorder are:

- Collating potential defects found during the individual review activity.
- Recording new potential defects, open points and decisions from the review meeting (when held).

During the logging meeting, the scribe (or recorder) has to record each defect mentioned and any suggestions for process improvement. In practice, it is often the author who plays the role of scribe, ensuring that the log is readable and understandable. If authors record their own defects, or at least make their own notes in their own words, it helps them to understand the log better when they look through it afterwards to fix the defects found. However, having someone other than the author take the role of the scribe (for example the facilitator) can have significant advantages, since the author is freed up to think about the work product rather than being tied down with lots of writing.

The role of the scribe or recorder may be less relevant, or even irrelevant, if the defects, discussion points, decisions and other issues raised in the review are recorded electronically, although the scribe may be the person doing that recording during a meeting.

Although we have described the roles and responsibilities as though they are all separate and distinct, this does not mean that they are done by different people. One person may take on more than one role and perform the responsibilities for multiple roles. The determination of who does what role also depends on the type of review.

3.2.3 Types of review

The different review types have different objectives and can be used for different purposes, but the most common objective of all review types is to uncover defects in the work product being reviewed. The type of review should be chosen, based on a number of factors, including the needs of the project, available resources, product type and risks, business domain and company culture.

The main review types, their main characteristics and attributes are described below.

Informal review (for example buddy check, pairing, pair review)

An informal review is characterized by the following attributes:

- Main purpose/objective: detecting potential defects.
- Possible additional purposes: generating new ideas or solutions, quickly solving minor problems.

- Not based on a formal (documented) review process.
- May not involve a review meeting.
- May be performed by a colleague of the author (buddy check) or by more people.
- Results may be documented (but often are not).
- Varies in usefulness depending on the reviewer(s).
- Use of checklists is optional.
- Very commonly used in Agile development.

An informal review may simply be one person saying to a colleague, 'Could you have a quick look at what I've just done?' The colleague may spend less than an hour looking through and giving any comments back to the author, such as typos, something missing, or a 'But have you thought of this?' comment. With the right person reviewing, this buddy check can be very effective (and at little cost in time). Other forms of informal review include pair working, when one person works with another to produce a work product, the second person continually evaluating (that is, reviewing) what the first person is typing.

Walkthrough

A **walkthrough** is characterized by the following attributes:

Walkthrough (Structured walkthrough) A type of review in which an author leads members of the review through a work product and the members ask questions and make comments about possible issues.

- Main purposes: find defects, improve the software product, consider alternative implementations, evaluate conformance to standards and specifications.
- Possible additional purposes: exchanging ideas about techniques or style variations, training of participants, achieving consensus.
- Individual preparation before the review meeting is optional.
- Review meeting is typically led by the author of the work product.
- Use of a scribe is mandatory.
- Use of checklists is optional.
- May take the form of scenarios, dry runs or simulations.
- Potential defect logs and review reports may be produced.
- May vary in practice from quite informal to very formal.

Within a walkthrough, the author does most of the preparation. The participants, who are selected from different departments and backgrounds, are not generally required to do a detailed study of the work products in advance (but it is an option). Because of the way the meeting is structured, a large number of people can participate and this larger audience can bring a great number of diverse viewpoints regarding the contents of the work product being reviewed. If the audience represents a broad cross-section of skills and disciplines, it can give assurance that no major defects are missed in the walkthrough. A walkthrough is especially useful for higher-level work products, such as requirement specifications and architectural documents.

A walkthrough is often used to transfer knowledge and educate a wider audience about a particular work product. In some cases, the educational value of a walkthrough is more important than finding defects (although defects should be welcomed).

If a large number of people are present, a scribe (someone other than the author) may be used to record the discussion, the questions raised and any decisions taken at the meeting.

Technical review

Technical review A formal review type by a team of technically-qualified personnel that examines the suitability of a work product for its intended use and identifies discrepancies from specifications and standards.

A **technical review** is characterized by the following attributes:

- Main purposes: gaining consensus, detecting potential defects.
- Possible further purposes: evaluating quality and building confidence in the work product, generating new ideas, motivating and enabling authors to improve future work products, considering alternative implementations.
- Reviewers should be technical peers of the author, and technical experts in relevant disciplines.
- Individual preparation before the review meeting is required.
- Review meeting is optional, ideally led by a trained facilitator (typically not the author).
- Scribe is mandatory, ideally not the author.
- Use of checklists is optional.
- Potential defect logs and review reports are typically produced.

A technical review is often a discussion meeting that focuses on achieving consensus about the technical content of a work product that all the participants have studied before the meeting. During technical reviews, defects are found by experts, who focus on the content of the work product. The experts who participate in a technical review may include, for example, architects, chief designers and key users. It is useful to have an independent facilitator, especially if there are a number of strong opinions about technical issues. Technical reviews are typically less formal than inspections, but generally more formal than walkthroughs. In practice, technical reviews may vary from quite informal to very formal.

Inspection

Inspection A type of formal review to identify issues in a work product, which provides measurement to improve the review process and the software development process.

An **inspection** is characterized by the following attributes:

- Main purposes: detecting potential defects, evaluating quality and building confidence in the work product, preventing future similar defects through author learning and root cause analysis.
- Possible further purposes: motivating and enabling authors to improve future work products and the software development process, achieving consensus.
- A defined process is followed, with formal documented outputs, based on rules and checklists.
- There are clearly defined roles, such as those specified in Section 3.2.2 which are mandatory and may include a dedicated reader who reads/paraphrases the work product aloud during the review meeting.
- Individual preparation before the review meeting is required.

- Reviewers are either peers of the author or experts in other disciplines that are relevant to the work product.
- Specified entry and exit criteria are used.
- A scribe is mandatory.
- The review meeting is led by a trained facilitator/moderator (not the author).
- The author cannot act as the review leader, facilitator, reader or scribe.
- Potential defect logs and review reports are produced.
- Metrics are collected and used to improve the entire software development process, including the inspection process.

Inspection is the most formal review type. The work product under inspection is prepared and checked thoroughly by the reviewers before the meeting, comparing the work product with its sources and other referenced work products, and using rules and checklists. In the inspection meeting, the defects found are logged and any discussion is postponed until the discussion part of the meeting. This makes the inspection meeting a very efficient meeting.

Depending on the organization and the objectives of a project, inspections can be balanced to serve a number of goals. For example, if the time to market is extremely important, the emphasis in inspections will be on efficiency. In a safety-critical market, the focus will be on effectiveness.

When inspections are done well, they not only help to identify defects in the work products being reviewed, but the emphasis on process improvement and learning leads to better ways of producing work products, both for the author and for other reviewers.

All review types

A single work product may be the subject of more than one review. If more than one type of review is used, the order may vary. For example, an informal review may be carried out before a technical review to make sure the work product is ready for the technical review, or an inspection may be carried out on a requirements specification or epic before a walkthrough with customers. No one of the types of review is the winner, but the different types serve different purposes at different stages in the life cycle of a work product.

A peer review is where all of the reviewers are at the same or similar organizational level as the author, that is, the author's equals are reviewing the work. A peer review is not another type of review, as all of the types of review could be carried out by peers, from informal review to inspection.

All review types have as at least one of their purposes to find defects. Section 3.1.3 gave some examples of defects that can be found by static testing, including reviews. The most important thing is to find the most severe defects, those that represent the highest risk to the organization. Reviewing the most important work products is one way to get greater value from reviews. For example, a decimal point error may not be very important in a report about sales of low-volume items, but a decimal point error in a multi-million dollar or euro contract could be very significant. The types of defect found depend on the work product, as well as on the reviewers, and the way in which the reviews are carried out.

3.2.4 Applying review techniques

Suppose you have agreed to participate in a review of some type. You have been given the work product to review, with instructions about which parts to concentrate on, and perhaps some related documentation. How do you actually do the individual reviewing, the 'individual preparation' of the review process?

There are a number of techniques that can be used, whatever type of review is being performed. The main aim is to find defects, but some of these techniques will be more effective than others in some situations.

The Syllabus lists five different techniques (although you could argue that the first one is more the absence of any technique).

Ad hoc reviewing

Ad hoc reviewing A review technique carried out by independent reviewers informally, without a structured process.

If you have been given no instructions or guidance for how to review, then you will probably be doing an **ad hoc review**. You will likely read the work product from the beginning, and you may notice a few things that you note as possible defects. If you are a very good reviewer, this may be quite effective, but if you really are not sure what you are supposed to be looking out for, you may not find much that is useful. If everyone uses only ad hoc reviewing, the reviewers are likely to all find the same things, which is wasteful.

Checklist-based reviewing

Checklist-based reviewing A review technique guided by a list of questions or required attributes.

Checklist-based reviewing is often far more effective, because there is helpful guidance given about what to look for in the supplied checklist. The checklist would be one of the things distributed when the review is initiated. Different reviewers may have different checklists. This helps to cut down the number of duplicate defects found by reviewers. A checklist may contain items to check or questions.

The checklist may contain questions relating to the work product, such as 'Have references been given for claims shown?', 'Are all pointers valid?' or 'Has the customer's viewpoint been considered?' It is a good idea to organize a checklist so that if the answer to a question is 'No', then there is a potential defect. The best checklist questions are often derived from something that went wrong or was missed in a previous review. If a major defect was missed, adding a specific question to a checklist to look for that type of defect is very effective.

However, checklists should not just be allowed to grow and grow. It is important to review the checklists regularly, and to remove questions that are no longer useful, so that the checklist is focused on the most important things. The longer a checklist is, the less likely it is to be fully used, so it is also a good idea to limit the size of the checklist to one page, and to put the most important checklist questions at the beginning.

Checklists are useful when reviewing any type of work product, from unit or component code to user stories or Help text. The questions would, of course, be different for each work product, as they are targeted at potential defects for that type of work product. For example, code-specific questions (about pointers) would be appropriate for a unit code checklist, and customer-related questions would be appropriate for a user story or requirements specification checklist.

When checklists are used well, this is a very effective review technique; if all of the checklist questions are evaluated, this is a systematic search for previous or typical defect types.

One last word of warning about checklist-based reviews: do not be constrained to check only what is on the checklist. The checklist can help to guide you to look for specific things, but also be aware of other things, and take inspiration from the checklist to look for related potential defects outside of the checklist.

Scenario-based reviewing and dry runs

A **scenario-based review** is one where the reviewers are given a structured perspective on how to work through a work product from a particular point of view. For example, if use cases are used, a use case can be a scenario to work through how the system will work from each actor's point of view (people and other systems).

Scenario-based reviewing A review technique where the review is guided by determining the ability of the work product to address specific scenarios.

When you are stepping through the scenario, this is also called a 'dry run', a term used for a rehearsal before a big event or speech. In this sense, going through the scenario is a rehearsal for the system. We are looking at the way the system is expected to be used, which is an important aspect of validation.

A scenario is a different way of reviewing from using a checklist. You are looking at how the system will be used from a specific perspective, and this can be more focused than just a list of questions or points to check. When using a scenario, we are acting out realistic ways of using the system and validating whether or not it will meet user needs and expectations. A checklist is more likely to be focused on verifying individual aspects, such as types of defect, that have occurred previously.

As with checklists, do not be constrained by the scenario, but use it as inspiration for finding other defects outside of the scenario. With user-focused scenarios, be particularly aware of missing features.

Role-based reviewing

Role-based reviewing is similar to scenario-based reviewing, but the viewpoints are different stakeholders rather than just users. Roles can include specific end-user types, such as experienced versus inexperienced, age-related (children, teenagers, adults, seniors), or accessibility roles such as vision impaired, hearing impaired, etc.

Role-based reviewing A review technique where reviewers evaluate a work product from the perspective of different stakeholder roles.

Personas may also be used, which are typically based on a profile of a specific set of characteristics. A persona is the characterization of a user who represents the target audience for your system. A number of different personas are used to represent a range of user profiles, needs or desires. For example, one persona may be a young adult who is single and likes skiing; another may be a married elderly person with significant health problems. Each persona is described in detail (job, where they live, income level, etc.). Each persona would represent a role in role-based reviewing.

Figure 3.1 shows some different roles with respect to documents or work products used within a review. The roles represent views of the work product under review:

- Focus on higher-level work products, for example does the design comply to the requirements (Type 1 in Figure 3.1).
- Focus on standards, for example internal consistency, clarity, naming conventions, templates (Type 2).
- Focus on related work products at the same level, for example interfaces between software functions (Type 3).
- Focus on usage of the work product, for example for testability or maintainability (Type 4).

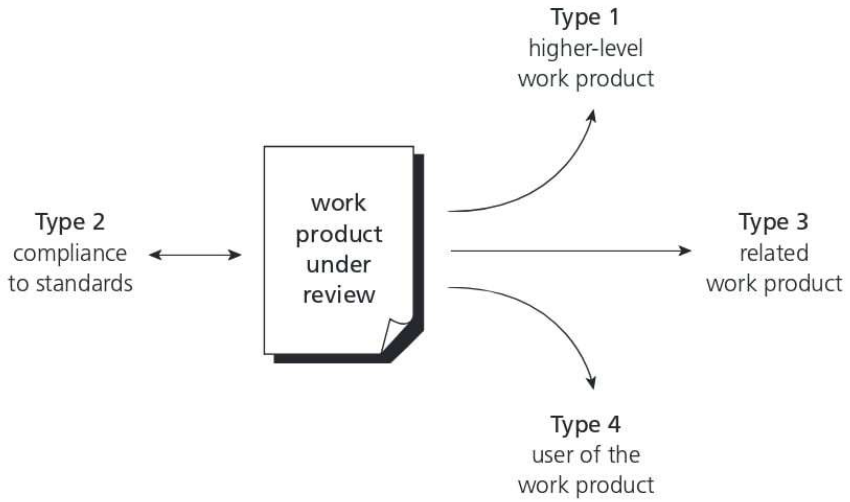


FIGURE 3.1 Basic review roles for a work product under review

The author may raise additional specific roles and questions to be addressed. The moderator has the option to also fulfil a role, alongside the task of being the facilitator. Checking the work product improves the facilitator's ability to lead the meeting, because it ensures better understanding. Furthermore, it improves the review efficiency because the facilitator replaces an engineer who would otherwise have to check the work product and attend the meeting. It is recommended that the facilitator take the role of checking compliance to standards, since this tends to be a highly objective role, with less discussion about the defects found.

Roles can also be organizational, such as from the perspective of a system administrator or user administrator. They can also be from testing perspectives such as performance testing or security testing.

In inspection, viewpoint roles can also include reviewing from the perspective of contractual issues, manufacturing (if relevant), legal issues, design or testing, operations or delivery or a third-party supplier. In addition to these viewpoint roles, there can be procedural roles, such as checking all financial calculations, starting from the back, looking for the most important things or cross-checking within or between work products. There may also be document or work product roles where each reviewer is given special responsibility for using one particular related work product. For example, if there are three user stories that are related to the work product being reviewed, and there are three reviewers, each would pay special attention to one of the three user stories.

Perspective-based reading

Perspective-based reading (Perspective-based reviewing) A review technique whereby reviewers evaluate the work product from different viewpoints.

Perspective-based reading is similar to role-based reviewing, but rather than playing a specific role, the reviewer typically tries to perform the tasks on a high level that they would be doing with the work product under review (taking that perspective), for example, as a tester making some test designs. Stakeholder perspectives include end-users, marketing, designer, tester or operations. These are similar to some of the inspection roles, and in fact this technique was devised for inspections. This technique goes beyond role-based and is therefore typically more expensive, but it also finds more defects.

The benefits of this technique (and also role-based and scenario-based reviewing) are that each individual reviewer has their own specialty within the review, and will be looking in more depth within their own assigned area. This makes the review more effective (since there is deeper checking) but also more efficient, since there is less duplication of issues found by different reviewers. Checklists can also be used for the different perspectives.

Another approach to perspective-based reading is for the reviewers to use the work product to generate other work products from it. When doing perspective-based reading on a requirements specification, a tester-perspective reviewer would be generating acceptance tests. This can help to identify missing information needed for the tests.

Perspective-based reading combines aspects of all of the review techniques (except ad hoc) and so can be the most effective. There is no one right way to do reviews, but when reviewers look at the work product in different ways, using checklists, scenarios, roles or perspectives, then the best results are gained. More information is available in Shull, Rus and Basili [2000].

3.2.5 Success factors for reviews

Implementing (formal) reviews is not easy. There is no one way to success and there are numerous ways to fail. The most common reasons for failure in reviews are due to either organizational factors or people-related factors. We will look at aspects of both of these.

One aspect (which is not emphasized in the Syllabus) is the importance of having a champion, the person who will lead the process on a project or organizational level. They need expertise, enthusiasm and a practical mindset in order to guide moderators and participants. The authority of this champion should be clear to the entire organization.

Organizational success factors for reviews

To be successful, a review culture should be part of the mindset of the organization, and there are a number of aspects which can make or break the effectiveness and efficiency of reviews within an organization.

Have clear objectives

Each review should have clear objectives. These are defined during the planning stage, communicated to all participants and may be used to measure the exit criteria or definition of done for the review.

Pick the right review type and technique

In the previous subsections, we discussed a number of review types and techniques, each of which has its strengths and weaknesses, and advantages and disadvantages in use. You should be careful to select and use review types and techniques that will best enable the achievement of the objectives of the project and the review itself. Be sure to consider the type, importance and risk level of the work product to be reviewed, and the reviewers who will participate. For example, do not try to review everything by inspection; fit the review to the risk associated with specific parts of the work product. Some work products may only warrant an informal review and others will repay using inspection. Of course, it is also of utmost importance that

the right people are involved. Consider the work product to be reviewed. Would a checklist-based review technique or a role-based technique be the most suitable for identifying defects?

Review materials need to be kept up to date

We use work products that support the review process to perform a review. These work products need to be up to date and of good quality in order to support the reviews. Review materials, such as checklists, need reviewing too! When significant defects are found in work products (either in reviews or in testing), add an item to the relevant checklist so that this type of work product defect can be identified earlier next time. At regular intervals, check to make sure that all of the items or questions on a checklist are still relevant, and remove any that are not useful any more.

Limit the scope of the review

Large documents or work products should be reviewed in small chunks, so that feedback can be given to the authors while they are still working on the work product. Early and frequent feedback is more effective and more efficient, partly because if a particular type of defect is detected early, the author can be aware of this in the rest of the work product; this prevents that type of defect from appearing in the parts of the work product written later.

It takes time

Reviews take time, and this time needs to be scheduled. But it is not just the time for review meetings. It is more important to allow adequate time for reviewers to do the preparation and individual studying of the work product before any meeting, as this is when most defects are identified. Skimping on preparation time is a false economy. The review meetings must be scheduled with adequate notice, so that reviewers can plan the review time into their own work schedules. It is not realistic to expect reviewers to be able to drop everything for a review, they need to balance their own work with the review work.

Management support is critical

Management support is essential for success. Managers should, among other things, incorporate adequate time for review activities in project schedules. They should visibly support the review process and help to foster a culture where reviews are seen as valuable both to the organization and to the individual. Managers especially must commit not to use metrics or other information from reviews for the evaluation of the author or the participants. If people are blamed for making mistakes, they do not make fewer mistakes, they hide them better!

One of the authors was called in to give training for reviews in an organization where the manager claimed to be very much in favour of using them. This sounded good, but during the training, it became clear that something was not right; there was great resistance to the idea of reviews. When raised with the manager, he said, 'But I am really supporting reviews here – that way I can find out who puts in the most defects and fire them.' Needless to say, reviews did not work in this organization!

To ensure that reviews become part of the day-to-day activities, the hours to be spent should be made visible within each project plan. The engineers involved should be prompted to schedule time for preparation and, very importantly, rework. Tracking these hours will improve planning of the next review. As stated earlier, management plays an important part in planning of review activities.

Report quantified and aggregated results and benefits at a high (not individual) level to all those involved as soon as possible, including discussion of the consequences of defects if they had not been found this early. Costs should of course be tracked, but benefits, especially when problems do not occur in the future, should be made visible by quantifying the benefits as well as the costs.

People-related success factors for reviews

Reviews are about evaluating someone's work product. Some reviewers tend to get too personal when they are not well managed by the facilitator (moderator). People issues and psychological aspects should be dealt with by the facilitator and should be part of the review training, thus making the review a positive experience for the author. During the review, defects should be welcomed. This is much more likely when they are expressed objectively. It is important that all participants create and operate in an atmosphere of trust.

Pick the right reviewers

The people chosen to participate in a review have a significant impact on the success and outcome of the review. Different review types or techniques may require different skills from the reviewers, and different types of work product may require different skills. For example, if code is being reviewed, the reviewers must at least be able to read and understand the code, so developer skills are needed. If role-based review techniques are used, people who have or can adopt those particular perspectives are needed as reviewers. If someone will be using a particular work product in their own work (for example a developer who will be working to implement a feature in a user story), that person would be a good candidate to include on the review team, as they have a vested interest in understanding their source (the user story), clarifying any ambiguities and removing any defects before they use it in their own development work.

Use testers

As discussed in Chapter 1, testers are professional pessimists. This focus on what could go wrong makes them good contributors to reviews, provided they observe the earlier points about keeping the review experience a positive one. In addition to providing valuable input to the review itself, testers who participate in reviews often learn about the product. This supports earlier testing, one of the principles discussed in Chapter 1. Using testers as reviewers is also beneficial to the testers, as the review can help them identify relevant test conditions and test cases, and to begin preparing the tests earlier.

Each participant does their review work well

Each reviewer has responsibilities and possibly roles to play in the review. It is important that each of them takes the reviewing work seriously and does it to the best of their ability. This includes spending adequate time on the review activities, and paying attention to detail as needed, for example by using a checklist.

Limit the scope of the review and pick things that really count

As mentioned under the organizational factors, do not try to review too much at one time. Even if a limited scope has not been formally defined, reviewers should be selective about what they spend their time on. Select the work products for review that are most important in a project. Reviewing highly critical, upstream work

products like requirements and architecture will most certainly show the benefits of the review process to the project. This investment in review hours will have a clear and high return on investment.

Another advantage of limiting the scope of a review is that it is easier to concentrate on a small piece of work rather than trying to take in a large amount. Reviewers need to keep up their concentration on the important things, both in individual preparation and in any review meeting.

Defects found should be welcomed

The attitude towards defects found in reviews is critical to success. Any defect found is an opportunity to improve not only the quality of the work product that it was found in but also to become aware of other similar defects in the same or in different work products. When reviews are working well, many more defects are prevented than are found.

But the attitude towards defects by the author of the work product is also critical. Defects should be acknowledged and appreciated (even if you are not sure at the time whether it really is a defect or not, acknowledge the point made by the person reporting it). Defects should be reported and handled objectively; defect reporting should never become personal, as this creates ill will and damages the review process and culture.

Review meetings are well managed

The role of the facilitator or moderator in the review meeting is one of the major factors in making the whole review experience useful and pleasant. The meeting should be focused on the review objectives, and any discussion should be tightly controlled. It is all too easy for discussions in meetings to go on and on, but if the purpose of the review is to identify defects as efficiently as possible, then most discussion is probably unnecessary. All review participants should feel that their time in the review (including the meeting) has been a valuable use of their time.

Trust is critical

As mentioned in the earlier section, it is very important that the defect information is considered sensitive data and handled accordingly. Even the rumour of a manager wanting to use review data to evaluate people is enough to destroy the effectiveness of a review process. There needs to be an atmosphere of trust among the reviewers; they all know that they are helping each other get better, and they need to be confident that the data will be used in the right way.

How you communicate is important

Particularly in review meetings, subliminal factors such as body language and tone of voice may damage the openness of a good review atmosphere. In addition, if defects raised are criticized, for example by the author, then reviewers are much less likely to raise other defects, so something very important may be missed. Language is important as well. Contrast ‘There may be a problem here’ (objective wording), with ‘You did it wrong here’ (personal attack).

Follow the rules but keep it simple

Follow all the formal rules until you know why and how to modify them, but make the process only as formal as the project culture or maturity level allows. Do not become too theoretical or too detailed. Checklists and roles are recommended to increase the effectiveness of defect identification.

Train participants

It is important that training is provided in review techniques, especially the more formal techniques, such as inspection. Otherwise the process is likely to be impeded by those who do not understand the process and the reasoning behind it. Special training should be provided to the facilitators (moderators) to prepare them for their critical role in the review process.

Continuously improve process and tools

Continuous improvement of process guidelines and supporting tools (for example checklists), based upon the ideas of participants, ensures the motivation of the engineers involved. Motivation is the key to a successful change process. There should also be an emphasis, in addition to defect finding, on learning and process improvement which becomes part of the culture of the organization.

Just do it!

Finally, the review process is simple but not easy. Every step of the process is clear, but experience is needed to execute them correctly. Try to get experienced people to observe and help where possible. But most importantly, start doing reviews and start learning from every review.

More information about successful reviews can be found in Weigers [2002], van Veenendaal [2004], Sauer [2000] and Kramer and Legeard [2016].

CHAPTER REVIEW

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

From Section 3.1, you should be able to recognize the software work products that can be examined by static testing techniques. You should be able to explain the value of static testing by using examples. You should be able to explain the difference between static testing and dynamic testing in terms of their objectives, types of defects to be found, and the role of these techniques within the software life cycle. You should know the Glossary terms **dynamic testing**, **static analysis** and **static testing**.

From Section 3.2, you should be able to summarize the activities of the work product review process. You should recognize the different roles and responsibilities in a formal review. You should be able to explain the differences between the various types of review: informal review, walkthrough, technical review and inspection. You should be able to apply a review technique to a work product to find defects. Finally, you should be able to explain the factors for successful performance of reviews, both organizational and people-related. You should know the Glossary terms **ad hoc reviewing**, **checklist-based reviewing**, **formal review**, **informal review**, **inspection**, **perspective-based reading**, **review**, **role-based reviewing**, **scenario-based reviewing**, **technical review** and **walkthrough**. Finally, you should be able to carry out a review, particularly the individual reviewing of a work product, as this is K3 level in the Syllabus.

SAMPLE EXAM QUESTIONS

Question 1 Which of the following artefacts can NOT be examined using review techniques?

a. Software code.

b. User story.

c. Test designs.

d. User's intentions.

Question 2 Which of the following are the main activities of the work product review process?

1. Planning.
2. Initiate review.
3. Select reviewers.
4. Individual review.
5. Review meeting.
6. Evaluating review findings against exit criteria.
7. Issue communication and analysis.
8. Fixing and reporting.

a. 1, 2, 4, 7, 8.

b. 2, 3, 4, 5, 8.

c. 1, 2, 3, 5, 7.

d. 1, 4, 5, 6, 7.

Question 3 Which statement about static and dynamic testing is True?

a. Static testing and dynamic testing have different objectives.

b. Static testing and dynamic testing find the same types of defect.

c. Static testing identifies defects through failures; dynamic testing finds defects directly.

d. Static testing can find some types of defect with less effort than dynamic testing.

Question 4 What statement about reviews is True?

a. Inspections are led by a trained moderator, but this is not necessary for technical reviews.

b. Technical reviews are led by a trained leader, inspections are not.

c. In a walkthrough, the author does not attend.

d. Participants for a walkthrough always need to be thoroughly trained.

Question 5 Which statement below is True?

a. Management ensures effective running of review meetings; the review leader decides who will be involved.

b. Management is responsible for review planning; the facilitator monitors ongoing cost effectiveness.

c. Management organizes when and where reviews will take place; the review leader assigns staff, budget and time.

d. Management decides on the execution of reviews; the facilitator is often the person on whom the success of the review depends.

Question 6 Match the following characteristics with the type of review.

1. Led by the author.
2. Undocumented.
3. Reviewers are technical peers of the author.
4. Led by a trained moderator or leader.
5. Uses entry and exit criteria.

INSP: Inspection

TR: Technical review

IR: Informal review

W: Walkthrough

a. INSP: 4, TR: 3, IR: 2 and 5, W: 1

b. INSP: 4 and 5, TR: 3, IR: 2, W: 1

c. INSP: 1 and 5, TR: 3, IR: 2, W: 4

d. INSP: 5, TR: 4, IR: 3, W: 1 and 2

Question 7 Which of the following statements about success factors in reviews are True?

1. Reviewers should try to review as much of the work product as they can.
2. The author acknowledges and appreciates defects found in their work.
3. Each review has clear objectives, which are communicated to the reviewers.
4. Testers are not normally involved in reviews, as their work focuses on test design.
5. Checklists should be standardized and used for all types of work product.

a. 2 and 3.

b. 1, 2 and 3.

c. 1, 2 and 4.

d. 2, 3 and 5.

Question 8 Which review technique is this: reviewing a requirements specification or user story from the point of view of an end-user of the system, using a checklist?

a. Checklist-based.

b. Role-based.

c. Perspective-based.

d. Scenario-based.

Question 9 Consider the following specification for review:

When you sign up, you must give your first and last name, postal address, phone number, email address and password. When you log in, you must give your last name, phone number and password. You are logged in until you select Log Out followed by answering 'Yes' to 'Are you sure?' When you are logged in, you can update your details, but you need to confirm the change by entering a secure code sent to your phone. You then need to log in again.

The following are potential defects in the specification:

1. Incorrect timeout for the secure code sent to the phone.
2. Buffer overflow for lengthy postal address.
3. Cannot change your phone number, as the code is sent to the old number.
4. Details are stored as an additional entry in the database for every change.

Which of the potential detects above would be most likely to be found by a review performed by developers?

a. 2 and 3.

b. 2, 3 and 4.

c. 1, 2 and 4.

d. 1 and 4.

EXERCISE

Perform an individual review of the functional specification shown in Document 3.1, using a checklist-based review. The checklist is as follows:

1. Do all requirements give sufficient detail needed for determining the expected results for tests?
2. Are all dependencies and restrictions specified?
3. Are alternatives specified for all options?

Make a list of potential defects in the specification, noting the severity level of each (High, Medium or Low). Note which checklist question has found the defect. Solution ideas are given in the next section.

DOCUMENT 3.1 Functional requirements specification

Functional requirements

This specification describes the required functionality of the booking system for a sports centre.

Browsing the facilities

Customers will be able to view all the available facilities using the following options to filter the selection:

- None – shows all facilities
- Outdoor/indoor
- Sport – selected from a pull-down list
- Date – using a standard calendar
- Time – selected from a pull-down list

Each facility shown will include colour-coded information on availability (not available, already booked, and available for booking) and the time slots.

Selecting a facility

Any facility available for booking can be selected by clicking on the required time period. A confirmation message will be displayed. The user may either cancel the message and return to the list of facilities or continue to the booking details page.

Booking details

If the customer is not already logged in, the customer login dialogue will be displayed.

Details of the facility together with any relevant regulations (such as age restrictions, footwear and no-show conditions, etc.) will be displayed. The customer name, member ID and mobile phone number fields will be pre-filled. The customer can either cancel or confirm the details. Confirming the details takes the customer to the payment details page; cancelling returns the customer to the list of facilities page.

Payment details

Payment can be made by credit card. The following details are required:

- Card type – Visa or Mastercard
- Name on card – as printed on card
- Card number – 16 digits, no spaces
- Expiry date – mm/yy
- CVC code – last 3 digits on the signature strip

Having entered these details, the system will seek authorization for the payment. When this is received a final confirmation message is displayed. The customer can choose to cancel or confirm. If confirmed, the booking is made and a confirmation email is sent to the customer's email address (as specified in their customer profile). If the payment authorization fails, an error message is displayed.

EXERCISE SOLUTION

The following are some potential defects in the functional requirement shown in Document 3.1. This is not an exhaustive list; you may have found others. We have noted the checklist question number used to discover each issue, and a severity level of High, Medium or Low.

TABLE 3.1 Potential defects in the functional requirements specification

Defect	Description	Checklist	Severity
1	Need to know the specific colours in the colour coding to be able to see if the test passes or fails.	1	M
2	Need to know what all the facilities are, in order to check if they are displayed correctly.	1	M
3	What facilities are in the 'Sport' pull-down list?	2	M
4	What are the rules for selection of options? Does Date have to be selected before Time, for example? Can you select only one option?	2	H
5	If login fails, what is supposed to happen: can you continue to book anyway?	3	H
6	If login is successful, what screen is displayed, already selected, or back to list?	3	L
7	Can filters be combined? For example can you select Indoor and Tennis?	2	L
8	Is it possible to choose more than one time slot at once, for example 3 consecutive half-hour times?	2	M
9	If credit card authorization fails, what happens after the error message is shown?	3	L
10	The customer can cancel after authorization has gone through? Has payment already been taken? If so, will it be refunded? Automatically?	3	H
11	When booking is cancelled after authorization, what screen is shown after the error message?	3	L
12	Missing from specification: Is cancellation possible after a booking has been made, either by customer or by the sports centre? Is a refund given? Automatically?	none	H

As mentioned above, these are some suggested potential defects. Note that we have phrased many of them as questions, which may be less threatening to the author of the requirements document. Note also that we have noted an additional high severity defect which was not specifically triggered by an item in the checklist.

CHAPTER FOUR

Test techniques

Chapter 3 covered static testing, looking at work products and code, but not running the code we are interested in. This chapter looks at dynamic testing, where the software we are interested in is run by executing tests on the running code.

4.1 CATEGORIES OF TEST TECHNIQUES

SYLLABUS LEARNING OBJECTIVES FOR 4.1 CATEGORIES OF TEST TECHNIQUES (K2)

FL-4.1.1 Explain the characteristics, commonalities and differences between black-box test techniques, white-box test techniques and experience-based test techniques (K2)

In this section we will look at the different types of test techniques, how they are used, how they differ and the factors to consider when choosing a test technique. The three types or categories are distinguished by their primary source: a description of what the system should do (for example requirements, user stories, etc.), the structure of the system or component, or a person's experience. All categories are useful and the three are complementary.

The purpose of a test technique is to identify test conditions, test cases and test data. Test conditions are identified during analysis, and then used to define test cases and test data during test design, which are then used in test implementation. For example, in risk-based testing strategies, we identify risk items (which are the test conditions) when performing an analysis of the risks to product quality. Those risk items, along with their corresponding levels of risk, are subsequently used to design the test cases and implement the test data. Risk-based testing will be discussed in Chapter 5.

In this section, look for the definitions of the Glossary terms **black-box test technique**, **coverage**, **experience-based test technique**, **test technique** and **white-box test technique**.

4.1.1 Choosing test techniques

In this section we will look at the factors that go into the decision about which techniques to use when.

Which technique is best? This is the wrong question! Each technique is good for certain things, and not as good for other things. For example, one of the benefits of white-box techniques is that they can find things in the code that are not supposed to

be there, such as Trojan Horses or other malicious code. However, if there are parts of the specification that are missing from the code, black-box techniques can find that. White-box techniques can only test what is there. If there are things missing both from the specification and from the code, then only experience-based techniques would find them. Each individual technique is aimed at particular types of defect as well. For example, state transition testing is unlikely to find boundary defects.

The choice of which **test technique** to use depends on a number of factors, which we discuss below.

Test technique (test case design technique, test specification technique, test design technique) A procedure used to derive and/or select test cases.

Some techniques are more applicable to certain situations and test levels; others are applicable to all test levels. The best testing uses a combination of test techniques.

This chapter covers the most popular and commonly used software test techniques. There are many others that fall outside the scope of the Syllabus that this book is based on. With so many testing techniques to choose from, how are testers to decide which ones to use?

The use of techniques can vary in formality, from very informal (with little or no documentation) to very formal, where information about test conditions and why particular techniques are used is recorded. The level of formality also depends on other factors as discussed below. For example, safety or regulatory industries require a higher level of formality. The maturity of the organization, the life cycle model being used, and the knowledge and skills of the testers also influence the level of formality.

Perhaps the single most important thing to understand is that the best test technique is no single test technique. Because each test technique is good at finding one specific class of defect, using just one technique will help ensure that many (perhaps most but not all) defects of that particular class are found. Unfortunately, it may also help to ensure that many defects of other classes are missed! Using a variety of techniques will therefore help ensure that a variety of defects are found, resulting in more effective testing.

So how can we choose the most appropriate test techniques to use? The decision will be based on a number of factors, both internal and external.

The factors that influence the decision about which technique to use are:

- *Type of component or system*. The type of component (for example embedded, graphical, financial, etc.) will influence the choice of techniques. For example, a financial application involving many calculations would benefit from boundary value analysis.
- *Component or system complexity*. More complex components or systems are likely to have more defects, and defects may be harder to find. Using a different technique to address aspects of that complexity will give better defect detection from the testing. For example, a simple field with numerical input would be a good candidate for equivalence partitioning and boundary value analysis, but a screen with many fields, with complex dependencies, calculations and validation rules depending on aspects that change over time, would benefit from additional techniques such as decision table testing, state transition testing and white-box test techniques.
- *Regulatory standards*. Some industries have regulatory standards or guidelines that govern the test techniques used. For example, the aircraft industry requires the use of equivalence partitioning, boundary value analysis and state transition testing for high integrity systems, together with statement, decision or modified condition decision coverage depending on the level of software integrity required.

- *Customer or contractual requirements.* Sometimes contracts specify particular test techniques to use (most commonly statement or branch coverage).
- *Risk levels and risk types.* The greater the risk level (for example safety-critical systems), the greater the need for more thorough, formal testing. Commercial risk may be influenced by quality issues (so more thorough testing would be appropriate) or by time to market issues (so exploratory testing would be a more appropriate choice). Risk type might, for example, tell us that a risk is related to usability, performance, security or functionality. Of course, the correct test technique needs to be chosen that is able to address the risk type that is being mitigated.
- *Test objectives.* If the test objective is simply to gain confidence that the software will cope with typical operational tasks, then use cases would be a sensible approach. If the objective is for very thorough testing, then more rigorous and detailed techniques (including white-box test techniques) should be chosen.
- *Available documentation.* Whether or not documentation (for example a work product such as a requirements specification) exists, and how up-to-date it is, will affect the choice of test techniques. The content and style of the work products will also influence the choice of techniques (for example, if decision tables or state graphs have been used, then the associated test techniques should be used).
- *Tester knowledge and skills.* How much testers know about the system and about test techniques will clearly influence their choice of techniques. Experience-based techniques are particularly based on tester knowledge and skills.
- *Available tools.* If tools are available for a particular technique, then that technique may be a good choice. Since test techniques are based on models, the models available (that is, developed and used during the specification, design and implementation of the system) will to some extent govern which test techniques can be used. For example, if the specification contains a state transition diagram, state transition testing would be a good technique to use.
- *Time and budget.* Ultimately how much time there is available will always affect the choice of test techniques. When more time is available, we can afford to select more techniques. When time is severely limited, we will be limited to those that we know have a good chance of helping us find just the most important defects.
- *Software development life cycle model.* A sequential life cycle model will lend itself to the use of more formal techniques, whereas an iterative life cycle model may be better suited to using an exploratory test approach.
- *Expected use of the software.* If the software is to be used in safety-critical situations, for example medical monitoring devices or car-driving technology, then the testing should be more thorough, and more techniques should be chosen.
- *Previous experience with using the test techniques on the component or system to be tested.* Testers tend to use techniques that they are familiar with and more skilled at, rather than less familiar techniques. With this familiarity, you can become very effective at finding similar defects to those that have occurred before. But be aware that you may get better results from using a technique that is less familiar, and when you do use it, you will increase your skill and familiarity with it.

- *The types of defects expected in the component or system.* Knowledge of the likely defects will be very helpful in choosing test techniques (since each technique is good at finding a particular type of defect). This knowledge could be gained through experience of testing a previous version of the system and previous levels of testing on the current version.

It is important to remember that intelligent, experienced testers see test techniques (and indeed test strategies, which we'll discuss in Chapter 5) as tools to be employed wherever needed and useful. You should use whatever techniques and strategies, in whatever combinations, make sense to ensure adequate coverage of the system under test, and achievement of the objectives of testing. Feel free to combine the test techniques discussed in this chapter with whatever inspiration you have, along with process-related, rule-based and data-driven techniques. Use your brain and do what makes sense.

4.1.2 Categories of test techniques and their characteristics

There are many different types of software test techniques, each with its own strengths and weaknesses. Each individual technique is good at finding particular types of defect and relatively poor at finding other types. For example, a technique that explores the upper and lower limits of a single input range is more likely to find boundary value defects than those associated with combinations of inputs. Similarly, testing performed at different stages in the software development life cycle will find different types of defects; component testing is more likely to find coding logic defects than system design defects or user experience problems.

Each test technique falls into one of a number of different categories. Broadly speaking there are two main categories, static and dynamic. Static test techniques, as discussed in Chapter 3, do not execute the code being examined and are generally used before any tests are executed on the software. They could be called non-execution techniques. Most static test techniques can be used to test any form of work product including source code, design documents and models, user stories, functional specifications and requirement specifications. Static analysis is a tool-supported type of static testing that concentrates on testing formal languages and so is most often used to statically test source code.

In this chapter we look at dynamic test techniques, which are subdivided into three more categories: black-box (also known as specification-based, behavioural or behaviour-based techniques), white-box (structure-based or structural techniques) and experience-based. Black-box test techniques include both functional and non-functional techniques (that is, testing of quality characteristics). The techniques covered in the Syllabus are summarized in Figure 4.1.

Black-box test technique (black-box technique, specification-based technique, specification-based test technique) A procedure to derive and/or select test cases based on an analysis of the specification, either functional or non-functional, of a component or system, without reference to its internal structure.

Black-box test techniques

The first of the dynamic test techniques we will look at are the **black-box test techniques**. They are called black-box because they view the software as a black box with inputs and outputs, but they have no knowledge of how the system or component is structured inside the box. In essence, the tester is concentrating on what the software does, not how it does it.

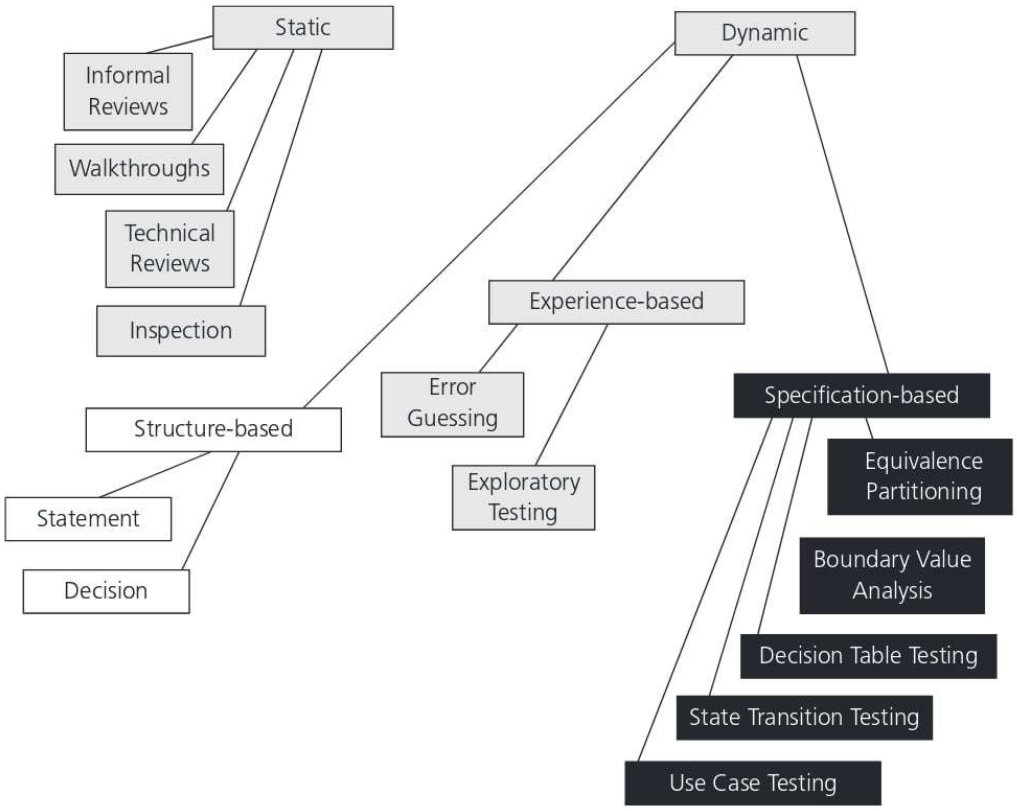


FIGURE 4.1 Test techniques

All black-box test techniques have the common characteristic that they are based on a model (formal or informal) of some aspect of the system, which enables test conditions and test cases to be derived from them in a systematic way.

Common characteristics of black-box test techniques include the following:

- Test conditions, test cases and test data are derived from a test basis that may include software requirements, specifications, use cases and user stories. The source of information for black-box tests is some description of what the system or software is supposed to do.
- Test cases may be used to detect gaps between the requirements and the implementation of the requirements, as well as deviations from the requirements. One of the strengths of test cases is that they make things specific, and this often highlights different understandings about the test basis, showing what is missing or interpreted differently.
- Coverage is measured based on the items tested in the test basis and the technique applied to the test basis. As we will see later, whenever you can make a list of some things that could be tested and can tell whether or not they have been tested, then you can measure coverage. Coverage at black-box level is based on items from the test basis. For example, does every requirement described have at least one test that exercises it?

Black-box test techniques are appropriate at all levels of testing (component testing through to acceptance testing) where a specification or other test basis exists. When

performing system or acceptance testing, the requirements specification or functional specification may form the basis of the tests. When performing component or integration testing, a design document or low-level specification may form the basis of the tests.

Notice that the definition mentions both functional and non-functional testing. Functional testing is concerned with what the system does, its features or functions. Non-functional testing is concerned with examining how well the system does something, rather than what it does. Non-functional aspects (also known as quality characteristics or quality attributes) include performance, usability, portability, maintainability, etc. Techniques to test these non-functional aspects are less procedural and less formalized than those of other categories, as the actual tests are more dependent on the type of system, what it does and the resources available for the tests.

Non-functional testing is part of the Syllabus and is also covered in Chapter 2. There are techniques for deriving non-functional tests [Gilb 1988], but they are not covered at the Foundation level.

Categorizing test techniques into black-box and white-box is mentioned in a number of testing books, including Beizer [1990], Black [2007] and Copeland [2004].

White-box test techniques

White-box test techniques (which are also dynamic rather than static) use the internal structure of the software to derive test cases. They are commonly called white-box or glass-box techniques (implying you can see into the system/box) since they require knowledge of how the software is implemented, that is, how it works. For example, a structural technique may be concerned with exercising loops in the software. Different test cases may be derived to exercise the loop once, twice and many times. This may be done regardless of the functionality of the software. All structure-based techniques have the common characteristic that they are based on how the software under test is constructed or designed. This structural information is used to assess which parts of the software have been exercised by a set of tests (often derived by other techniques). Additional test cases can then be derived in a systematic way to cover the parts of the structure that have not been touched by any test before.

White-box test technique (structural test technique, structure-based test technique, structure-based technique, white-box technique) A procedure to derive and/or select test cases based on an analysis of the internal structure of a component or system.

Common characteristics of white-box test techniques include the following:

- Test conditions, test cases and test data are derived from a test basis that may include code, software architecture, detailed design or any other source of information regarding the structure of the software. White-box test techniques are most commonly used for code structure, but these techniques are also useful for other structures. For example, the menu structure of an app could be tested using white-box techniques.
- **Coverage** is measured based on the items tested within a selected structure, for example the code statements, the decisions, the interfaces, the menu structure or any other identified structural element. See Section 4.3 for more on coverage.
- White-box test techniques determine the path through the software that either was taken or that you want to be taken, and this is determined by specific inputs to the software. However, in order to be a test, we also need to know what the expected outcome of the test case should be, even though this does not affect the path taken. A test oracle of some kind, for example a specification, is used to determine the expected outcome.

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

White-box test techniques can also be used at all levels of testing. Developers use white-box techniques in component testing and component integration testing,

especially where there is good tool support for code coverage. White-box techniques are also used in system and acceptance testing, but the structures are different. For example, the coverage of major business transactions could be the structural element in system or acceptance testing.

Experience-based test techniques

Experience-based test technique (experience-based technique) A procedure to derive and/or select test cases based on the tester's experience, knowledge and intuition.

In **experience-based test techniques**, people's knowledge, skills and background are a prime contributor to the test conditions, test cases and test data. The experience of both technical and business people is important, as they bring different perspectives to the test analysis and design process. Due to previous experience with similar systems, they may have insights into what could go wrong, which is very useful for testing. Both structural and behavioural insights are used to design experience-based tests.

All experience-based test techniques have the common characteristic that they are based on human knowledge and experience, both of the system itself (including the knowledge of users and stakeholders) and of likely defects. Test cases are therefore derived in a less systematic way, but may be more effective.

Experience-based test techniques are used to complement black-box and white-box techniques, and are also used when there is no specification, or if the specification is inadequate or out-of-date. This may be the only type of technique used for low-risk systems, but this approach may be particularly useful under extreme time pressure. In fact this is one of the factors leading to exploratory testing.

4.2 BLACK-BOX TEST TECHNIQUES

SYLLABUS LEARNING OBJECTIVES FOR 4.2 BLACK-BOX TEST TECHNIQUES (K3)

FL-4.2.1 **Apply equivalence partitioning to derive test cases from given requirements (K3)**

FL-4.2.2 **Apply boundary value analysis to derive test cases from given requirements (K3)**

FL-4.2.3 **Apply decision table testing to derive test cases from given requirements (K3)**

FL-4.2.4 **Apply state transition testing to derive test cases from given requirements (K3)**

FL-4.2.5 **Explain how to derive test cases from a use case (K2)**

In this section we will look in detail at four black-box test techniques. These four techniques are K3 in the Syllabus. This means that you need to be able to use these techniques to design test cases. We will also cover briefly (not at K3 level) the technique of use case testing. In Section 4.3, we will look at the K2 white-box test techniques.

In this section, look for the definitions of the Glossary terms **boundary value analysis**, **decision table testing**, **equivalence partitioning**, **state transition testing** and **use case testing**.

The four black-box test techniques we will cover in detail are:

- Equivalence partitioning.
- Boundary value analysis.
- Decision table testing.
- State transition testing.

4.2.1 Equivalence partitioning

Equivalence partitioning (EP) is a good all round black-box test technique. It can be applied at any level of testing and is often a good technique to use first. It is a common sense approach to testing, so much so that most testers practice it informally even though they may not realize it. However, while it is better to use the technique informally than not at all, it is much better to use the technique in a formal way to attain the full benefits that it can deliver. This technique will be found in most testing books, including Myers [2011], Copeland [2004], Jorgensen [2014] and Kaner *et al.* [2013].

Equivalence partitioning (partition testing) A black-box test technique in which test cases are designed to exercise equivalence partitions by using one representative member of each partition.

The idea behind the technique is to divide (that is, to partition) a set of test conditions into groups or sets where all elements of the set can be considered the same, so the system should handle them equivalently, hence 'equivalence partitioning'. **Equivalence partitions** are also known as equivalence classes: the two terms mean exactly the same thing.

Equivalence partition (equivalence class) A portion of the value domain of a data element related to the test object for which all values are expected to be treated the same based on the specification.

The EP technique then requires that we need test only one condition from each partition. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software. If one condition in a partition works, we assume all of the conditions in that partition will work, and so there is little point in testing any of these others. Conversely, if one of the conditions in a partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition. Of course these are simplifying assumptions that may not always be right, but if we write them down, at least it gives other people the chance to challenge the assumptions we have made and hopefully help to identify better partitions. If you have time, you may want to try more than one value from a partition, especially if you want to confirm a selection of typical user inputs.

For example, a savings account in a bank earns a different rate of interest depending on the balance in the account. In order to test the software that calculates the interest due, we can identify the ranges of balance values that earn the different rates of interest. For example, if a balance in the range $0 up to $100 has a 3% interest rate, a balance over $100 and up to $1,000 has a 5% interest rate, and balances of $1,000 and over have a 7% interest rate, we would initially identify three valid equivalence partitions and one invalid partition as shown below.

Invalid partition	Valid (for 3% interest)		Valid (for 5%)		Valid (for 7%)
–$0.01	$0.00	$100.00	$100.01	$999.99	$1,000.00

Notice that we have identified four partitions here, even though the specification only mentions three. This illustrates a very important task of the tester: not only do we test what is in our specification, but we also think about things that have not been specified. In this case we have thought of the situation where the balance is less than zero. We have not (yet) identified an invalid partition on the right, but this would also be a good thing to consider. In order to identify where the 7% partition ends, we would need to know what the maximum balance is for this account (which may not be easy to find out). In our example we have left this open for the time being. Note that non-numeric input is also an invalid partition (for example the letter 'a') but we discuss only the numeric partitions for now.

We have made an assumption here about what the smallest difference is between two values. We have assumed two decimal places, that is $100.00, but we could have assumed zero decimal places (that is $100) or more than two decimal places (for example $100.0000). In any case it is a good idea to state your assumptions: then other people can see them and let you know if they are correct or not.

We have also made an assumption about exactly which amount starts the new interest rate: a cent over to go into the 5% interest rate, but exactly on the $1,000.00 to go into the 7% rate. By making our assumptions explicit, and documenting them in this technique, we may highlight any differences in understanding exactly what the specification means.

When designing the test cases for this software we would ensure that the three valid equivalence partitions are each covered once, and we would also test the invalid partition at least once. So for example, we might choose to calculate the interest on balances of –$10.00, $50.00, $260.00 and $1,348.00. If we had not specifically identified these partitions, it is possible that at least one of them could have been missed at the expense of testing another one several times over. Note that we could also apply EP to outputs as well. In this case we have three interest rates: 3%, 5% and 7%, plus the error message for the invalid partition (or partitions). In this example, the output partitions line up exactly with the input partitions.

How would someone test this without thinking about the partitions? A naïve tester (let's call him Robbie) might have thought that a good set of tests would be to test every $50. That would give the following tests: $50.00, $100.00, $150.00, $200.00, $250.00, . . . say up to $800.00 (then Robbie would have got tired of it and thought that enough tests had been carried out). But look at what Robbie has tested: only two out of four partitions! So if the system does not correctly handle a negative balance or a balance of $1,000 or more, he would not have found these defects, so the naïve approach is less effective than EP. At the same time, Robbie has four times more tests (16 tests versus our four tests using equivalence partitions), so he is also much less efficient! This is why we say that using techniques such as this makes testing both more effective and more efficient.

Note that when we say a partition is invalid, it does not mean that it represents a value that cannot be entered by a user or a value that the user is not supposed to enter. It just means that it is not one of the expected inputs for this particular field. The software should correctly handle values from the invalid partition, by replying with an error message such as 'Balance must be at least $0.00'.

Note also that the invalid partition may be invalid only in the context of crediting interest payments. An account that is overdrawn will require some different action.

Here is a summary of EP characteristics:

- Valid values should be accepted by the component or system. An equivalence partition containing valid values is called a valid equivalence partition.

- Invalid values should be rejected by the component or system. An equivalence partition containing invalid values is called an invalid equivalence partition.
- Partitions can be identified for any data element related to the test object, including inputs, outputs, internal values, time-related values (for example before or after an event) and for interface parameters (for example integrated components being tested during integration testing).
- Any partition may be divided into sub-partitions if required, where smaller differences of behaviour are defined or possible. For example, if a valid input range goes from –100 to 100, then we could have three sub-partitions: valid and negative, valid and zero, and valid and positive.
- Each value belongs to one and only one equivalence partition from a set of partitions. However, it is possible to apply EP more than once and end up with different sets of partitions, as we will see later under 'Applying more than once' in the section on 'Extending equivalence partitioning and boundary value analysis').
- When values from valid partitions are used in test cases, they can be combined with other valid values in the same test, as the whole set should pass. We can therefore test many valid values at the same time.
- When values from invalid partitions are used in test cases, they should be tested individually, that is, not combined with other invalid equivalence partitions, to ensure that failures are not masked. Failures can be masked when several failures occur at the same time but only one is visible, causing the other failures to be undetected.
- EP is applicable at all test levels.

We have more things to say about EP (including how to measure coverage), but we will return to that after we cover boundary value analysis, since the two techniques are closely related.

4.2.2 Boundary value analysis

Boundary value analysis (BVA) is based on testing at the boundaries between partitions that are ordered, such as a field with numerical input or an alphabetical list of values in a menu. It is essentially an enhancement or extension of EP and can also be used to extend other black-box (and white-box) test techniques. If you have ever done 'range checking', you were probably using the BVA technique, even if you were not aware of it. Note that we have both valid boundaries (in the valid partitions) and invalid boundaries (in the invalid partitions).

Boundary value analysis A black-box test technique in which test cases are designed based on boundary values.

As an example, consider a printer that has an input option of the number of copies to be made, from 1 to 99.

Invalid	Valid		Invalid
0	1	99	100

To apply BVA, we will take the minimum and maximum (boundary) values from the valid partition (1 and 99 in this case) together with the first or last value respectively in each of the invalid partitions adjacent to the valid partition (0 and 100 in this case). In this example we would have three EP tests (one from each of the three partitions) and four boundary value tests.

Let's return to the savings account system described in the previous section:

Invalid partition	Valid (for 3% interest)		Valid (for 5%)		Valid (for 7%)
–$0.01	$0.00	$100.00	$100.01	$999.99	$1,000.00

Because the boundary values are defined as those values on the edge of a partition, we have identified the following boundary values: –$0.01 (an invalid boundary value because it is at the edge of an invalid partition), $0.00, $100.00, $100.01, $999.99 and $1,000.00, all valid boundary values.

So by applying BVA we will have six tests for boundary values. Compare what our naïve tester Robbie had done: he did actually hit one of the boundary values ($100) though it was more by accident than design. So in addition to testing only half of the partitions, Robbie has only tested one-sixth of the boundaries (so he will be less effective at finding any boundary defects). If we consider all of our tests for both EP and BVA, the techniques give us a total of ten tests, compared to the 16 that Robbie had, so we are still considerably more efficient as well as being over three times more effective (testing four partitions and six boundaries, so ten conditions in total compared to three).

Note that in this savings account interest example, we have valid partitions next to other valid partitions. If we were to consider an invalid boundary for the 3% interest rate, we have –$0.01, but what about the value just above $100.00? The value of $100.01 is not an *invalid* boundary; it is actually a *valid* boundary because it falls into a valid partition. So the partition for 5%, for example, has no invalid boundary values associated with partitions next to it.

A good way to represent the valid and invalid partitions and boundaries is in a table such as Table 4.1:

TABLE 4.1 Equivalence partitions and boundaries

Test conditions	Valid partitions	Invalid partitions	Valid boundaries	Invalid boundaries
Balance in account	$0.00 – $100.00 $100.01 – $999.99 $1,000.00 – $Max	< $0.00 > $Max non-integer (if balance is an input field)	$0.00 $100.00 $100.01 $999.99 $1,000.00 $Max	– $0.01 $Max + 0.01
Interest rates	3% 5% 7%	Any other value Non-integer No interest calculated	Not applicable	Not applicable

By showing the values in the table, we can see that no maximum has been specified for the 7% interest rate. We would now want to know what the maximum value is for an account balance, so that we can test that boundary. This is called an open boundary, because one of the sides of the partition is left open, that is, not defined. But that doesn't mean we can ignore it. We should still try to test it, but how? We have called it $Max to remind ourselves to investigate this.

Open boundaries are more difficult to test, but there are ways to approach them. Actually the best solution to the problem is to find out what the boundary should be specified as! One approach is to go back to the specification to see if a maximum has been stated somewhere else for a balance amount. If so, then we know what our boundary value is. Another approach might be to investigate other related areas of the system. For example, the field that holds the account balance figure may be only six figures plus two decimal figures. This would give a maximum account balance of $999,999.99 so we could use that as our maximum boundary value. If we really cannot find anything about what this boundary should be, then we probably need to use an intuitive or experience-based approach to probe various large values trying to make it fail.

We could also try to find out about the lower open boundary. What is the lowest negative balance? Although we have omitted this from our example, setting it out in the table shows that we have omitted it, so helps us be more thorough if we wanted to be.

Representing the partitions and boundaries in a table such as this also makes it easier to see whether or not you have tested each one (if that is your objective). To achieve 100% coverage for EP, we need to ensure that there is at least one test for each identified equivalence partition. To achieve 100% coverage of boundary values, we need to ensure that there is at least one test for each boundary value identified. Of course, just one test for either a partition or a boundary may not be sufficient testing, but it does show some degree of thoroughness, since we have not left any partition or boundary untested.

BVA can be applied at all test levels.

Extending equivalence partitioning and boundary value analysis

So far, by using EP and BVA we have identified conditions that could be tested, that is, partitions and boundary values. The techniques are used to identify test conditions, which could be at a fairly high level (for example low interest account) or at a detailed level (for example value of $100.00). We have been looking at applying these techniques to ranges of numbers. However, we can also apply the techniques to other things.

Applying to more than numbers

For example, if you are booking a flight, you have a choice of Economy/Coach, Premium Economy, Business or First Class tickets. Each of these is an equivalence partition in its own right and should be tested, but it does not make sense to talk about boundaries for this type of partition, which is a collection of valid things. The invalid partition would be an attempt to type in any other type of flight class (for example Staff). If this field is implemented using a drop-down list, then it should not be possible to type anything else in, but it is still a good test to try at least once in some drop-down field. When you are analyzing the test basis (for example a requirements specification or user story), EP can help to identify where a drop-down list would be appropriate.

When trying to identify a defect, you might try several values in a partition. If this results in different behaviour where you expected it to be the same, then there may be two (or more) partitions where you initially thought there was only one.

We can apply EP and BVA to all levels of testing. The examples here were at a fairly detailed level, probably most appropriate in component testing or in the detailed testing of a single screen.

At a system level, for example, we may have three basic configurations which our users can choose from when setting up their systems, with a number of options for each configuration. The basic configurations could be system administrator, manager and customer liaison. These represent three equivalence partitions that could be tested. We could have serious problems if we forget to test the configuration for the system administrator, for example.

Applying more than once

We can also apply EP and BVA more than once to the same specification item. For example, if an internal telephone system for a company with 200 telephones has three-digit extension numbers from 100 to 699, we can identify the following partitions and boundaries:

- Digits (characters 0 to 9) with the invalid partition containing non-digits.
- Number of digits, 3 (so invalid boundary values of two digits and four digits).
- Range of extension numbers, 100 to 699 (so invalid boundary values of 099 and 700).
- Extensions that are in use and those that are not (two valid partitions, no boundaries).
- The lowest and highest extension numbers that are in use could also be used as boundary values.

One test case could test more than one of these partitions/boundaries. For example, Extension 409 which is in use would test four valid partitions: digits, the number of digits, the valid range and the in use partition. It also tests the boundary values for digits 0 and 9.

How many test cases would we need to test all of these partitions and boundaries, both valid and invalid? We would need a non-digit, a two-digit and a four-digit number, the values of 99, 100, 699 and 700, one extension that is not in use, and possibly the lowest and highest extensions in use. This is 10 or 11 test cases: the exact number would depend on what we could combine in one test case.

Using EP and BVA helps us to identify tests that are most likely to find defects, and to use fewer test cases to find them. This is because the contents of a partition are representative of all of the possible values. Rather than test all ten individual digits, we test one in the middle (for example 4) and the two edges (0 and 9). Instead of testing every possible non-digit character, one can represent all of them.

Applying to output

As we mentioned earlier, we can also apply these techniques to output partitions. Consider the following extension to our bank interest rate example. Suppose that a customer with more than one account can have an extra 1% interest on this account if they have at least $1,000 in it. Now we have two possible output values (7% interest and 8% interest) for the same account balance, so we have identified another test condition (8% interest rate). (We may also have identified that same output condition by looking at customers with more than one account, which is a partition of types of customer.)

Applying to more than human inputs

EP can be applied to different types of input as well. Our examples have concentrated on inputs that would be typed in (by a human) when using the system. However, systems receive input data from other sources as well, such as from other systems via some interface. This is also a good place to look for partitions (and boundaries). For example, the value of an interface parameter may fall into valid and invalid equivalence partitions. This type of defect is often difficult to find in testing once the interfaces have been joined together, so is particularly useful to apply in integration testing, either component integration, for example between APIs (Application Programming Interfaces) or system integration. If you are receiving data from a third-party supplier, a value sent by the supplier may be larger than the maximum value your system is expecting. If you do not check the value when it arrives, this could cause problems.

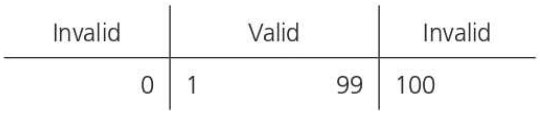
Boundary value analysis can be applied to a whole string of characters (for example a name or address). The number of characters in the string is a partition, for example between 1 and 30 characters is the valid partition with valid boundaries of 1 and 30. The invalid boundaries would be zero characters (null, just hit the Return key) and 31 characters. Both of these should produce an error message.

Partitions can also be identified when setting up test data. If there are different types of record, your testing will be more representative if you include a data record of each type. The size of a record is also a partition with boundaries, so we could include maximum and minimum size records in the test database.

If you have some inside knowledge about how the data is physically organized, you may be able to identify some hidden boundaries. For example, if an overflow storage block is used when more than 255 characters are entered into a field, the boundary value tests would include 255 and 256 characters in that field. This may be verging on white-box testing, since we have some knowledge of how the data is structured, but it does not matter how we classify things as long as our testing is effective at finding defects. Don't get hung up on a fine distinction: just do whatever testing makes sense, based on what you know. An old Chinese proverb says, 'It doesn't matter whether the cat is white or black; all that matters is that the cat catches mice'.

Two- and three-value boundary analysis

With BVA, we think of the boundary as a dividing line between two things. Hence we have a value on each side of the boundary, but the boundary itself is not a value.



Looking at the values for our printer example, 0 is in an invalid partition, 1 and 99 are in the valid partition and 100 is in the other invalid partition. So the boundary is between the values of 0 and 1, and between the values of 99 and 100. There is a school of thought that regards an actual value as a boundary value. By tradition, these are the values in the valid partition (that is, the values specified). This approach then requires three values for every boundary, so you would have 0, 1 and 2 for the left boundary, and 98, 99 and 100 for the right boundary in this example. The boundary values are said to be on and either side of the boundary and the value that is 'on' the boundary is generally taken to be in the valid partition.

Note that Beizer [1990] and Kaner, Padmanabhan and Hoffman [2013] talk about domain testing, a generalization of EP, with three-value boundaries. Beizer makes a distinction between open and closed boundaries, where a closed boundary is one where the point is included in the domain. So the convention is for the valid partition to have closed boundaries. You may be pleased to know that you do not have to know this for the exam! Three-value boundary testing is covered in ISO/IEC/IEEE 29119-4 [2015].

So which approach is best? If you use the two-value approach together with EP, you are equally effective and slightly more efficient than the three-value approach. (We will not go into the details here, but this can be demonstrated.) In this book we will use the two-value approach. In the exam, you may have a question based on either the two-value or the three-value approach, but it should be clear what the correct choice is in either case.

Designing test cases using EP and BVA

Having identified the test conditions that you wish to test, in this instance by using EP and BVA, the next step is to design the test cases. The more test conditions that can be covered in a single test case, the fewer test cases will be needed in order to cover all the conditions. This is usually the best approach to take for positive tests and for tests that you are reasonably confident will pass. However if a test fails, then we need to find out why it failed. Which test condition was handled incorrectly? We need to get a good balance between covering too many and too few test conditions in our tests.

Let's look at how one test case can cover one or more test conditions. Using the bank balance example, our first test could be of a new customer with a balance of $500. This would cover a balance in the partition from $100.01 to $999.99 and an output partition of a 5% interest rate. We would also be covering other partitions that we have not discussed yet, for example a valid customer, a new customer, a customer with only one account, etc. All of the partitions covered in this test are valid partitions.

When we come to test invalid partitions, the safest option is probably to try to cover only one invalid test condition per test case. This is because programs may stop processing input as soon as they encounter the first problem. So if you have an invalid customer name, invalid address and invalid balance, you may get an error message saying 'Invalid input' and you do not know whether the test has detected only one invalid input or all of them. This is also why specific error messages are much better than general ones!

However, if it is known that the software under test is required to process all input regardless of its validity, then it is sensible to continue as before and design test cases that cover as many invalid conditions in one go as possible. For example, if every invalid field in a form has some red text above or below the field saying that this field is invalid and why, then you know that each field has been checked, so you have tested all of the error processing in one test case. In either case, there should be separate test cases covering valid and invalid conditions.

To cover the boundary test cases, it may be possible to combine all of the minimum valid boundaries for a group of fields into one test case and also the maximum valid boundary values. The invalid boundaries could be tested together if the validation is done on every field; otherwise they should be tested separately, as with the invalid partitions.

Why do both equivalence partitioning and boundary value analysis?

Technically, because every boundary is in some partition, if you did only BVA you would also have tested every equivalence partition. However, this approach may cause problems if that value fails – was it only the boundary value that failed or did the whole partition fail? Also by testing only boundaries we would probably not give the users much confidence as we are using extreme values rather than normal values. The boundaries may be more difficult (and therefore more costly) to set up as well.

For example, in the printer copies example described earlier we identified the following boundary values:

Invalid	Valid		Invalid
0	1	99	100

Suppose we test only the valid boundary values 1 and 99 and nothing in between. If both tests pass, this seems to indicate that all the values in between should also work. However, suppose that one page prints correctly, but 99 pages do not. Now we do not know whether any set of more than one page works, so the first thing we would do would be to test for say ten pages, which is a value from the equivalence partition.

We recommend that you test the partitions separately from boundaries. This means choosing partition values that are NOT boundary values.

However, if you use the three-value boundary value approach, then you would have valid boundary values of 1, 2, 98 and 99, so having a separate equivalence value in addition to the extra two boundary values would not give much additional benefit. But notice that one equivalence value, for example 10, replaces both of the extra two boundary values (2 and 98). This is why EP with two value BVA is more efficient than three-value BVA.

Which partitions and boundaries you decide to exercise (you do not need to test them all), and which ones you decide to test first, depends on your test objectives. If your goal is the most thorough approach, then follow the procedure of testing valid partitions first, then invalid partitions, then valid boundaries and finally invalid boundaries. However, if you are under time pressure (and who isn't?), then you can't test as much as you would like. Now your test objectives will help you decide what to test. If you are after user confidence of typical transactions with a minimum number of tests, you might do valid partitions only. If you want to find as many defects as possible as quickly as possible, you may start with boundary values, both valid and invalid. If you want confidence that the system will handle bad inputs correctly, you may do mainly invalid partitions and boundaries. Your previous experience of types of defects can help you find similar defects; for example, if there are typically a lot of boundary defects, then you would start by testing boundaries.

EP and BVA are described in most testing books, including Myers [2011], Copeland [2004], Kaner, Padmanabhan and Hoffman [2013] and Jorgensen [2014]. EP and BVA are described in ISO/IEC/IEEE 29119-4 [2015], including designing tests and measuring coverage.

4.2.3 Decision table testing

The techniques of EP and BVA are often applied to specific situations or inputs. However, if different combinations of inputs result in different actions being taken, this can be more difficult to show using EP and BVA, which tend to be more focused on the user interface. The other two specification-based techniques, **decision table testing** and state transition testing, are more focused on business logic or business rules.

Decision table testing A black-box test technique in which test cases are designed to execute the combinations of inputs and/or stimuli (causes) shown in a decision table.

A decision table is a good way to deal with combinations of things (for example inputs). This technique is sometimes also referred to as a 'cause-effect' table. The reason for this is that there is an associated logic diagramming technique called 'cause-effect graphing' which was sometimes used to help derive the decision table. (Myers describes this as a combinatorial logic network [Myers 2011]). However, most people find it more useful just to use the table described in Copeland [2004].

If you begin using decision tables to explore what the business rules are that should be tested, you may find that the analysts and developers find the tables very helpful and want to begin using them too. Do encourage this, as it will make your job easier in the future. Decision tables provide a systematic way of stating complex business rules, which is useful for developers as well as for testers. Decision tables can be used in test design whether or not they are used in development, as they help testers explore the effects of combinations of different inputs and other software states that must correctly implement business rules. Helping the developers do a better job can also lead to better relationships with them.

Testing combinations can be a challenge, as the number of combinations can often be huge. Testing all combinations may be impractical, if not impossible. We have to be satisfied with testing just a small subset of combinations, but it's not easy to choose which combinations to test and which not to test. If you don't have a systematic way of selecting combinations, an arbitrary subset will be used and this may well result in an ineffective test effort.

Decision tables aid the systematic selection of effective test cases and can have the beneficial side-effect of finding problems and ambiguities in the specification. It is a technique that works well in conjunction with EP. The combination of conditions explored may be combinations of equivalence partitions.

In addition to decision tables, there are other techniques that deal with testing combinations of things: pairwise testing and orthogonal arrays. These are described in Copeland [2004]. Other sources of techniques are Pol, Teunissen and van Veenendaal [2001] and Black [2007]. Decision tables and cause–effect graphing are described in ISO/IEC/IEEE 29119-4 [2015], including designing tests and measuring coverage.

Using decision tables for test design

The first task is to identify a suitable function or subsystem that has a behaviour which reacts according to a combination of inputs or events. The behaviour of interest must not be too extensive (that is, should not contain too many inputs) otherwise the number of combinations will become cumbersome and difficult to manage. It is better to deal with large numbers of conditions by dividing them into subsets and dealing with the subsets one at a time.

Once you have identified the aspects that need to be combined, then you put them into a table, listing all the combinations of True and False for each of the aspects. Take an example of a loan application, where you can enter the amount of the monthly repayment or the number of years you want to take to pay it back (the term of the loan). If you enter both, we assume that the system will make a compromise between the two if they conflict. The two conditions are the repayment amount and the term, so we put them in a table (see Table 4.2).

TABLE 4.2 Empty decision table

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered				
Term of loan has been entered				

Next, we will identify all of the combinations of True and False (see Table 4.3). With two conditions, each of which can be True or False, we will have four combinations (two to the power of the number of things to be combined). Note that if we have three things to combine, we will have eight combinations, with four things, there are 16, etc. This is why it is good to tackle small sets of combinations at a time. In order to keep track of which combinations we have, we will alternate True and False on the bottom row, put two Trues and then two Falses on the row above the bottom row, etc., so the top row will have all Trues and then all Falses (and this principle applies to all such tables).

TABLE 4.3 Decision table with input combinations

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F

We are using T and F for the inputs in our example; we could also have used Y and N or 1 (the number one) and 0 (zero) respectively. The conditions can also be numbers, numeric ranges or enumerated types (for example red, green or blue).

The next step (at least for this example) is to identify the correct outcome for each combination (see Table 4.4). In this example, we can enter one or both of the two fields. Each combination is sometimes referred to as a rule. In this example, we are using Y for the actions which should occur and leaving it blank if that action should not occur. Other options are to use X or 1 if an action should occur, and N, F, '–' or 0 for actions that should not occur. We could use a number or range of numbers if an outcome occurs for those numbers and does not occur for others. You could also use discrete values, for example an action should occur if an input is red, which does not occur if it is green or yellow.

TABLE 4.4 Decision table with combinations and outcomes

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount	Y	Y		
Process term	Y		Y	

At this point, we may now realize that we had not thought about what happens if the customer doesn't enter anything in either of the two fields. The table has highlighted a combination that was not mentioned in the specification for this example. We could assume that this combination should result in an error message, so we need to add another action (see Table 4.5). This highlights the strength of this technique to discover omissions and ambiguities in specifications. It is not unusual for some combinations to be omitted from specifications; therefore this is also a valuable technique to use when reviewing the test basis.

TABLE 4.5 Decision table with additional outcome

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount	Y	Y		
Process term	Y		Y	
Error message				Y

Suppose we change our example slightly, so that the customer is not allowed to enter both repayment and term. Now our table will change, because there should also be an error message if both are entered, so it will look like Table 4.6.

TABLE 4.6 Decision table with changed outcomes

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount		Y		
Process term			Y	
Error message	Y			Y

You might notice now that there is only one Yes in each column, that is, our actions are mutually exclusive: only one action occurs for each combination of conditions. We could represent this in a different way by listing the actions in the cell of one row, as shown in Table 4.7. Note that if more than one action results from any of the combinations, then it would be better to show them as separate rows rather than combining them into one row.

TABLE 4.7 Decision table with outcomes in one row

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Result	Error message	Process loan amount	Process term	Error message

The final step of this technique is to write test cases to exercise each of the four rules in our table.

In this example we started by identifying the input conditions and then identifying the outcomes. However, in practice it might work the other way around – we can see that there are a number of different outcomes and have to work back to understand what combination of input conditions actually drive those outcomes. The technique works just as well doing it in this way and may well be an iterative approach as you discover more about the rules that drive the system.

Credit card worked example

Let's look at another example. If you are a new customer opening a credit card account, you will get a 15% discount on all your purchases today. If you are an existing customer and you hold a loyalty card, you get a 10% discount. If you have a coupon, you can get 20% off today (but it cannot be used with the new customer discount). Discount amounts are added, if applicable. This is shown in Table 4.8.

In Table 4.8, the conditions and actions are listed in the left-hand column. All the other columns in the decision table each represent a separate rule, one for each combination of conditions. We may choose to test each rule/combination and if there are only a few, this will usually be the case. However, if the number of rules/combinations is large we are more likely to sample them by selecting a rich subset for testing.

Note that we have put X for the discount for two of the columns (Rules 1 and 2). This means that this combination should not occur. You cannot be both a new customer and already hold a loyalty card! There should be an error message stating this, but even if we don't know what that message should be, it will still make a good test.

TABLE 4.8 Decision table for credit card example

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
New customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Actions								
Discount (%)	X	X	20	15	30	10	20	0

Because we have exactly the same action (and presumably the same error message) for both columns 1 and 2, we can see that having a coupon makes no difference to the outcome, that is, we do not care whether you have a coupon or not, as it makes no difference to the actions or outcome. This means that we can slightly collapse or shorten the table as shown in Table 4.9. We have put a dash (–) to show that it does not matter whether coupon is True or False. This is not very significant for such a small example. We could also put 'N/A' (Not Applicable) to show that this value does not affect the outcome. We can combine two or more columns in this way, as long as they all have the same resulting actions or outcomes. See Copeland [2004] for more information.

With more complex decision tables, being able to collapse the table by combining columns (or deleting a column) can make the table much easier to read and understand. For clarity, we have just removed 'Rule 2' and kept the other rule numbers as they were. This could be changed to make the rule numbers sequential in the final table. More information about collapsing decision tables is covered in the ISTQB Advanced Level Test Analyst Syllabus.

TABLE 4.9 Collapsed decision table for credit card example

Conditions	Rule 1	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
New customer (15%)	T	T	T	F	F	F	F
Loyalty card (10%)	T	F	F	T	T	F	F
Coupon (20%)	–	T	F	T	F	T	F
Actions							
Discount (%)	X	20	15	30	10	20	0

We have made an assumption in Rule 3. Since the coupon has a greater discount than the new customer discount, we assume that the customer will choose 20% rather than 15%. We cannot add them, since the coupon cannot be used with the new customer discount. The 20% action is an assumption on our part, and we should check

that this assumption (and any other assumptions that we make) is correct, by asking the person who wrote the specification or the users.

For Rule 5, however, we can add the discounts, since both the coupon and the loyalty card discount should apply (at least that is our assumption).

Rules 4, 6 and 7 have only one type of discount and Rule 8 has no discount, so is 0%.

If we are applying this technique thoroughly, we would have one test for each column or rule of our decision table. The advantage of doing this is that we may test a combination of things that otherwise we might not have tested and that could find a defect. Coverage of decision table testing is measured by the number of columns that have at least one test case, divided by the total number of columns.

However, if we have a lot of combinations, it may not be possible or sensible to test every combination. Or if we are time-constrained, we may not have time to test all combinations. Do not just assume that all combinations need to be tested; it is better to prioritize and test the most important combinations. Having the full table enables us to see which combinations we decided to test and which not to test this time.

There may also be many different actions as a result of the combinations of conditions. In the example above we just had one: the discount to be applied. The decision table shows which actions apply to each combination of conditions.

In the example above all the conditions are binary, that is, they have only two possible values: True or False (or, if you prefer Yes or No). Often it is the case that conditions are more complex, having potentially many possible values. Where this is the case the number of combinations is likely to be very large, so the combinations may only be sampled rather than exercising all of them.

4.2.4 State transition testing

State transition testing is used where some aspect of the system can be described in what is called a 'finite state machine'. This simply means that the system can be in a limited (finite) number of different states, and the transitions from one state to another are determined by the rules of the 'machine'. This is the model on which the system and the tests are based. Any system where you get a different output for the same input, depending on what has happened before, is a finite state system. A finite state system is often shown as a state diagram (see Figure 4.2).

State transition testing (finite state testing) A black-box test technique using a state transition diagram or state table to derive test cases to evaluate whether the test item successfully executes valid transitions and blocks invalid transitions.

For example, if you request to withdraw $100 from a bank ATM, you may be given cash. Later you may make exactly the same request but be refused the money (because your balance is insufficient). This later refusal is because the state of your bank account has changed from having sufficient funds to cover the withdrawal to having insufficient funds. When the same event can result in two different transitions, depending on some True/False condition (in this case sufficient funds), this is referred to as a 'guard condition' on the transition. The funds will be dispensed only when the guard condition of sufficient funds is True. Guard conditions are shown on a state diagram in square brackets by the transition that they are guarding. The transaction that caused your account to change its state was probably the earlier withdrawal. A state diagram can represent a model from the point of view of the system, the account or the customer.

Another example is a word processor. If a document is open, you are able to close it. If no document is open, then Close is not available. After you choose Close once, you cannot choose it again for the same document unless you open that document. A document thus has two states: open and closed.

A state transition model has four basic parts:

- The states that the software may occupy (open/closed or funded/insufficient funds).
- The transitions from one state to another (not all transitions are allowed).
- The events that cause a transition (closing a file or withdrawing money).
- The actions that result from a transition (an error message or being given your cash).

Note that in any given state, one event can cause only one action, but that the same event from a different state may cause a different action and a different end state.

We will look first at test cases that execute valid state transitions.

Figure 4.2 shows an example of entering a PIN to a bank account. The states are shown as circles, the transitions as lines with arrows and the events as the text near the transitions. We have not shown the actions explicitly on this diagram, but they would be a message to the customer saying things such as 'Please enter your PIN'.

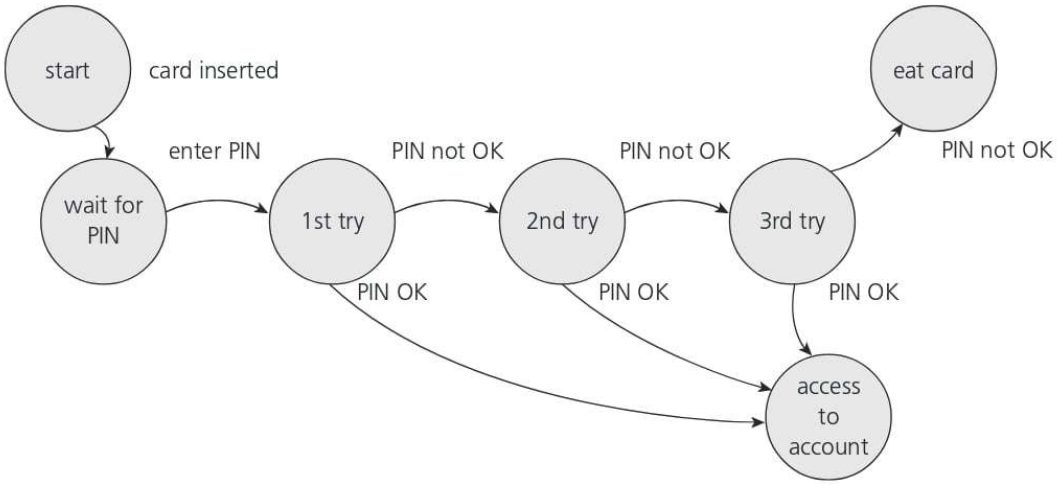


FIGURE 4.2 State diagram for PIN entry

The state diagram shows seven states but only four possible events (Card inserted, Enter PIN, PIN OK and PIN not OK). We have not specified all of the possible transitions here: there would also be a time-out from 'wait for PIN' and from the three tries. The system would go back to the start state after the time had elapsed and would probably eject the card. There would also be a transition from the 'eat card' state back to the start state. We have not specified all the possible events either – there would be a 'cancel' option from 'wait for PIN' and from the three tries, which would also go back to the start state and eject the card. The 'access account' state would be the beginning of another state diagram showing the valid transactions that could now be performed on the account. This state diagram, even though it is incomplete, still gives us information on which to design some useful tests and to explain the state transition technique.

In deriving test cases, we may start with a typical scenario. A sensible first test case here would be the normal situation, where the correct PIN is entered the first time. To be more thorough, we may want to make sure that we cover every state (that is, at least one test goes through each state) or we may want to cover every transition. A second test (to visit every state) would be to enter an incorrect PIN

each time, so that the system eats the card. We still have not tested every transition yet. In order to do that, we would want a test where the PIN was incorrect the first time but OK the second time, and another test where the PIN was correct on the third try. These tests are probably less important than the first two.

Note that a transition does not need to change to a different state (although all of the transitions shown above do go to a different state). There could be a transition from 'access account' which just goes back to 'access account' for an action such as 'request balance'.

Test conditions can be derived from the state diagram in various ways. Each state can be noted as a test condition, as can each transition. In the Syllabus, we need to be able to identify the coverage of a set of tests in terms of states or transitions.

Tests can be designed to cover a typical sequence of states, to visit all states, to exercise every transition, to exercise specific transition sequences, or to test invalid transitions (see below).

Going beyond the level expected in the Syllabus, we can also consider transition pairs and triples and so on. Coverage of all individual transitions is also known as 0-switch coverage, coverage of transition pairs is 1-switch coverage, coverage of transition triples is 2-switch coverage, etc. Deriving test cases from the state transition model is a black-box approach. Measuring how much you have tested (covered) is getting close to a white-box perspective. However, state transition testing is regarded as a black-box technique. More information on coverage criteria for state transition testing is covered in the ISTQB Advanced Level Test Analyst Syllabus.

One of the advantages of the state transition technique is that the model can be as detailed or as abstract as you need it to be. Where a part of the system is more important (that is, requires more testing) a greater depth of detail can be modelled. Where the system is less important (requires less testing), the model can use a single state to signify what would otherwise be a series of different states.

Testing for invalid transitions

Deriving tests only from a state diagram or chart (also known as a state graph or chart) is very good for seeing the valid transitions, but we may not easily see the negative tests, where we try to generate invalid transitions. In order to see the total number of combinations of states and transitions, both valid and invalid, a state table is useful.

The state table lists all the states down one side of the table and all the events that cause transitions along the top (or vice versa). Each cell then represents a state-event pair. The content of each cell indicates which state the system will move to when the corresponding event occurs while in the associated state. This will include possible erroneous events – events that are not expected to happen in certain states. These are negative test conditions.

Table 4.10 lists the states in the first column and the possible events across the top row. So, for example, if the system is in State 1, inserting a card will take it to State 2. If we are in State 2, and a valid PIN is entered, we go to State 6 to access the account. In State 2 if we enter an invalid PIN, we go to State 3. We have put a dash in the cells that should be impossible, that is, they represent invalid transitions from that state.

We have put a question mark for two cells, where we enter either a valid or invalid PIN when we are accessing the account. Perhaps the system will take our PIN number as the amount of cash to withdraw? It might be a good test! Most of the other invalid cells would be physically impossible in this example. Invalid (negative) tests will attempt to generate invalid transitions, transitions that should not be possible (but often make good tests when it turns out they are possible).

TABLE 4.10 State table for the PIN example

	Insert card	Valid PIN	Invalid PIN
S1) Start state	S2	–	–
S2) Wait for PIN	–	S6	S3
S3) 1st try invalid	–	S6	S4
S4) 2nd try invalid	–	S6	S5
S5) 3rd try invalid	–	–	S7
S6) Access account	–	?	?
S7) Eat card	S1 (for new card)	–	–

State transition testing is used for screen-based applications, for example an ATM, and also within the embedded software industry. It is useful for modelling business scenarios which have specific states, or for testing navigation around screens.

A more extensive description of state machines is found in Marick [1994]. State transition testing is also described in Craig and Jaskiel [2002], Copeland [2004], Beizer [1990], Broekman and Notenboom [2003], and Black [2007]. State transition testing is described in ISO/IEC/IEEE 29119-4 [2015], including designing tests and coverage measures.

4.2.5 Use case testing

Use case testing (scenario testing, user scenario testing) A black-box test technique in which test cases are designed to execute scenarios of use cases.

Use case testing is a technique that helps us identify test cases that exercise the whole system on a transaction by transaction basis from start to finish. They are described by Ivar Jacobson in his book *Object-Oriented Software Engineering: A Use Case Driven Approach* [Jacobson 1992]. Information on use cases can also be found at Omg.org in UML 2.5 [2017].

A use case is a description of a particular use of the system by an actor (a human user of the system, external hardware or other components or systems). Each use case describes the interactions the actor has with the subject (i.e. the component or system to which the use case is applied), in order to achieve a specific task (or, at least, produce something of value to the actor). Use cases are a sequence of steps that describe the interactions between the actor and the subject. A use case is a specific way of designing interactions with software items, incorporating requirements for the software functions represented by the use cases.

Each use case specifies some behaviour that a subject can perform in collaboration with one or more actors. The interactions between the actors and the subject may result in changes to the state of the subject. These interactions can be represented graphically by workflows activity diagrams or business process models. They can also be described as a series of steps or even in natural language.

Use cases are defined in terms of the actor, not the system, describing what the actor does and what the actor sees rather than what inputs the system expects and

what the system outputs. They often use the language and terms of the business rather than technical terms, especially when the actor is a business user. They serve as the foundation for developing test cases, mostly at the system and acceptance testing levels.

Use cases can uncover integration defects, that is, defects caused by the incorrect interaction between different components. Used in this way, the actor may be something that the system interfaces to such as a communication link or subsystem.

Use cases describe the process flows through a system based on its most likely use. This makes the test cases derived from use cases particularly good for finding defects in the real-world use of the system (that is, the defects that the users are most likely to come across when first using the system). Each use case usually has a mainstream (or most likely) scenario and sometimes additional alternative branches (covering, for example, special cases or exceptional conditions and error conditions, such as system response and recovery from errors in programming, the application and communication). Each use case must specify any preconditions that need to be met for the use case to work. Use cases must also specify postconditions that are observable results and a description of the final state of the system after the use case has been executed successfully.

The PIN example that we used for state transition testing could also be defined in terms of use cases, as shown in Figure 4.3. We show a success scenario and the extensions (which represent the ways in which the scenario could fail to be a success).

For use case testing, we would have a test of the success scenario and one test for each extension. In this example, we may give extension 4b a higher priority than 4a from a security point of view.

System requirements can also be specified as a set of use cases. This approach can make it easier to involve the users in the requirements gathering and definition process.

	Step	Description
Main Success Scenario **A: Actor** **S: System**	1	A: Inserts card
	2	S: Validates card and asks for PIN
	3	A: Enters PIN
	4	S: Validates PIN
	5	S: Allows access to account
Extensions	2a	Card not valid S: Display message and reject card
	4a	PIN not valid S: Display message and ask for re-try (twice)
	4b	PIN invalid 3 times S: Eat card and exit

FIGURE 4.3 Partial use case for PIN entry

Coverage of use case testing can be measured by the percentage of use case behaviours tested divided by the total number of use case behaviours. It is possible to measure coverage of normal behaviour, exceptional and/or error behaviour, or all behaviours. More information on coverage criteria for use case testing is covered in the ISTQB Advanced Level Test Analyst Syllabus.

4.3 WHITE-BOX TEST TECHNIQUES

SYLLABUS LEARNING OBJECTIVES FOR 4.3 WHITE-BOX TEST TECHNIQUES (K2)

FL-4.3.1 Explain statement coverage (K2)

FL-4.3.2 Explain decision coverage (K2)

FL-4.3.3 Explain the value of statement and decision coverage (K2)

In this section we will look in detail at the concept of coverage and how it can be used to measure some aspects of the thoroughness of testing. In order to see how coverage actually works, we will use some code-level examples (although coverage also applies to other levels such as business procedures). In particular, we will show how to measure coverage of statements and decisions, and how to write test cases to extend coverage if it is not 100%.

As mentioned, we will illustrate white-box test techniques that would typically apply at the component level of testing. At this level, white-box techniques focus on the structure of a software component, such as statements, decisions, branches or even distinct paths. These techniques can also be applied at the integration level. At this level, white-box techniques can examine structures such as a call tree, which is a diagram that shows how modules call other modules. These techniques can also be applied at the system level. At this level, white-box techniques can examine structures such as a menu structure, business process or web page structure.

There are also more advanced white-box test techniques at component level, particularly for safety-critical, mission-critical or high-integrity environments to achieve higher levels of coverage. For more information on these techniques see the ISTQB Advanced Level Technical Test Analyst Syllabus.

In this section, look for the definitions of the Glossary terms **decision coverage** and **statement coverage**.

White-box test techniques serve two purposes: coverage measurement and structural test case design. They are often used first to assess the amount of testing performed by tests derived from black-box test techniques, that is, to assess coverage. They are then used to design additional tests with the aim of increasing the coverage.

White-box test techniques used to design tests are a good way of generating additional test cases that are different from existing tests. They can help ensure more breadth of testing, in the sense that test cases that achieve 100% coverage in any measure will be exercising all parts of the software from the point of view of the items being covered.

What is coverage?

Coverage measures the amount of testing performed by a set of tests that may have been derived in a different way, for example using black-box techniques. Wherever we can count things and can tell whether or not each of those things has been tested by some test, then we can measure coverage. The basic coverage measure is:

$$\text{Coverage} = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}} \times 100\%$$

where the coverage item is whatever we have been able to count and see whether a test has exercised or used this item.

A Christmas gift to one of the authors was a map of the world where you can scratch off the countries you have visited. This is a very good analogy for coverage. If I visit a small country, for example Belgium, it seems quite OK to scratch off the whole country. But if I visit the US, Canada, Australia, China or Russia, then do I scratch off the whole country? Even if I have only visited one or a few cities, I can still count the whole country and this seems not quite right. But this would be country coverage. I can count the countries I have visited, divided by the total number of countries in the world, and get a percentage, for example around 18% for visiting 36 countries. However, if you look at the map, it looks like I have covered around 25% of the area (since I have been to the US, Canada, Australia and China). If I were to do area-of-the-map coverage, I would get a higher coverage percentage, but I would not have seen any more of the world. If I now visit St Petersburg in Russia, I increase my country coverage by less than 1%, but increase my area coverage by 12.5%! Why are we talking about this map? Because it highlights some of the same aspects and pitfalls of coverage for components and systems.

There are several dangers (pitfalls or caveats) in using coverage measures:

- 100% coverage does *not* mean 100% tested! Coverage techniques measure only one dimension of a multi-dimensional concept. Country coverage is a higher level than state, county or city coverage.
- Two different test cases may achieve exactly the same coverage but the input data of one may find an error that the input data of the other does not. If you do not execute a line or block of code that contains a bug, you are guaranteed not to see the failures that bug can cause. However, just because you did execute that line or block, does not guarantee that you will see the failures that bug can cause. There may be many different data combinations that exercise the same True/False decision outcome, but some will cause a failure and others will not.
- Coverage looks only at what *has* been written, that is the code itself. It cannot say anything about the software that has *not* been written. If a specified function has not been implemented, black-box test techniques will reveal this. If a function was omitted from the specification, then experience-based techniques may find it. But white-box techniques can only look at a structure which is already there.
- Just because some coverage item has been covered, this does NOT mean that this part of the system is actually doing what it should. Coverage only assesses whether or not you have exercised something, not whether that test passed or failed or whether it was a good test worth running. Coverage says nothing about the quality of either the system or the tests.

So is coverage worth measuring? Yes, coverage can be useful. It is a way of assessing one aspect of thoroughness. But it is best used when you understand exactly what you are measuring, and are aware of the pitfalls, particularly if you are reporting coverage to stakeholders.

Types of coverage

Coverage can be measured based on a number of different structural elements in a system or component. Coverage can be measured at component testing level, integration testing level or at system or acceptance testing levels. For example, at system or acceptance level, the coverage items may be requirements, menu options, screens or typical business transactions. Other coverage measures include things such as database structural elements (records, fields and sub-fields) and files. It is worth checking for any new tools, as the test tool market develops quite rapidly.

At integration level, we could measure coverage of interfaces or specific interactions that have been tested. The call coverage of APIs, modules, objects or procedure calls can also be measured (and is supported by tools to some extent).

There is good tool support for code coverage, that is, for component-level testing. We can measure coverage for each of the black-box test techniques as well:

- Equivalance partitioning: percentage of equivalence partitions exercised (we could measure valid and invalid partition coverage separately if this makes sense).
- Boundary value analysis: percentage of boundaries exercised (we could also separate valid and invalid boundaries if we wished).
- Decision tables: percentage of business rules or decision table columns tested.
- State transition testing: there are a number of possible coverage measures:
  - Percentage of states visited.
  - Percentage of (valid) transitions exercised (this is known as Chow's 0-switch coverage).
  - Percentage of pairs of valid transitions exercised ('transition pairs' or Chow's 1-switch coverage) – and longer series of transitions, such as transition triples, quadruples, etc.
  - Percentage of invalid transitions exercised (from the state table).

The coverage measures for black-box techniques would apply at whichever test level the technique has been used (for example system or component level).

When coverage is discussed by product owners, business analysts, system testers or users, it most likely refers to the percentage of requirements that have been tested by a set of tests. This may be measured by a tool such as a requirements management tool or a test management tool.

However, when coverage is discussed by programmers, it most likely refers to the coverage of code, where the structural elements can be identified using a tool. We will cover statement and decision coverage shortly. However, at this point, note that the word coverage is often misused to mean, 'How many or what percentage of tests have been run?' This is **NOT** what the term coverage refers to. Coverage is the coverage **of** something else **by** the tests. The percentage of tests run should be called test completeness or something similar.

Statements and decision outcomes are both structures that can be measured in code and there is good tool support for these coverage measures. Code coverage

is normally done in component and component integration testing, if it is done at all. If someone claims to have achieved code coverage, it is important to establish exactly what elements of the code have been covered, as statement coverage (often what is meant) is significantly weaker than decision coverage or some of the other code coverage measures.

How to measure coverage

For most practical purposes, coverage measurement is something that requires tool support. However, knowledge of the steps typically taken to measure coverage is useful in understanding the relative merits of each technique. Our example assumes an intrusive coverage measurement tool that alters the code by inserting instrumentation:

1. Decide on the structural element to be used, that is, the coverage items to be counted.
2. Count the structural elements or items.
3. Instrument the code.
4. Run the tests for which coverage measurement is required.
5. Using the output from the instrumentation, determine the percentage of elements or items exercised.

Instrumenting the code (step 3) involves inserting code alongside each structural element in order to record when that structural element has been exercised. Determining the actual coverage measure (step 5) is then a matter of analyzing the recorded information.

Coverage measurement of code is best done using tools (as described in Chapter 6) and there are a number of such tools on the market. These tools can help to support increased quality and productivity of testing. They support increased quality by ensuring that more structural aspects are tested, so defects on those structural paths can be found (and fixed, otherwise quality has not increased). They support increased productivity and efficiency by highlighting tests that may be redundant, that is, testing the same structure as other tests, although this is not necessarily a bad thing, since we may find a defect testing the same structure with different data.

In common with all white-box test techniques, code coverage techniques are best used on areas of software code where more thorough testing is required. Safety-critical code, code that is vital to the correct operation of a system, and complex pieces of code are all examples of where white-box techniques are particularly worth applying. For example, some standards for safety-critical systems such as avionics and vehicle control, require white-box coverage for certain types of system. White-box test techniques should normally be used in addition to black-box and experience-based test techniques rather than as an alternative to them.

White-box test design

If you are aiming for a given level of coverage (say 95%) but you have not reached your target (for example you only have 87% so far), then additional test cases can be designed with the aim of exercising some or all of the structural elements not yet reached. This is white-box or structure-based test design. These new tests are then run through the instrumented code and a new coverage measure is calculated. This is repeated until the required coverage measure is achieved (or until you decide that your goal was too ambitious!). Ideally all the tests ought to be run again on the un-instrumented code.

We will look at some examples of structure-based coverage and test design for statement and decision testing below.

4.3.1 Statement testing and coverage

Statement coverage
The percentage of executable statements that have been exercised by a test suite.

Statement coverage is calculated by:

$$\text{Statement coverage} = \frac{\text{Number of statements exercised}}{\text{Total number of statements}} \times 100\%$$

Studies and experience in the industry have indicated that what is considered reasonably thorough black-box testing may actually achieve only 60% to 75% statement coverage. Typical ad hoc testing is likely to achieve only around 30%, leaving 70% of the statements untested.

Different coverage tools may work in slightly different ways, so they may give different coverage figures for the same set of tests on the same code, although at 100% coverage they should be the same.

We will illustrate the principles of coverage on code. In order to explain our examples, we will use two types of code examples, one a basic pseudo-code – this is not any specific programming language, but should be readable and understandable to you, even if you have not done any programming yourself – and the second is more like javascript. Both give the same control flow. We have omitted the set-up code that is needed to actually run the code, to concentrate on the logic.

For example, consider Code samples 4.1a and 4.1b.

```
READ A
READ B
IF A > B THEN C = 0
ENDIF
```

Code sample 4.1a

```
let a = Number(args[2])
let b = Number(args[3])
if (a > b) {
        c = 0
}
```

Code sample 4.1b

To achieve 100% statement coverage of this code segment just one test case is required, one which ensures that variable A contains a value that is greater than the value of variable B, for example, A = 12 and B = 10. Note that here we are doing structural test *design* first, since we are choosing our input values in order to ensure statement coverage.

Let's look at an example where we measure coverage first. In order to simplify the example, we will regard each line as a statement. Different tools and methods may count different things as statements, but the basic principle is the same however they are counted. A statement may be on a single line, or it may be spread over several lines. One line may contain more than one statement, just one statement or only part of a statement. Some statements can contain other statements inside them. In Code samples 4.2, we have two read statements, one assignment statement and then one IF statement on three lines, but the IF statement contains another statement (print) as part of it.

```
1 READ A
2 READ B
3 C = A + 2*B
4 IF C > 50 THEN
5      PRINT 'Large C'
6 ENDIF
```

Code sample 4.2a

```
1 let a = Number(args[2])
2 let b = Number(args[3])
3 let c = a + 2*b
4 if (c > 50) {
5     console.log('C large')
6 }
```

Code sample 4.2b

Although it is not completely correct, we have numbered each line and will regard each line as a statement. Some tools may group statements that would always be executed together in a basic block which is regarded as a single statement. However, we will just use numbered lines to illustrate the principle of coverage of statements (lines). Let's analyze the coverage of a set of tests on our six-statement program:

TEST SET 1

Test 1_1: A = 2; B = 3
Test 1_2: A = 0; B = 25
Test 1_3: A = 47, B = 1

Which statements have we covered?

- In Test 1_1, the value of C will be 8, so we will cover the statements on lines 1 to 4 and line 6.
- In Test 1_2, the value of C will be 50, so we will cover exactly the same statements as Test 1_1.
- In Test 1_3, the value of C will be 49, so again we will cover the same statements.

Since we have covered five out of six statements, we have 83% statement coverage (with three tests). What test would we need in order to cover statement 5, the one statement that we haven't exercised yet? How about this one:

Test 1_4: A = 20; B = 25

This time the value of C is 70, so we will print 'Large C' and we will have exercised all six of the statements, so now statement coverage = 100%. Notice that we measured coverage first, and then designed a test to cover the statement that we had not yet covered.

Note that Test 1_4 on its own is more effective (towards our goal of achieving 100% statement coverage) than the first three tests together. Just taking Test 1_4 on its own is also more efficient than the set of four tests, since it has used only one test instead of four. Being more effective and more efficient is the mark of a good test technique.

4.3.2 Decision testing and coverage

Decision coverage
The coverage of decision outcomes. (Note: this is the Glossary definition at publication, but a fuller definition would be: The percentage of decision outcomes that have been exercised by a test suite.)

A decision is an IF statement, a loop control statement (for example DO–WHILE or REPEAT–UNTIL), or a CASE statement, where there are two or more possible exits or outcomes from the statement. With an IF statement, the exit can either be True or False, depending on the value of the logical condition that comes after IF. With a loop control statement, the outcome is either to perform the code within the loop or not – again a True or False exit. **Decision coverage** is calculated by:

$$\text{Decision coverage} = \frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} \times 100\%$$

What feels like reasonably thorough black-box testing may achieve only 40 to 60% decision coverage. Typical ad hoc testing may cover only 20% of the decisions, leaving 80% of the possible outcomes untested. Even if your testing seems reasonably thorough from a functional or black-box perspective, you may have only covered two-thirds or three-quarters of the decisions. Decision coverage is stronger than statement coverage. It 'subsumes' statement coverage – this means that 100% decision coverage always guarantees 100% statement coverage. Any stronger coverage measure may require more test cases to achieve 100% coverage.

Let's go back to Code samples 4.2 again. We saw earlier that just one test case was required to achieve 100% statement coverage. However, decision coverage requires each decision to have had both a True and False outcome. Therefore, to achieve 100% decision coverage, a second test case is necessary where A is less than or equal to B. This will ensure that the decision statement IF A > B has a False outcome. So one test is sufficient for 100% statement coverage, but two tests are needed for 100% decision coverage. Note that 100% decision coverage guarantees 100% statement coverage, but *not* the other way around!

Now let us consider slightly different Code samples shown in Code samples 4.3a and 4.3b.

```
1 READ A
2 READ B
3 C = A - 2*B
4 IF C < 0 THEN
5      PRINT 'C negative'
6 ENDIF
```

Code sample 4.3a

```
1 let a = Number(args[2])
2 let b = Number(args[3])
3 let c = a - 2*b
4 if (c < 0) {
5     console.log('C negative')
6 }
```

Code sample 4.3b

Let us suppose that we already have the following test, which gives us 100% statement coverage for Code samples 4.3.

TEST SET 2

Test 2_1: A = 20; B = 15

Which decision outcomes have we exercised with our test? The value of C is −10, so the condition C < 0 is True, so we will print 'C negative' and we have exercised the True outcome from that decision statement. But we have not exercised the decision outcome of False. What other test would we need to exercise the False outcome and to achieve 100% decision coverage?

Before we answer that question, let's have a look at another way to represent this code. Sometimes the decision structure is easier to see in a control flow diagram (see Figure 4.4).

The dotted line shows where Test 2_1 has gone and clearly shows that we have not yet had a test that takes the False exit from the IF statement.

Let's modify our existing test set by adding another test:

TEST SET 2

Test 2_1: A = 20; B = 15
Test 2_2: A = 10; B = 2

This now covers both of the decision outcomes, True (with Test 2_1) and False (with Test 2_2). If we were to draw the path taken by Test 2_2, it would be a straight line from the read statement down the False exit and through the ENDIF. Note that we could have chosen other numbers to achieve either the True or False outcomes.

4.3.3 The value of statement and decision testing

Coverage is a partial measure of some aspect of the thoroughness of testing; in this section, we have looked at two types of coverage, statement and decision. The value of statement and decision testing is in seeing what new tests are needed in order to achieve a higher level of coverage, whatever dimension of coverage we are looking at. By being more thorough (even in a very limited way), we are leaving less of the component or system completely untested.

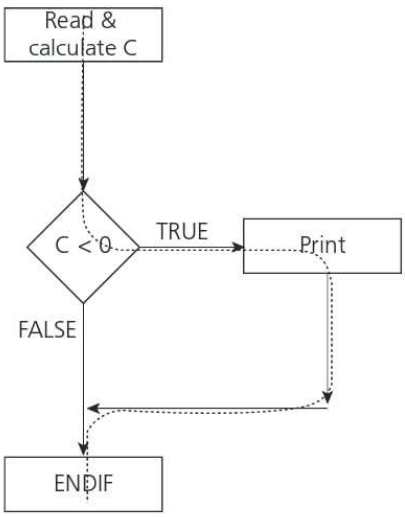


FIGURE 4.4 Control flow diagram for Code samples 4.3

One client decided to measure coverage and found that their supposedly thorough tests only reached 60% decision coverage. They decided to write more tests to increase this to 80%. Although they spent 3 weeks designing and implementing these new tests, they found enough high-severity defects to justify spending that time.

White-box coverage measures and related test techniques are described in ISO/IEC /IEEE 29119-4 [2015]. White-box test techniques are also discussed in Copeland [2003] and Myers [2011]. A good description of the graph theory behind structural testing can be found in Jorgensen [2014], and Hetzel [1988] also shows a structural approach. Pol, Teunissen and van Veenendaal [2001] describes a white-box approach called an algorithm test.

4.4 EXPERIENCE-BASED TEST TECHNIQUES

SYLLABUS LEARNING OBJECTIVES FOR 4.4 EXPERIENCE-BASED TEST TECHNIQUES (K2)

FL-4.4.1 Explain error guessing (K2)

FL-4.4.2 Explain exploratory testing (K2)

FL-4.4.3 Explain checklist-based testing (K2)

In this section we will look at three experience-based techniques, why and when they are useful, and how they fit with black-box test techniques.

Although it is true that testing should be rigorous, thorough and systematic, this is not all there is to testing. There is a definite role for non-systematic techniques, that is, tests based on a person's knowledge, experience, imagination and intuition. The reason is that some defects are hard to find using more systematic approaches, so a good bug hunter can be very creative at finding those elusive defects.

One aspect which is more difficult for experience-based techniques is the measurement of coverage. In order to measure coverage, we need to have some idea of a full set of things that we could possibly test, coverage then being the percentage of that set that we did test. For example, we could look at the coverage of the items in the test charter, a checklist or a set of heuristics. We should also be able to use the traceability of tests to requirements or user stories and look at coverage of those.

In this section, look for the definitions of the Glossary terms **checklist-based testing**, **error guessing** and **exploratory testing**.

4.4.1 Error guessing

Error guessing A test technique in which tests are derived on the basis of the tester's knowledge of past failures, or general knowledge of failure modes.

Error guessing is a technique that is good to be used as a complement to other more formal techniques. The success of error guessing is very much dependent on the skill of the tester, as good testers know where the defects are most likely to lurk. Some people seem to be naturally good at testing. Others are good testers because they have a lot of experience either as a tester or working with a particular system and so are able to pin-point its weaknesses. This is why error guessing, used after more formal techniques have been applied to some extent, can be very effective. In using more formal techniques, the tester is likely to gain a better understanding of the system, what it does and how it works. With this better understanding, they are likely to be better at guessing ways in which the system may not work properly.

There are no rules for error guessing. The tester is encouraged to think of situations in which the software may not be able to cope. Here are some typical things to try: division by zero, blank (or no) input, empty files and the wrong kind of data (for example alphabetic characters where numeric are required). If anyone ever says of a system or the environment in which it is to operate 'That could never happen', it might be a good idea to test that condition, as such assumptions about what will and will not happen in the live environment are often the cause of failures.

Error guessing may be based on:

- How the application has worked in the past.
- What types of mistakes the developers tend to make.
- Failures that have occurred in other applications.

A structured approach to the error-guessing technique is to list possible defects or failures and to design tests that attempt to produce them. These defect and failure lists can be built based on the tester's own experience or that of other people, available defect and failure data, and from common knowledge about why software fails. This way of trying to force specific types of fault to occur is sometimes called an 'attack' or 'fault attack'. See Whittaker [2003].

4.4.2 Exploratory testing

Exploratory testing is a hands-on approach in which testers are involved in minimum planning and maximum test execution. The planning involves the creation of a test charter, a short declaration of the scope of a short (one- to two-hour) time-boxed test effort, the objectives and possible approaches to be used.

Exploratory testing
An approach to testing whereby the testers dynamically design and execute tests based on their knowledge, exploration of the test item and the results of previous tests.

The test design and test execution activities are performed in parallel, typically without formally documenting the test conditions, test cases or test scripts. The tests are informal, because they are not pre-defined or documented in advance in detail (although a test charter is most often written in advance). This does not mean that other, more formal testing techniques will not be used. For example, the tester may decide to use BVA, but will think through and test the most important boundary values without necessarily writing them down. Some notes will be written while the exploratory testing is going on, so that a report can be produced afterwards.

One typical way to organize and manage exploratory testing is to have sessions, hence this is also known as session-based testing. Each session is time-boxed, for example with a firm time limit of 90 minutes. A test charter will give a list of test conditions (sometimes referred to as objectives for the test session), but the testing does not have to conform completely to that charter, particularly if new areas of high risk are discovered in the session. The test session is completely dedicated to the testing, without extraneous interruptions.

Test logging is undertaken as test execution is performed, documenting (at a high level) the key aspects of what is tested, any defects found and any thoughts about possible further testing, possibly using test session sheets. A key aspect of exploratory testing is learning: learning by the tester about the software, its use, its strengths and its weaknesses. As its name implies, exploratory testing is about exploring, finding out about the software, what it does, what it does not do, what works and what does not work. The tester is constantly making decisions about what to test next and where to spend the (limited) time.

This is an approach that is most useful when there are no or poor specifications and when time is severely limited. It can also serve to complement other, more formal testing, helping to establish greater confidence in the software. In this way, exploratory testing can be used as a check on the formal test process by helping to ensure that the most serious defects have been found.

Exploratory testing is described in Kaner, Bach and Petticord [2002] and Copeland [2003]. Other ways of testing in an exploratory way (attacks) are described by Whittaker [2003].

4.4.3 Checklist-based testing

Checklist-based testing An experience-based test technique whereby the experienced tester uses a high-level list of items to be noted, checked, or remembered, or a set of rules or criteria against which a product has to be verified.

Checklist-based testing is testing based on experience, but that experience has been summarized and documented in a checklist. Testers use the checklist to design, implement and execute tests based on the items or test conditions found in the checklist. The checklist may be based on:

- experience of the tester
- knowledge, for example what is important for the user
- understanding of why and how software fails.

When using a checklist, the tester may modify it by adding new questions or things to check, or may just use what is already there. An experienced tester may be the one who writes the first version of the checklist as a way to help less experienced testers do better testing.

Checklists can be general, or more likely, aimed at particular areas such as different test types and levels. For example, a checklist for functional testing would be quite different from one aimed at testing non-function quality attributes.

If different people use the same checklist, it is more likely that there will a degree of consistency in what is tested. However, the actual tests executed may be quite different, since the checklist is only mentioning high-level items. This variability, even while using the same checklist, may help to uncover more defects and achieve different levels of coverage (if that is measured), even though it is less repeatable due to human creativity (which is a good thing).

CHAPTER REVIEW

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

From Section 4.1 (categories of test techniques), you should be able to give reasons why black-box, white-box, and experience-based approaches are useful, and be able to explain the characteristics and differences between these types of techniques. You should be able to list the factors that influence the selection of the appropriate test technique for a particular type of problem, such as the type of system, risk, customer requirements, models for use case modelling, requirements models or testing knowledge. You should know the Glossary terms **black-box test technique**, **coverage**, **experience-based test technique**, **test technique** and **white-box test technique**.

From Section 4.2, you should be able to write test cases from given software models using equivalence partitioning (EP), Boundary Value Analysis (BVA), decision table testing and state transition testing. You should understand and be able to apply each of these four techniques, understand what level and type of testing could use each technique and how coverage can be measured for each of them. You should also understand the concept and benefits of use case testing. You should know the Glossary terms **boundary value analysis**, **decision table testing**, **equivalence partitioning**, **state transition testing** and **use case testing**.

From Section 4.3, you should be able to describe the concept and importance of code coverage. You should be able to explain the concepts of statement and decision coverage and understand that these concepts can also be used at test levels other than component testing (such as business procedures at system test level). You should be able to write test cases from given control flows using statement testing and decision testing, and you should be able to assess statement and decision coverage for completeness. You should know the Glossary terms **coverage**, **decision coverage** and **statement coverage**.

From Section 4.4, you should be able to explain the reasons for writing test cases based on intuition, experience and knowledge about common defects and you should be able to compare experience-based techniques with black-box test techniques. You should know the Glossary terms **checklist-based testing**, **error guessing** and **exploratory testing**.

SAMPLE EXAM QUESTIONS

Question 1 Which of the following statements about checklist-based testing is true?

a. A checklist contains test conditions and detailed test cases and procedures.

b. A checklist can be used for functional testing, but not for non-functional testing.

c. Checklists should always be used exactly as written and should not be modified by the tester.

d. Checklists may be based on experience of why and how software fails.

Question 2 In a competition, ribbons are awarded as follows: less than 12 metres, no ribbon, a yellow ribbon up to 25 metres, a red ribbon up to 35 metres, and a blue ribbon for further than that.

What distances (in metres) would be chosen using BVA?

a. 0, 11, 12, 25, 26, 35, 36.

b. 11, 12, 13, 29, 30, 31, 40.

c. 7, 18, 32, 39.

d. 0, 12, 13, 26, 27, 36, 37.

Question 3 Which statement about tests based on increasing statement or decision coverage is true?

a. Increasing statement coverage may find defects where other tests have not taken both true and false outcomes.

b. Increasing statement coverage may find defects in code that was exercised by other tests.

c. Increasing decision coverage may find defects where other tests have not taken both true and false outcomes.

d. Increasing decision coverage may find defects in code that was exercised by other tests.

Question 4 Why are both black-box and white-box test techniques useful?

a. They find different types of defect.

b. Using more techniques is always better.

c. Both find the same types of defect.

d. Because specifications tend to be unstructured.

Question 5 What is a key characteristic of white-box test techniques?

a. They are mainly used to assess the structure of a specification.

b. They are used both to measure coverage and to design tests to increase coverage.

c. They are based on the skills and experience of the tester.

d. They use a formal or informal model of the software or component.

Question 6 Which of the following would be an example of decision table testing for a financial application, applied at the system-test level?

a. A table containing rules for combinations of inputs to two fields on a screen.

b. A table containing rules for interfaces between components.

c. A table containing rules for mortgage applications.

d. A table containing rules for basic arithmetic to two decimal places.

Question 7 Which of the following could be a coverage measure for state transition testing?

V All states have been reached.

W The response time for each transaction is adequate.

X Every transition has been exercised.

Y All boundaries have been exercised.

Z Specific sequences of transitions have been exercised.

a. X, Y and Z.

b. V, X, Y and Z.

c. W, X and Y.

d. V, X and Z.

Question 8 Postal rates for light letters are $0.25 up to 10g, $0.35 up to 50g plus an extra $0.10 for each additional 25g up to 100g.

Which test inputs (in grams) would be selected using equivalence partitioning (EP)?

a. 8, 42, 82, 102.

b. 4, 15, 65, 92, 159.

c. 10, 50, 75, 100.

d. 5, 20, 40, 60, 80.

Question 9 Which of the following could be used to assess the coverage achieved for black-box test techniques?

V Decision outcomes exercised.

W Partitions exercised.

X Boundaries exercised.

Y State transitions exercised.

Z Statements exercised.

a. V, W, Y or Z.

b. W, X or Y.

c. V, X or Z.

d. W, X, Y or Z.

Question 10 Which of the following would white-box test techniques be most likely to be applied to?

1) Boundaries between mortgage interest rate bands.
2) An invalid transition between two different arrears statuses.
3) The business process flow for mortgage approval.
4) Control flow of the program to calculate repayments.

a. 2, 3 and 4.

b. 2 and 4.

c. 3 and 4.

d. 1, 2 and 3.

Question 11 Use case testing is useful for which of the following?

P Designing acceptance tests with users or customers.

Q Making sure that the mainstream business processes are tested.

R Finding defects in the interaction between components.

S Identifying the maximum and minimum values for every input field.

T Identifying the percentage of statements exercised by a sets of tests.

a. P, Q and R.

b. Q, S and T.

c. P, Q and S.

d. R, S and T.

Question 12 Which of the following statements about the relationship between statement coverage and decision coverage is correct?

a. 100% decision coverage is achieved if statement coverage is greater than 90%.

b. 100% statement coverage is achieved if decision coverage is greater than 90%.

c. 100% decision coverage always means 100% statement coverage.

d. 100% statement coverage always means 100% decision coverage.

Question 13 Why are experience-based test techniques good to use?

a. They can find defects missed by black-box and white-box test techniques.

b. They do not require any training to be as effective as formal techniques.

c. They can be used most effectively when there are good specifications.

d. They will ensure that all of the code or system is tested.

Question 14 If you are flying with an economy ticket, there is a possibility that you may get upgraded to business class, especially if you hold a gold card in the airline's frequent flier program. If you do not hold a gold card, there is a possibility that you will get bumped off the flight if it is full and you check in late. This is shown in Figure 4.5. Note that each box (that is, statement) has been numbered.

Three tests have been run:

Test 1: Gold card holder who gets upgraded to business class.

Test 2: Non-gold card holder who stays in economy.

Test 3: A person who is bumped from the flight.

What is the statement coverage of these three tests?

a. 60%.

b. 70%.

c. 80%.

d. 90%.

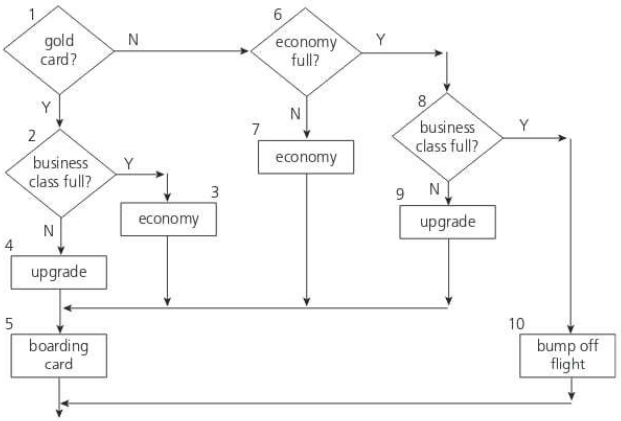


FIGURE 4.5 Control flow diagram for flight check-in

Question 15 Consider the control flow shown in Figure 4.6. In this example, if someone is a member, then they get a 10% discount, but only on items with an ItemCode of 25 or less. The following tests have already been run:

Test 1: Name is a member, ItemCode = 50

Test 2: Name is not a member, ItemCode = 27

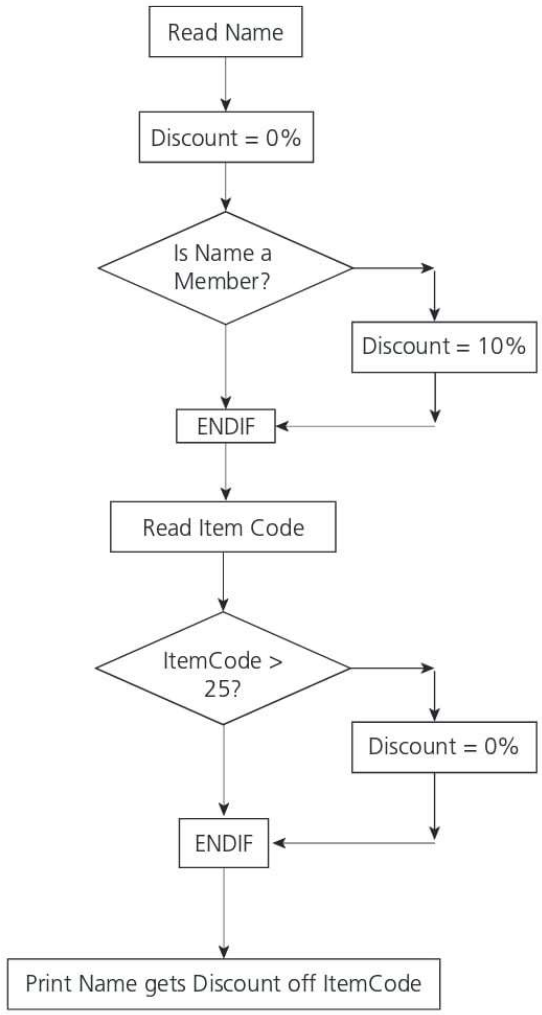


FIGURE 4.6 Control flow diagram for Question 15

What is the statement coverage and decision coverage of these tests?

a. Statement coverage = 100%, Decision coverage = 100%.

b. Statement coverage = 100%, Decision coverage = 50%.

c. Statement coverage = 75%, Decision coverage = 100%.

d. Statement coverage = 100%, Decision coverage = 75%.

Question 16 When choosing which technique to use in a given situation, which factors should be taken into account?

U Previous experience of types of defects found in this or similar systems.

V The existing knowledge of the testers.

W Regulatory standards that apply.

X The type of test execution tool that will be used.

Y The work products available.

Z Previous experience in the development language.

a. V, W, Y and Z.

b. U, V, W and Y.

c. U, X and Y.

d. V, W and Y.

Question 17 Given the state diagram in Figure 4.7, which test case is the minimum series of valid transitions to cover every state?

a. SS – S1 – S2 – S4 – S1 – S3 – ES.

b. SS – S1 – S2 – S3 – S4 – ES.

c. SS – S1 – S2 – S4 – S1 – S3 – S4 – S1 – S3 – ES.

d. SS – S1 – S4 – S2 – S1 – S3 – ES.

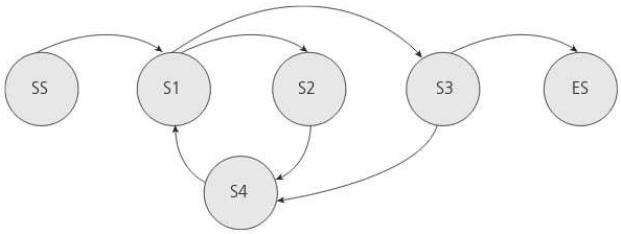


FIGURE 4.7 State diagram for PIN entry

EXERCISES

Exercises based on the techniques covered in this chapter are given in this section. Worked solutions are given in the next section.

Equivalence Partitioning/Boundary Value Analysis exercise

Scenario: If you take the train before 9:30 am or in the afternoon after 4:00 pm until 7:30 pm (the rush hour), you must pay full fare. A saver ticket is available for trains between 9:30 am and 4:00 pm, and after 7:30 pm.

What are the partitions and boundary values to test the train times for ticket types? Which are valid partitions and which are invalid partitions? What are the boundary values? (A table may be helpful to organize your partitions and boundaries.) Derive test cases for the partitions and boundaries.

Are there any questions you have about this requirement? Is anything unclear?

Decision table exercise

Scenario: If you hold an over 60s rail card, you get a 34% discount on whatever ticket you buy. If you are travelling with a child (under 16), you can get a 50% discount on any ticket if you hold a family rail card, otherwise you get a 10% discount. You can only hold one type of rail card.

Produce a decision table showing all the combinations of fare types and resulting discounts and derive test cases from the decision table.

State transition exercise

Scenario: A website shopping basket starts out as empty. As purchases are selected, they are added to the shopping basket. Items can also be removed from the shopping basket. When the customer decides to check out, a summary of the items in the basket and the total cost are shown, for the customer to say whether this is OK or not. If the contents and price are OK, then you leave the summary display and go to the payment system. Otherwise you go back to shopping (so you can remove items if you want).

a. Produce a state diagram showing the different states and transitions. Define a test, in terms of the sequence of states, to cover all transitions.

b. Produce a state table. Give an example test for an invalid transition.

Statement and decision testing exercise

Note that statement and decision testing are no longer K3, as they were in the previous Syllabus, but this exercise should help you to understand the technique a bit better anyway!

Scenario: A vending machine dispenses either hot or cold drinks. If you choose a hot drink (for example tea or coffee), it asks if you want milk (and adds milk if required), then it asks if you want sugar (and adds sugar if required), then your drink is dispensed.

a. Draw a control flow diagram for this example. (Hint: regard the selection of the type of drink as one statement.)

b. Given the following tests, what is the statement coverage achieved? What is the decision coverage achieved?

   Test 1: Cold drink.
   Test 2: Hot drink with milk and sugar.

c. What additional tests would be needed to achieve 100% statement coverage? What additional tests would be needed to achieve 100% decision coverage?

EXERCISE SOLUTIONS

EP/BVA exercise

The first thing to do is to establish exactly what the boundaries are between the full fare and saver fare. Let's put these in a table to organize our thoughts:

Scheduled departure time	≤ 9:29 am	9:30 am – 4:00 pm	4:01 pm – 7:30 pm	≥ 7:31 pm
Ticket type	full	saver	full	saver

We have assumed that the boundary values are: 9:29 am, 9:30 am, 4:00 pm, 4:01 pm, 7:30 pm and 7:31 pm. By setting out exactly what we think is meant by the specification, we may highlight some ambiguities or, at least, raise some questions. This is one of the benefits of using the technique! For example:

'When does the morning rush hour start? At midnight? At 11:30 pm the previous day? At the time of the first train of the day? If so, when is the first train? 5:00 am?'

This is a rather important omission from the specification. We could make an assumption about when it starts, but it would be better to find out what is correct.

- If a train is due to leave at exactly 4:00 pm, is a saver ticket still valid?
- What if a train is due to leave before 4:00 pm but is delayed until after 4:00 pm? Is a saver ticket still valid? (That is, if the actual departure time is different from the scheduled departure time.)

Our table above has helped us to see where the partitions are. All of the partitions in the table above are valid partitions. It may be that an invalid partition would be a time that no train was running, for example before 5:00 am, but our specification did not mention that! However it would be good to show this possibility also. We could be a bit more formal by listing all valid and invalid partitions and boundaries in a table, as we described in Section 4.3.1, but in this case it does not actually add a lot, since all partitions are valid.

Here are the test cases we can derive for this example:

Test case reference	**Input**	**Expected outcome**
1	Depart 4:30 am	Pay full fare
2	Depart 9:29 am	Pay full fare
3	Depart 9:30 am	Buy saver ticket
4	Depart 11:37 am	Buy saver ticket
5	Depart 4:00 pm	Buy saver ticket
6	Depart 4:01 pm	Pay full fare
7	Depart 5:55 pm	Pay full fare
8	Depart 7:30 pm	Pay full fare
9	Depart 7:31 pm	Buy saver ticket
10	Depart 10:05 pm	Buy saver ticket

Note that test cases 1, 4, 7 and 10 are based on equivalence partition values; test cases 2, 3, 5, 6, 8 and 9 are based on boundary values. There may also be other information about the test cases, such as preconditions, that we have not shown here.

Decision table exercise

The fare types mentioned are an over 60s rail card, a family rail card and whether you are travelling with a child or not. With three conditions or causes, we have eight columns in our decision table below.

Causes (inputs)	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**
over 60s rail card?	Y	Y	Y	Y	N	N	N	N
family rail card?	Y	Y	N	N	Y	Y	N	N
child also travelling?	Y	N	Y	N	Y	N	Y	N
Effects (outputs)								
Discount (%)	X/?/50%	X/?/34%	34%	34%	50%	0%	10%	0%

When we come to fill in the effects, we may find this a bit more difficult. For the first two rules, for example, what should the output be? Is it an X because holding more than one rail card should not be possible? The specification does not actually say what happens if someone does hold more than one card, that is, it has not specified the output, so perhaps we should put a question mark in this column. Of course, if someone does hold two rail cards, they probably would not admit this, and perhaps they would claim the 50% discount with their family rail card if they are travelling with a child, so perhaps we should put 50% for Rule 1 and 34% for Rule 2 in this column. Our notation shows that we do not know what the expected outcome should be for these rules!

This highlights the fact that our natural language (English) specification is not very clear as to what the effects should actually be. A strength of this technique is that it forces greater clarity. If the answers are spelled out in a decision table, then it is clear what the effect should be. When different people come up with different answers for the outputs, then you have an unclear specification!

The word 'otherwise' in the specification is ambiguous. Does 'otherwise' mean that you always get at least a 10% discount or does it mean that if you travel with a child and an over 60s card but not a family card you get 10% and 34%? Depending on what assumption you make for the meaning of 'otherwise', you will get a different last row in your decision table.

Note that the effect or output is the same (34%) for both Rules 3 and 4. This means that our third cause (whether or not a child is also travelling) actually has no influence on the output. These columns could therefore be combined with 'do not care' as the entry for the third cause. This rationalizing of the table means we will have fewer columns and therefore fewer test cases. The reduction in test cases is based on the assumption we are making about the factor having no effect on the outcome, so a more thorough approach would be to include each column in the table.

Here is a rationalized table, where we have shown our assumptions about the first two outcomes and we have also combined Rules 6 and 8 above, since having a family rail card has no effect if you are not travelling with a child.

Causes (inputs)	**R1**	**R2**	**R3**	**R5**	**R6**	**R7**
over 60s rail card?	Y	Y	Y	N	N	N
family rail card?	Y	Y	N	Y	–	N
child also travelling?	Y	N	–	Y	N	Y
Effects (outputs)						
Discount (%)	50%	34%	34%	50%	0%	10%

Here are the test cases that we derive from this table. (If you did not rationalize (or collapse) the table, then you will have eight test cases rather than six.) Note that you would not necessarily test each column, but the table enables you to make a decision about which combinations to test and which not to test this time.

Test case reference	Input	Expected outcome
1	S. Wilkes, with over 60s rail card and family rail card, travelling with grandson Josh (age 11)	50% discount for both tickets
2	Mrs M. Davis, with over 60s rail card and family rail card, travelling alone	34% discount
3	J. Rogers, with over 60s rail card, travelling with his wife	34% discount (for J. Rogers only, not his wife)
4	S. Gray, with family rail card, travelling with her daughter Betsy	50% discount for both tickets
5	Miss Congeniality, no rail card, travelling alone	No discount
6	Joe Bloggs with no rail card, travelling with his 5-year-old niece	10% discount for both tickets

Note that we may have raised some additional issues when we designed the test cases. For example, does the discount for a rail card apply only to the traveller or to someone travelling with them? Here we have assumed that it applies to all travellers for the family rail card, but to the individual passenger only for the over 60s rail card.

State transition exercise

The state diagram is shown in Figure 4.8. The initial state (S1) is when the shopping basket is empty. When an item is added to the basket, it goes to state (S2), where there are potential purchases. Any additional items added to the basket do not change the state (just the total number of things to purchase). Items can be removed, which does not change the state unless the total items ordered goes from 1 to 0. In this case, we go back to the empty

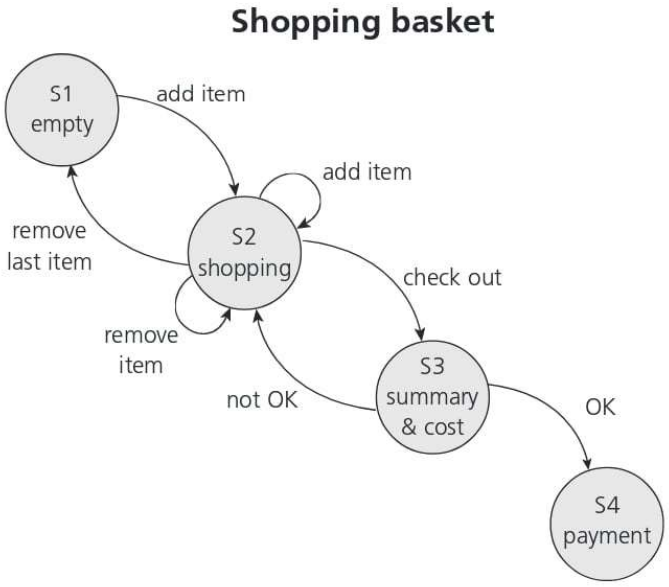


FIGURE 4.8 State diagram for shopping basket

basket (S1). When we want to check out, we go to the summary state (S3) for approval. If the list and prices are approved, we go to payment (S4); if not, we go back to the shopping state (possibly to remove some items to reduce the total price we have to pay). There are four states and seven transitions.

Note that S1 is our start state for this example and S4 is the end state – this means that we are not concerned with any event that happens once we get to State S4.

Here is a test to cover all transitions. Note that the end state from one step or event is the start state for the next event, so these steps must be done in this sequence.

State	Event (action)
S1	Add item
S2	Remove (last) item
S1	Add item
S2	Add item
S2	Remove item
S2	Check out
S3	Not OK
S2	Check out
S3	OK
S4	Payment

Although our example is not interested in what happens from State 4, there would be other events and actions once we enter the payment process that could be shown by another state diagram (for example check validity of the credit card, deduct the amount, email a receipt, etc.).

The corresponding state table is:

State or event	Add item	Remove item	Remove last item	Check out	Not OK	OK
S1 Empty	S2	–	–	–	–	–
S2 Shopping	S2	S2	S1	S3	–	–
S3 Summary	–	–	–	–	S2	S4
S4 Payment	–	–	–	–	–	–

All of the boxes that contain '–' (dash) are invalid transitions in this example. Example negative tests would include:

- Attempt to add an item from the summary and cost state (S3).
- Try to remove an item from the empty shopping basket (S1).
- Try to enter OK while in the shopping state (S2).

Statement and decision testing exercise

The control flow diagram is shown in Figure 4.9. Note that drawing a control diagram here illustrates that white-box testing can also be applied to the structure of general processes, not just to computer algorithms. Flowcharts are generally easier to understand than text when you are trying to describe the results of decisions taken on later events.

On Figure 4.10, we can see the route that Tests 1 and 2 have taken through our control flow graph. Test 1 has gone straight down the left-hand side to select a cold drink. Test 2 has gone to the right at each opportunity, adding both milk and sugar to a hot drink.

Every statement (represented by a box on the diagram) has been covered by our two tests, so we have 100% statement coverage.

We have not taken the No Exit from either the 'milk?' or 'sugar?' decisions, so there are two decision outcomes that we have not tested yet. We did test both of the outcomes from the 'hot or cold?' decision, so we have covered four out of six decision outcomes. Decision coverage is 4/6 or 67% with the two tests.

No additional tests are needed to achieve statement coverage, as we already have 100% coverage of the statements.

One additional test is needed to achieve 100% decision coverage:

Test 3: Hot drink, no milk, no sugar.

This test will cover both of the 'No' decision outcomes from the milk and sugar decisions, so we will now have 100% decision coverage.

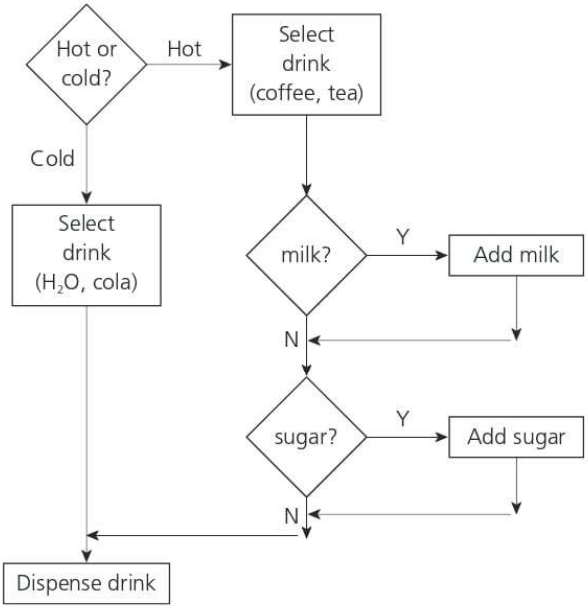


FIGURE 4.9 Control flow diagram for drinks dispenser

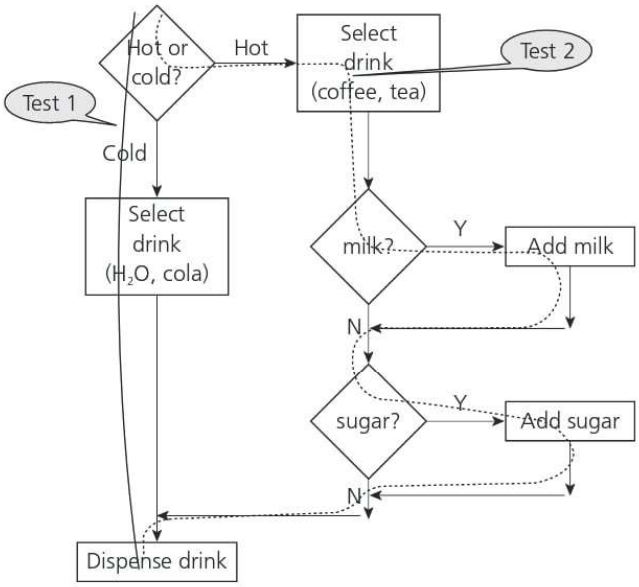


FIGURE 4.10 Control flow diagram showing coverage of tests

CHAPTER FIVE

Test management

Testing is a complex activity. It can be a distinct sub-project within the larger software development, maintenance or integration project. It usually accounts for a substantial proportion of the overall project budget. Therefore we must understand how we should manage the testing we do.

In this chapter, we cover essential topics for test management in six sections. The first relates to how to organize the testers and the testing. The second concerns the estimation, planning and strategizing of the test effort. The third addresses test progress monitoring, test reporting and test control. The fourth explains configuration management and its relationship to testing. The fifth covers the central topic of risk and how testing affects and is affected by product and project risks. The sixth and final section discusses the management of defects.

5.1 TEST ORGANIZATION

SYLLABUS LEARNING OBJECTIVES FOR 5.1 TEST ORGANIZATION (K2)

FL-5.1.1 Explain the benefits and drawbacks of independent testing (K2)

FL-5.1.2 Identify the tasks of a test manager and tester (K1)

In this section, We'll talk about organizing a test effort within a project. We'll look at the value of independent testing, and discuss the potential benefits and risks associated with independent testing. We'll examine the various types of different team members we might want on a test team. And we'll familiarize ourselves with the typical tasks performed by test managers and testers.

As we go through this section, keep your eyes open for the Glossary terms **tester**, and **test manager**.

5.1.1 Independent testing

In Chapter 1 we talked about independent testing from the perspective of individual tester psychology. In this chapter, we'll look at the organizational and managerial implications of independence.

Testing tasks may be done by people with a specific testing role, for example with tester as part of their job title, but testers are definitely not the only people who do testing. Developers, business analysts, users and customers also do testing tasks for

different reasons and at different times. Even those who are full-time testers may do a variety of different tasks at different times. But if lots of people do testing, why have people dedicated to it? One reason is that the view of a different person, especially one who is trained to look for problems, can be much more effective at finding those problems. Many organizations, especially in safety-critical areas, have separate teams of independent testers to capitalize on this effect. As we saw in Chapter 1 Section 1.5, independence can overcome cognitive bias.

If there is a separate test team, approaches to organizing it vary, as do the places in the organizational structure where the test team fits. Since testing is an assessment of quality, and since that assessment may not always be perceived as positive, many organizations strive to create an organizational climate where testers can deliver an independent, objective assessment of quality.

Levels of independence

When thinking about how independent the test team is, recognize that independence is not an either/or condition, but a continuum.

At one end of the continuum lies the absence of independence, where the developer performs testing on their own code within the development team. Of course, every good developer does do some testing of their own code, but this should not be the ONLY testing!

Moving toward independence, you find an integrated tester or group of testers working alongside the developers, but still within and reporting to the development manager. For example, developers may test each other's code after testing their own, or a tester on an Agile team may help developers and do some independent testing within the team. Pair programming is one way of having another pair of eyes on the code as it is being developed, whether it is two developers pairing, or a developer and a tester.

A further level of independence would be to have a team of testers who are independent and outside the development team, reporting to project management or business management.

Sometimes there are testers or teams of testers with special skills or responsibilities within an organization; such specialists may also be outside the development organization, which would be the other end of the independence continuum. For example, there may be a tester or team that specializes in performance or security testing.

In fully independent testing, you might see a separate test team reporting into the organization at a point equal to the development or project team or at a higher level. You might find specialists in the business domain (such as users of the system), specialists in technology (such as database experts), and specialists in testing (such as security testers or performance test experts) in a separate test team, as part of a larger independent test team, or as part of a contracted outsourced test team.

Potential benefits of independence

Let's examine the potential benefits and risks of independence, starting with the benefits.

An independent tester can often see more, different defects than a tester working within a development team – or a tester who is by profession a developer. While business analysts, marketing staff, designers and developers bring their own assumptions to the specification and implementation of the item under test, an independent tester brings a different set of assumptions to testing and to reviews, which often helps expose hidden defects and problems related to the group's way of thinking,

as we discussed in Chapter 3. An independent tester brings a sceptical attitude of professional pessimism, a sense that, if there's any doubt about the observed behaviour, they should ask: 'Is this a defect?'

At the team level, an independent test team reporting to a senior or executive manager may enjoy (once they earn it) more credibility in the organization than a test manager or tester who is part of the development team. An independent tester who reports to senior management may be able to report his or her results honestly and without concern for reprisals that might result from pointing out problems in co-workers' or, worse yet, the manager's work. An independent test team often has a separate budget, which helps ensure the proper level of money is spent on tester training, testing tools, test equipment and so forth. In addition, in some organizations, testers in an independent test team may find it easier to have a career path that leads up into more senior roles in testing.

Potential drawbacks of test independence

Independent test teams are not risk-free. It is possible for the testers and the test team to become isolated. This can take the form of interpersonal isolation from the developers, the designers, and the project team itself. It can also take the form of isolation from the broader view of quality and the business objectives, for example, an obsessive focus on defects, often accompanied by a refusal to accept business prioritization of defects. This leads to communication problems, feelings of alienation and antipathy, a lack of identification with and support for the project goals, spontaneous blame festivals and political backstabbing.

Even well-integrated test teams can suffer problems. Other project stakeholders might come to see the independent test team (rightly or wrongly) as a bottleneck and a source of delay.

Some developers abdicate their responsibility for quality, saying, 'Well, we have this test team now, so why do I need to unit test my code?'

Independent testers may not have all of the information that they need about the test object, since they may be outside the development organization itself (where often much information is communicated informally). This leads to them being less effective than they should or could be.

Independence is not a replacement for familiarity. Although an independent view sees things that those closer to it miss, those who know the code or the test object best may be able to see some things that the independent tester would miss because of their limited knowledge. It is not a case of independence is always best, but of getting the best balance between independence and familiarity.

Independence varies

Due to a desire for the benefits of an independent test team, companies sometimes establish them, only to break them up again later. Why does that happen? A common cause is the failure of the test manager to effectively manage the risks of independence listed above. Some test teams succumb to the temptation to adopt a 'No can do' attitude, coming up with reasons why the project should bend to their needs rather than each side being flexible so as to enable project success. Testers take to acting as enforcers of process or as auditors, without a proper management mandate and support. Resentments and pressures build, until at last the organization decides that the independent test team causes more problems than it solves. It is especially important for testers and test managers to understand the mission they serve and the reasons why the organization wants an independent test team. Often, the entire test

team must realize that, whether they are part of the project team or independent, they exist to provide a service to the project team.

There is no one right approach to organizing testing. For each project, you must consider whether to use an independent test team, based on the project, the application domain and the levels of risk, among other factors. As the size, complexity and criticality of the project increases, it is important to have independence in later levels of testing (like integration test, system test and acceptance test), though some testing is often best done by other people such as project managers, quality managers, developers, business and domain experts or infrastructure or IT operations experts.

In most projects, there will be multiple test levels; the amount of independence in testing varies between levels as well (as it should). Often more independence is most effective at the higher levels, for example system and user acceptance testing.

The type of development life cycle also influences the level of independence of testing. In Agile development, a tester may provide some independence as part of the development team and may (also) be part of an independent team performing independent testing at higher levels. Product owners may perform acceptance testing to validate user stories at the end of each iteration.

5.1.2 Tasks of a test manager and tester

We have seen that the location of testers or of a test team within a project organization can vary widely. Similarly, there is wide variation in the roles that people within testing play. Some of these roles occur frequently, some infrequently. Two roles that are found within many organizations are those of the **test manager** and the **tester**, though the same people may play both roles at various points during the project. The activities and tasks performed by these two roles will vary, depending on the organization, the project and product context, and the skills of the individuals. Let's take a look at the work typically done in these roles, starting with the test manager.

Test manager The person responsible for project management of testing activities and resources, and evaluation of a test object. The individual who directs, controls, administers, plans and regulates the evaluation of a test object.

Tester A skilled professional who is involved in the testing of a component or system.

Test manager tasks

A test manager tends to be the person tasked with overall responsibility for the test process and successful leadership of the test activities. The test manager role may be performed by someone who holds this as their full-time position (a professional test manager), or it might be done by a quality assurance manager, project manager or a development manager. Regarding the last two people on this list, warning bells about independence should be ringing in your head now, in addition to thoughts about how we can ensure that such non-testers gain the knowledge and outlook needed to manage testing. The test manager tasks may also be done by a senior tester. Whoever is playing the role, expect them to plan, monitor and control the testing work.

A test manager may be a single person trying to ensure that all testing is done as well as it can be, or a test manager may have one or more teams of people reporting to them (for example in larger organizations). Sometimes the person at the top of the hierarchy would be called a test coordinator or test coach, and the individual teams may be led by a test leader or lead tester. In any case, the test manager's role is to manage the testing!

Typical tasks of the test manager role may include the following:

- At the outset of the project, in collaboration with the other stakeholders, devise the test objectives, organizational test policies (if not already in place), and test strategies.

- Plan the test activities, based on the test objectives and risks, and the context of the organization and the project. This may involve selecting the test approaches, estimating time, effort and cost for testing, acquiring resources, defining test levels, types and test cycles and planning defect management.
- Write and update over time any test plan(s).
- Coordinate the test plan(s) with other project stakeholders, project managers, product owners and anyone else who may affect or be affected by the project or the testing.
- Share the testing perspective with other project activities, such as integration planning, especially where third-party suppliers are involved.
- Lead, guide and monitor the analysis, design, implementation and execution of the tests, monitor test progress and results, and check the status of exit criteria (or definition of done).
- Prepare and deliver test progress reports and test summary reports, based on information gathered from the testers.
- Adapt the test planning based on test results and progress (whether documented in test progress or summary reports or not) and take any actions necessary for test control.
- Support setting up the defect management system and adequate configuration management of the testware, and traceability of the tests to the test basis.
- Produce suitable metrics for measuring test progress and evaluating the quality of the testing and the product (test object).
- Recognize when test automation is appropriate and, if it is, plan and support the selection and implementation of tools to support the test process, including setting a budget for tool selection (and possible purchase, lease and support and training of the team), allocating time and effort for pilot projects and providing continuing support in the use of the tool(s). (See Chapter 6 for more on tool support for testing.)
- Decide about the implementation of test environment(s) and ensure that they are put into place before test execution and managed during test execution.
- Promote and advocate the testers, the test team and the test profession within the organization.
- Develop the skills and careers of testers, through training, performance evaluations, coaching and other activities, such as lunch-time discussions or presentations.

Although this is a list of typical activities of a test manager, the tasks may be carried out by people who are not labelled as a test manager. For example, in Agile development, the Agile team may perform some of these tasks, especially those to do with day-to-day testing within the team. Test managers who are outside an individual development team, who work with several teams or the whole organization, may be called a test coach. See Black [2009] for more on managing the test process.

Tester tasks

As with test managers, projects should include testers at the outset. In the planning and preparation of the testing, testers should review and contribute to test plans, as well as analyzing, reviewing and assessing requirements and design specifications.

They may be involved in or even be the primary people identifying test conditions and creating test designs, test cases, test procedure specifications and test data, and may automate or help to automate the tests. They often set up the test environments or assist system administration and network management staff in doing so.

In sequential life cycles, as test execution begins, the number of testers often increases, starting with the work required to implement tests in the test environment. They may play such a role on all test levels, even those not under the direct control of the test group; for example they might implement unit tests which were designed by developers. Testers execute and log the tests, evaluate the results and document problems found. They monitor the testing and the test environment, often using tools for this task, and often gather performance metrics. Throughout the testing life cycle, they review each other's work, including test specifications, defect reports and test results.

In Agile development, testers are involved in every iteration or sprint, performing a wide variety of testing tasks on whatever is being developed at the time, so they are continuously involved throughout. They may be reviewing user stories and identifying test conditions, implementing the tests, running them and reporting on results.

Testers may have specializations such as test analysis, test design, specific test types (especially non-functional testing), or test automation. Such specialists may take the role of tester at different test levels and at different times. For example, the person doing the testing tasks at component or component integration level is often the developer (but see caveats about independence in Section 5.1.1). At system testing or system integration testing, an independent test team may do the testing activities. In acceptance testing, the test tasks may be done by business analysts, subject matter experts, users or product owners. At operational acceptance testing, operations and/or system administration staff may be performing the tester tasks.

Typical tasks of the tester may include the following:

- Reviewing and contributing to test plans from the tester perspective.
- Analyzing, reviewing and assessing requirements, user stories and acceptance criteria, specifications and models (that is, the test basis) for testability and to detect defects early.
- Identifying and documenting test conditions and test cases, capturing traceability between test cases, test conditions and the test basis to assist in checking the thoroughness of testing (coverage), the impact of failed tests and the impact on the tests of changes in the test basis.
- Designing, setting up and verifying test environments(s), coordinating with system administration and network management.
- Designing and implementing test cases and test procedures, including automated tests where appropriate.
- Acquiring and preparing test data to be used in the tests.
- Creating a detailed test execution schedule (for manual tests).
- Executing the tests, evaluating the results and documenting deviations from expected results as defect reports.
- Using appropriate tools to help the test process.
- Automating tests as needed (for technical test specialists), as supported by a test automation engineer or expert or a developer. (See Chapter 6 for more on test automation.)

- Evaluating non-functional characteristics such as performance efficiency, reliability, usability, security, compatibility and portability.
- Reviewing tests developed by others, including other testers, business analysts, developers or product owners. Part of a tester's role is to help educate others about doing better testing.

Defining the skills test staff need

This section is outside what is required by the Syllabus, but we include it as useful and practical advice anyway!

Doing testing properly requires more than defining the right positions and number of people for those positions. Good test teams have the right mix of skills based on the tasks and activities they need to carry out, and people outside the test team who are in charge of test tasks need the right skills, too.

People involved in testing need basic professional and social qualifications such as literacy, the ability to prepare and deliver written and verbal reports, the ability to communicate effectively and so on. Going beyond that, when we think of the skills that testers need, three main areas come to mind:

- **Application or business domain:** A tester must understand the intended behaviour, the problem the system will solve, the process it will automate and so forth, in order to spot improper behaviour while testing, and recognize the 'must-work' functions and features.
- **Technology:** A tester must be aware of issues, limitations and capabilities of the chosen implementation technology, in order to effectively and efficiently locate problems and recognize the 'likely-to-fail' functions and features.
- **Testing:** A tester must know the testing topics discussed in this book, and often more advanced testing topics, in order to effectively and efficiently carry out the test tasks assigned.

The specific skills in each area and the level of skill required vary by project, organization, application and the risks involved.

The set of testing tasks and activities are many and varied, and so too are the skills required, so we often see specialization of skills and separation of roles. For example, due to the special knowledge required in the areas of testing, technology and business domain, respectively, test automation experts may handle automating the regression tests, developers may perform component and integration tests and users and operators may be involved in acceptance tests.

We have long advocated pervasive testing, the involvement of people throughout the project team in carrying out testing tasks. Let's close this section, though, on a cautionary note. Software and system companies (for example producers of shrink-wrapped software and consumer products) typically overestimate the technology knowledge required to be an effective tester. Businesses that use information technology (for example banks and insurance companies) typically overestimate the business domain knowledge needed.

All types of projects tend to underestimate the testing knowledge required. We have seen a project fail in part because people without proper testing skills tested critical components, leading to the disastrous discovery of fundamental architectural problems later. Most projects can benefit from the participation of professional testers, as amateur testing alone will usually not suffice.

5.2 TEST PLANNING AND ESTIMATION

SYLLABUS LEARNING OBJECTIVES FOR 5.2 TEST PLANNING AND ESTIMATION (K3)

FL-5.2.1 **Summarize the purpose and content of a test plan (K2)**

FL-5.2.2 **Differentiate between various test strategies (K2)**

FL-5.2.3 **Give examples of potential entry and exit criteria (K2)**

FL-5.2.4 **Apply knowledge of prioritization, and technical and logical dependencies, to schedule test execution for a given set of test cases (K3)**

FL-5.2.5 **Identify factors that influence the effort related to testing (K1)**

FL-5.2.6 **Explain the difference between two estimation techniques: the metrics-based technique and the expert-based technique (K2)**

In this section, we'll talk about a complicated trio of test topics: plans, estimates and strategies. Plans, estimates and strategies depend on a number of factors, including the level, targets and objectives of the testing we are setting out to do. Writing a plan, preparing an estimate and selecting test strategies tend to happen concurrently and ideally during the planning period for the overall project, though we must be ready to revise them as the project proceeds and we gain more information.

Let's look closely at how to prepare a test plan, examining issues related to planning for a project, for a test level, for a specific test type and for test execution. We'll discuss selecting test strategies and ways to establish adequate exit criteria for testing. In addition, we'll look at various tasks related to test execution that need planning. We'll examine typical factors that influence the effort related to testing and see two different estimation techniques: metrics-based and expert-based.

Look out for the Glossary terms **entry criteria**, **exit criteria**, **test approach**, **test estimation**, **test plan**, **test planning** and **test strategy** in this section.

5.2.1 The purpose and content of test plan

Test plan
Documentation describing the test objectives to be achieved and the means and the schedule for achieving them, organized to coordinate testing activities.

While people tend to have different definitions of what goes in a **test plan**, for us a test plan is the project plan for the testing work to be done. It is not a test design specification, a collection of test cases or a set of test procedures; in fact, most of our test plans do not address that level of detail.

Why do we write test plans? We have three main reasons: to guide our thinking, to communicate to others and to help manage future changes.

First, writing a test plan guides our thinking. We find that if we can explain something in words, we understand it. If we cannot explain it, there's a good chance we don't understand. Writing a test plan forces us to confront the challenges that await us and focus our thinking on important topics. In Chapter 2 of Fred Brooks' brilliant

and essential book on software engineering management, *The Mythical Man-Month*, he explains the importance of careful estimation and planning for testing as follows:

> *Failure to allow enough time for system test, in particular, is peculiarly disastrous. Since the delay comes at the end of the schedule, no one is aware of schedule trouble until almost the delivery date [and] delay at this point has unusually severe … financial repercussions. The project is fully staffed, and cost-per-day is maximum [as are the associated opportunity costs]. It is therefore very important to allow enough system test time in the original schedule.*
>
> Brooks [1995]

This is particularly applicable to sequential development life cycles (as many were back in 1995). Agile development was designed to address some of these problems, but it does not obviate the need for system testing, so the advice is still relevant.

We find that using a template when writing test plans helps us remember the important challenges. You can use the template and examples shown in ISO/IEC/IEEE 29119-3 [2013], use someone else's template or create your own template over time.

Test planning The activity of establishing or updating a test plan.

Second, the **test planning** process and the plan itself serve as vehicles for communicating with other members of the project team, testers, peers, managers and other stakeholders. This communication allows the test plan to influence the project team and allows the project team to influence the test plan. This is especially important in the areas of organization-wide testing policies and motivations; test scope, objectives, and critical areas to test; project and product risks, resource considerations and constraints; and the testability of the item under test.

You can accomplish this communication through circulation of one or two test plan drafts and through review meetings. Such a draft may include many notes such as the following examples:

> *[To Be Determined: Jennifer: Please tell me what the plan is for releasing the test items into the test lab for each cycle of system test execution?]*
>
> *[Dave – please let me know which version of the test tool will be used for the regression tests of the previous increments.]*

As you document the answers to these kinds of questions, the test plan becomes a record of previous discussions and agreements between the testers and the rest of the project team. Thus test planning is an ongoing activity performed throughout the product's life cycle (and beyond into maintenance).

Third, the test plan helps us manage change. During early stages of the project, as we gather more information, we revise our plans. As the project evolves and situations change, we adapt our plans. Written test plans give us a baseline against which to measure such revisions and changes. Furthermore, updating the plan at major milestones helps keep testing aligned with project needs. As we run the tests, we make final adjustments to our plans based on the results. You might not have the time – or the energy – to update your test plans every time a variance occurs, as some projects can be quite dynamic. A simple approach is described in Black [2009] Chapter 6, for documenting variances from the test plan. You can implement it using a database or spreadsheet. You can include these change records in a periodic test plan update, as part of a test status report, or as part as an end of project test summary.

We have found that it is better to write multiple test plans in some situations. For example, when we manage both integration and system test levels, those two test execution periods may occur at different points in time and will have different objectives.

For some systems projects, a hardware test plan and a software test plan will address different techniques and tools, as well as different audiences. However, since there might be overlap between these test plans, a master test plan that addresses the common elements can reduce the amount of redundant documentation.

What to do with your brain while planning tests

Writing a good test plan is easier than writing a novel, but both tasks require an organized approach and careful thought. In fact, since a good test plan is kept short and focused, unlike some novels, some might argue that it is harder to write a good test plan. Let's look at some of the planning tasks you need to carry out.

At a high level, you need to consider the purpose served by the testing work. In terms of the overall organizational needs, this purpose is referred to variously as the test team's mission or the organization's testing policy. A test plan is influenced by many factors; as well as test policy and the organization's test strategy, the development life cycle and methods being used for development, and the availability of resources. In terms of the specific project, understanding the purpose of testing means knowing the answers to questions such as:

- What is in scope and what is out of scope for this testing effort?
- What are the test objectives?
- What are the important project and product risks? (More on risks in Section 5.5.)
- What is the overall approach of testing in this project?
- How will test activities be integrated and coordinated into the software life cycle activities?
- How do we decide what to test, what people and other resources are needed to perform test activities, and how test activities will be carried out?
- What constraints affect testing (for example budget limitations, hard deadlines, etc.)?
- What is most critical for this product and project?
- Which aspects of the product are more (or less) testable?
- What should be the overall test execution schedule and how should we decide the order in which to do test analysis, test design, implementation, execution and evaluation of specific tests, either on specific dates or in the context of an iteration? (Product and planning risks, discussed later in this chapter, will influence the answers to these questions.)
- What metrics will be used for test monitoring and control and how will they be gathered and analyzed?
- What is the budget for all test activities?
- What should be the level of detail and structure for test documentation? (Templates or example documents or work products are helpful for this.)

You should then select strategies which are appropriate to the purpose of testing (more on the topic of selecting strategies in Section 5.2.2).

In addition, you need to decide how to split the testing work into various levels, as discussed in Chapter 2 (for example component, integration, system and acceptance). If that decision has already been made, you need to decide how to best fit your testing work in the level you are responsible for with the testing work done in those other test levels. During the analysis and design of tests, you will want to reduce gaps and overlap between levels and, during test execution, you will want to coordinate between

the levels. Such details dealing with inter-level coordination are often addressed in the master test plan.

In addition to integrating and coordinating between test levels, you should also plan to integrate and coordinate all the testing work to be done with the rest of the project. For example, what items must be acquired for the testing? Are there ongoing supply issues, such as with imitation bank notes (that is, simulated bank notes) for a financial application such as an ATM? When will the developers complete work on the component or system under test? What operations support is required for the test environment? What kind of information must be delivered to the maintenance team at the end of testing?

Moving down into the details, what makes a plan a plan (rather than a statement of principles, a laundry list of good ideas or a collection of suggestions) is that the author specifies in it who will do what when and (at least in a general way) how. Resources are required to carry out the work. There are often hard decisions that require careful consideration and building a consensus across the team, including with the project manager.

The entire testing process, from planning through to completion, produces information, some of which you will need to document. How precisely should testers write the test designs, cases and procedures? How much should they leave to the judgement of the tester during test execution, and what are the reproducibility issues associated with this decision? What kinds of templates can testers use for the various documents they will produce? How do those documents relate to one another? If you intend to use tools for tasks such as test design or execution, as discussed in Chapter 6, you will need to understand how the models or automated tests will integrate with manual testing, and plan who will be responsible for automation design, implementation and support. There may be a separate test automation plan, but this needs to be coordinated with other test plans.

Some information you will need to gather in the form of raw data and then distil. What metrics to do you intend to use to monitor, control and manage the testing? Which of those metrics, and perhaps other metrics, will you use to report your results? We'll look more closely at possible answers to those questions in Section 5.3, but a good test plan provides answers early in the project.

Test strategy (organizational test strategy) Documentation that expresses the generic requirements for testing one or more projects run within an organization, providing detail on how testing is to be performed, and is aligned with the test policy.

Test approach The implementation of the test strategy for a specific project.

5.2.2 Test strategy and test approach

A **test strategy** is the general way in which testing will happen within each of the levels of testing, independent of project, across the organization. The **test approach** is the name the ISTQB gives to the implementation of the test strategy on a specific project. Since the test approach is specific to a project, you should define and document the approach in the test plans, refining and providing further detail in the test designs.

Deciding on the test approach involves careful consideration of the testing objectives, the project's goals and overall risk assessment. These decisions provide the starting point for planning the test process, for selecting the test design techniques and test types to be applied, and for defining the entry and exit criteria. In your decision-making on the approach, you should take into account the project, product and organization context, issues related to risks, hazards and safety, the available resources, the team's level of skills, the technology involved, the nature of the system under test, considerations related to whether the system is custom-built or assembled from commercial off-the-shelf components (COTS), the organization's test objectives and any applicable regulations.

The choice of test approaches or strategies is one powerful factor in the success of the test effort and the accuracy of the test plans and estimates. This factor is under the control of the testers and test managers. Of course, having choices also means that you can make mistakes, so we'll go into more detail about how to pick the right test strategies in a minute. First, though, let's survey the major types of test strategies that are commonly found.[1]

- **Analytical:** In this strategy, tests are determined by analyzing some factor, such as requirements (or other test basis) or risk. For example, the risk-based strategy involves performing a risk analysis using project documents and stakeholder input, then planning, estimating, designing and prioritizing the tests based on risk. We will talk more about risk analysis later in this chapter. Another analytical test strategy is the requirements-based strategy, where an analysis of the requirements specification forms the basis for planning, estimating and designing tests. Analytical test strategies have in common the use of some formal or informal analytical technique, usually during the requirements and design stages of the project.
- **Model-based:** In this strategy, tests are designed based on some model of the test object. For example, you can build mathematical models for loading and response for e-commerce servers, and test based on that model. If the behaviour of the system under test conforms to that predicted by the model, the system is deemed to be working. Model-based test strategies have in common the creation or selection of some formal or informal model for critical system behaviours, usually during the requirements and design stages of the project. Examples also include business process models, state models, for example state transition diagrams, etc., or reliability grown models.
- **Methodical:** In this strategy, a pre-defined and fairly stable list of test conditions is used. For example, you might have a checklist that you have put together over the years that suggests the major areas that testing should cover, or you might follow an industry standard for software quality, such as ISO/IEC 25010 [2011], for your outline of major test areas. You then methodically design, implement and execute tests following this outline. Methodical test strategies have in common the adherence to a pre-planned, systematized approach. This may have been developed in-house, assembled from various concepts developed in-house and gathered from outside, or adapted significantly from outside ideas, and may have an early or late point of involvement for testing. Some examples include: working through a list (that is, taxonomy) of typical defects, or working through a list of desired quality characteristics, such as company-wide look-and-feel standards for websites and mobile apps.
- **Process- or standard-compliant:** In this strategy, an external standard or set of rules is used to analyze, design and implement tests. For example, you might adopt the standard ISO/IEC/IEEE 29119-3 [2013] for your testing, or you may use TMMi (Test Maturity Model integration, www.tmmi.org) to assess your current testing and improve it. More information about TMMi can be found in van Veenendaal and Wells [2012]. Alternatively, process- or standard-compliant strategies have in common reliance upon an externally developed approach to

[1] The catalogue of testing strategies that has been included in the ISTQB Foundation Syllabus grew out of an email discussion between Rex Black, Ross Collard, Kathy Iberle and Cem Kaner. We thank them for their thought provoking comments.

testing, often with little (if any) customization. They may have an early or late point of involvement for testing. For example, some organizations need to conform to safety-critical or financial regulations which may include processes for identifying tests in a rigorous way from the test basis.

- **Directed (or consultative):** In this strategy, stakeholders or experts (technology or business domain experts) may direct the testing according to their advice and guidance. For example, you might ask the users or developers of the system to tell you what to test or even rely on them to do the testing. Consultative or directed strategies have in common the reliance on a group of non-testers to guide or perform the testing effort, and typically emphasize the later stages of testing simply due to the lack of recognition of the value of early testing. This strategy may be helpful when the developing organization is a new start-up without a lot of testing knowledge or expertise. For example, an outside expert may advise on security testing.
- **Regression-averse:** In this strategy, the most important factor is to ensure that the system's performance does not deteriorate or get worse when it is changed and enhanced. To protect existing functionality, automated regression tests would be extensively used, as well as standard test suites and reuse of existing tests and test data. For example, you might try to automate all the tests of system functionality so that, whenever anything changes, you can re-run every test to ensure nothing has broken. Regression-averse strategies have in common a set of procedures – usually automated – that allow them to detect regression defects. A regression-averse strategy may involve automating functional tests prior to release of the function, in which case it requires early testing, but sometimes the testing is almost entirely focused on testing functions that have already been released, which is in some sense a form of post-release test involvement.
- **Reactive (or dynamic):** In this strategy, the tests react and evolve based on what is found while test execution occurs, rather than being designed and implemented before test execution starts. For example, you might create a lightweight set of testing guidelines that focus on rapid adaptation or known weaknesses in software. Reactive strategies, using exploratory testing, have in common concentrating on finding as many defects as possible during test execution and adapting to the realities of the system under test as it is when delivered, and they typically emphasize the later stages of testing. See, for example, the attack-based approach of Whittaker [2002] and Whittaker and Thompson [2003] and the exploratory approach of Kaner, Bach and Petticord [2002].

Some of these strategies are more preventive, others more dynamic. For example, analytical test strategies involve up-front analysis of the test basis and tend to identify problems in the test basis prior to test execution. This allows the early, cheap removal of defects. That is a strength of preventive approaches.

Reactive test strategies focus on the test execution period. Such strategies allow the location of defects and defect clusters that might have been hard to anticipate until you have the actual system in front of you. That is a strength of reactive approaches.

Rather than see the choice of strategies, particularly the preventive or reactive strategies, as an either/or situation, we'll let you in on the worst kept secret of testing (and many other disciplines). There is no one best way. We suggest that you adopt whatever test strategies make the most sense for your particular test approach, and feel free to borrow and blend.

How do you know which strategies to pick or blend for the best chance of success? There are many factors to consider, but let's highlight a few of the most important:

- **Risks:** Testing is about risk management, so consider the risks and the level of risk. For a well-established application that is evolving slowly, regression is an important risk, so regression-averse strategies make sense. For a new application, a risk analysis may reveal different risks if you pick a risk-based analytical strategy.
- **Skills:** Strategies must not only be chosen, they must also be executed. So you have to consider the skills that your testers possess and lack. A standard-compliant strategy is a smart choice when you lack the time and skills in your team to create your own approach.
- **Objectives:** Testing must satisfy the needs of stakeholders to be successful. If the objective is to find as many defects as possible with a minimal amount of up-front time and effort invested – for example, at a typical independent test lab – then a dynamic or reactive strategy makes sense.
- **Regulations:** Sometimes you must satisfy not only stakeholders, but also regulators. In this case, you may need to devise a methodical test strategy that satisfies these regulators that you have met all their requirements.
- **Product:** Some products such as weapons systems and contract development software tend to have well-specified requirements. This leads to synergy with a requirements-based analytical strategy.
- **Business:** Business considerations and business continuity are often important. If you can use a legacy system as a model for a new system, you can use a model-based strategy.

We mentioned above that a good team can sometimes triumph over a situation where materials, process and delaying factors are ranged against its success. However, talented execution of an unwise strategy is the equivalent of going very fast down a motorway in the wrong direction. Therefore, you must make smart choices in terms of testing strategies. Furthermore, you must choose testing strategies with an eye toward the factors mentioned earlier, the schedule, budget and feature constraints of the project and the realities of the organization and its politics.

5.2.3 Entry criteria and exit criteria (definition of ready and definition of done)

Two important things to think about when determining the approach and planning the testing are: how do we know we are ready to start a given test activity, and how do we know we are finished (with whatever testing we are concerned with)? At what point can you safely start a particular test level? When are you confident that it is complete? The factors to consider in such decisions are called entry and exit criteria or in Agile development, definition of ready and definition of done (DoD).

Typical **entry criteria** include the following:

Entry criteria
(definition of ready) The set of conditions for officially starting a defined task.

- Availability of testable requirements, user stories and/or models, for example when following a model-based testing strategy, that is, the test basis is available.
- Availability of test items that have met the exit criteria for any previous test levels.
- Availability of the test environment.

- Availability of necessary test tools and any other materials needed.
- Availability of test data and other necessary resources.
- Availability of staff for testing tasks.
- Availability of the component or system to be tested, in a state where tests can be done, that is, availability of the test object.

Why are entry criteria important? If everything needed for testing to go ahead is in place before you start, the testing will go much more smoothly. The problems of getting started with the testing often turn out to be that something needed is not actually ready or in place. Then the testing process gets the blame for the delays. If entry criteria, which are the preconditions for testing, are enforced, or at least thought about beforehand, everything is more likely to go better. If not, you are increasing risk, introducing delays and additional costs, and making life more difficult for yourself.

Exit criteria (completion criteria, test completion criteria, definition of done) The set of conditions for officially completing a defined task.

Typical **exit criteria** include the following:

- **Tests:** the number planned, prepared, run, passed, failed, blocked, skipped etc. are acceptable.
- **Coverage:** the extent to which the test basis (for example requirements, user stories, acceptance criteria), risk, functionality, supported configurations, and the software code have been tested (that is, achieved a defined level of coverage) – or have not.
- **Defects:** the number known to be present, the arrival rate, the number estimated to remain, the number resolved and the number of unresolved defects are within an agreed limit.
- **Quality:** the status of the important quality characteristics for the system, for example reliability, performance efficiency, usability, security and other relevant quality characteristics are adequate.
- **Money:** the cost of finding the next defect in the current level of testing compared to the cost of finding it in the next level of testing (or in production).
- **Schedule:** the project schedule implications of starting or ending testing.
- **Risk:** the undesirable outcomes that could result from shipping too early (such as latent defects or untested areas), or too late (such as loss of market share).

When writing exit criteria, we try to remember that a successful project or iteration is a balance of quality, budget, schedule and feature considerations. This is important in each sprint and is important in other development life cycles when exit criteria are applied at the end of the project.

Why are exit criteria important? Knowing what your goals are is important in any human endeavour. If we do not think about 'How will we know we are done' beforehand, then we may stop before we have done enough testing (increasing risk) or we might keep testing when we have done enough (not likely but possible, and this would be wasteful).

In practice, testing is often stopped rather than finished, because time pressure seems to hold the trump card for everything else. But if (or rather when) this happens, if you do have documented exit criteria, you can make the risks more visible to stakeholders and managers by showing what has not yet been completed in the testing. You may not win the argument for doing more testing at the time, but you will have good information to explain next time why the exit criteria are important, especially

when/if there are undesirable consequences to not meeting them this time. We also want a 'Stop testing now!' decision to be made with knowledge of all the factors, not just basing that decision on a calendar date.

5.2.4 Test execution schedule

Part of test management is the management of tests. Once test cases and test procedures have been designed and implemented (some as automated tests), they may be assembled into test suites for convenience of running sets of tests together. A schedule for the execution of the test suites should be based on a number of factors, including the priorities of the tests (from risk analysis), technical or logical dependencies of tests or test suites and the type of tests, for example confirmation tests after defects have been fixed, or regression tests. These factors need to be balanced with a sensible and efficient sequence of executing the tests.

For example, it may be that several high-priority tests are dependent on a single low-priority test to set up essential data or starting conditions. In that case, the low-priority test should be executed before the high-priority tests, even when risk priority is the most important factor.

The same logic may apply where tests are dependent on other tests. Actually, this is one reason why tests should ideally be designed to be independent of any other tests. Independent tests can be run in any order.

In Agile development, rapid feedback from tests is key to efficient working of the team, but this means that the test execution schedule is biased toward confirmation tests and short tests, which can be run quickly. This also shows why it is important to be able to identify (or tag) tests so that they can be assembled into different test suites for different execution schedules, something that is particularly important for automated tests.

It is more likely to be the tester or the team rather than the test manager who is making the test execution schedule, but whoever does it, they need to balance the trade-off between efficiency, priority of the tests and the objective of the test execution at the time.

This learning objective is a K3 in the Syllabus, which means that you need to be able to produce a test execution schedule, taking priorities and dependencies into account. See the exercise at the end of this chapter, and the mock exam in Chapter 7.

5.2.5 Factors influencing the test effort

In this section, we'll look at **test estimation**, first describing how to go about estimating testing, what it will involve and what it can cost, and then looking at factors that influence the test effort, many of which are significant for testing even though they are not part of what we are estimating when we estimate testing.

Test estimation
The calculated approximation of a result related to various aspects of testing, (for example, effort spent, completion date, costs involved, number of test cases, etc.), which is usable even if input data may be incomplete, uncertain or noisy.

Estimating what testing will involve and what it will cost

The testing work to be done can often be seen as a subproject within the larger project. We can adapt fundamental techniques of estimation for testing. We could start with a work-breakdown structure that identifies the stages, activities and tasks.

Starting at the highest level, we can break down a testing project into major activities using the test process identified in the ISTQB Syllabus (and described in Chapter 1 Section 1.4.2): test planning, test monitoring and control, test analysis, test design, test implementation, test execution and test completion. Within each activity, we identify tasks and perhaps subtasks. To identify the activities and tasks, we work both forward

and backward. When we say we work forward, we mean that we start with the planning activities and then move forward in time step-by-step, asking 'Now, what comes next?'

Working backward means that we consider the risks that we identified during risk analysis (which we'll discuss in Section 5.5). For those risks which you intend to address through testing, ask yourself, 'What activities and tasks are required in each stage to carry out this testing?' Let's look at an example of how you might work backward.

Suppose that you have identified performance as a major area of risk for your product. Performance testing is an activity in test execution. You now estimate the tasks involved with running a performance test, how long those tasks will take and how many times you will need to run the performance tests.

Now, those tests did not just appear out of thin air: someone had to develop them. Performance test development entails activities in test analysis, design and implementation. You now estimate the tasks involved in developing a performance test, such as writing test scripts and creating test data.

Typically, performance tests need to be run in a special test environment that is designed to look like the production or field environment, at least in those ways which would affect response time and resource utilization. Performance test environment acquisition and configuration is an activity in the test implementation activity. You now estimate tasks involved in acquiring and configuring such a test environment, such as simulating performance based on the production environment design to look for potential bottlenecks, getting the right hardware, software and tools and setting up that hardware, software and tools. Performance tests need special tools to generate load and check response. The acquisition and implementation of such tools also needs to be planned (more on tools in Chapter 6).

It may be possible to use virtualization for your test environment and performance testing tools; this seems to be an attractive idea, since it should save money as you do not have to acquire your own environment or tools. However, the use of virtual environments and using the tools in those environments still needs careful planning.

Not everyone knows how to use performance testing tools or to design performance tests. Performance testing training or staffing is a task in the test planning activity. Depending on the approach you intend to take, you now estimate the time required to identify and hire a performance test professional or to train one or more people in your organization to do the job.

Finally, in many cases a detailed test plan is written for performance testing, due to its differences from other test types. Performance testing planning is a task in test planning. You now estimate the time required to draft, review and finalize a performance test plan.

When you are creating your work-breakdown structure, remember that you will want to use it for both estimation (at the beginning) and monitoring and control (as the project continues). To improve the accuracy of the estimate and enable more precise control, make sure that you subdivide the work finely enough. This means that tasks should be short in duration, say one to three days. If they are much longer – say two weeks – then you run the risk that long and complex subtasks are hiding within the larger task, only to be discovered later. This can lead to nasty surprises during the project.

Factors that affect the test effort

Testing is a complex endeavour on many projects and a variety of factors can influence it. When creating test plans and estimating the testing effort and schedule, you must keep these factors in mind or your plans and estimates will deceive you at the beginning of the project and betray you at the middle or end.

The test strategies or approaches you pick will have a major influence on the testing effort, as we discussed in Section 5.2.2. In this section, let's look at factors related to the product, the development process, the people involved and the results of testing.

Product characteristics

The characteristics of the product that we are testing have a major impact on how we will test it:

- The risks associated with the product. A high-risk product needing a lot of testing will suffer much more severe impacts if testing is not estimated correctly, especially if it is under-estimated.
- The quality of the test basis. We want sufficient product documentation so that the testers can figure out what the system is, how it is supposed to work and what correct behaviour looks like. In other words, adequate and high-quality information about the test basis will help us do a better, more efficient job of defining the tests.
- The size of the product. A larger product leads to increases in the size of the project and the project team. This will also increase the difficulty of predicting and managing the projects and the team. This leads to the disproportionate rate of collapse of large projects.
- The requirements for quality characteristics such as usability, reliability, security, performance etc. also influences the testing effort. These test types can be expensive and time-consuming.
- The complexity of the product domain. Examples of complexity considerations include:
  - The difficulty of comprehending and correctly handling the problem the system is being built to solve, for example avionics and oil exploration software.
  - The use of innovative technologies, especially those long on hyperbole and short on proven track records.
  - The need for intricate and perhaps multiple test configurations, especially when these rely on the timely arrival of scarce software, hardware and other supplies.
  - The prevalence of stringent security rules, strictly regimented processes or other regulations.
- The geographical distribution of the team, especially if the team crosses time zones (as many outsourcing efforts do).
- The required level of detail for test documentation. While good project documentation is a positive factor, it is also true that having to produce detailed documentation, such as meticulously specified test cases, results in delays. During test execution, having to maintain such detailed documentation requires lots of effort, as does working with fragile test data that must be maintained or restored frequently during testing.
- Requirements for legal and regulatory compliance. If you are working in a regulated industry, the time and effort taken to meet those regulatory requirements can be significant.

Development process characteristics

The life cycle and development process in use has an impact on how the test effort is spent. Testing is quite different in Agile development compared to a sequential development life cycle, and other aspects of development also influence testing:

- The stability and maturity of the organization. Mature organizations tend to have better requirements, architecture and unit tests, thus saving test effort later in the life cycle.
- The development life cycle model in use. The life cycle model itself is an influential process factor, as the V-model tends to be more fragile in the face of late change while incremental models such as Agile development tend to have high regression testing costs.
- The test approach. Choosing the right approach is important for good testing, as we have seen. With a less than ideal approach, testing will take longer and take more effort than is necessary.
- The tools used. Tools are supposed to increase efficiency and reduce time spent on some tasks, so test tools, especially those that reduce the effort associated with test execution which is on the critical path for release, should decrease execution time for tests. Of course, other factors about automation may be far more significant, as we will see in Chapter 6. On the development side, debugging tools and a dedicated debugging environment (as opposed to debugging in the test environment) also reduce the time required to complete testing.
- The test process. A test process that is well-understood, with testers trained to perform the activities and tasks they need to do in the most effective and efficient way is the optimum. Test process maturity is another factor, since more mature testing will be more efficient and effective.
- Time pressure. This is another factor to be considered. Pressure should not be an excuse to take unwarranted risks. However, it is a reason to make careful, considered decisions and to plan and re-plan intelligently throughout the process, which is another hallmark of mature processes.

People characteristics

People execute the process, and people factors are as important or more important than any other. Indeed, even when many troubling things are true about a project, an excellent team can often make good things happen on the project and in testing. Important people factors include:

- The skills and experience of the people involved. The skills of individuals and the team as a whole are important, as well as the alignment of those skills with the project's needs. Domain knowledge is likely to be more relevant when the problem being solved is complex, thus requiring specific knowledge on the part of the tester to operate the software and to determine correct versus incorrect behaviour.
- Team cohesion and leadership. Since a project team is a team, solid relationships, reliable execution of agreed-upon commitments and responsibilities and a determination to work together toward a common goal are important. This is especially important for testing, where so much of what we test,

use and produce either comes from, relies upon or goes to people outside the testing group. Because of the importance of trusting relationships and the lengthy learning curves involved in software and system engineering, the stability of the project team is an important people factor, too.

Test results

The test results themselves are important in the total amount of test effort during test execution:

- The number and severity of defects found. Wouldn't testing be easy if we never found any defects? Initial tests would just run with no problems, and there would be no need for any tests to be repeated. However, in the real world, the more defects there are and the more severe those defects, the greater the impact on the testing and the test estimates.
- The amount of rework required. Delivery of good quality software at the start of test execution and quick, solid defect fixes during test execution prevents delays in the test execution process. A defect, once identified, should not have to go through multiple cycles of fix/retest/re-open, at least not if the initial estimate is going to be held to. Good design of the test object should make changes easier. For example, good modular design may have a low-level function called by a number of higher-level functions. If a change is made in the low-level function, the functions calling it should work without being changed (if correctly designed).

You probably noticed from this list that we included a number of factors outside the scope and control of the test manager. Indeed, events that occur before or after testing can bring these factors about. For this reason, it is important that testers, especially test managers, be attuned to the overall context in which they operate. Some of these contextual factors result in specific project risks for testing, which should be addressed in the test plan. Project risks are discussed in more detail in Section 5.5.

5.2.6 Test estimation techniques

There are two techniques for estimation covered by the ISTQB Foundation Syllabus. One involves consulting the people who will do the work and other people with expertise on the tasks to be done (expert-based). The other involves analyzing metrics from past projects and from industry data (metrics-based). Let's look at each in turn.

Asking the individual contributors and experts involves working with experienced staff members to develop a work-breakdown structure for the project. With that done, you work together to understand, for each task, the effort, duration, dependencies and resource requirements. The idea is to draw on the collective wisdom of the team to create your test estimate. Using project management software or a whiteboard and sticky notes, you and the team can then predict the testing end date and major milestones. This technique is often called bottom-up estimation, because you start at the lowest level of the hierarchical breakdown in the work-breakdown structure (the task) and let the duration, effort, dependencies and resources for each task add up across all the tasks. This is the expert-based technique.

Analyzing metrics can be as simple or sophisticated as you make it. The simplest approach is to ask, 'How many testers do we typically have per developer on a project?' A somewhat more reliable approach involves classifying the project in terms of size (small, medium or large) and complexity (simple, moderate or complex) and then seeing on average how long projects of a particular size and complexity combination have taken in the past. Another simple and reliable approach we have used is to look at the average effort per test case in similar past projects and to use the estimated number of test cases to estimate the total effort. Sophisticated approaches involve building mathematical models in a spreadsheet or other tool that look at historical or industry averages for certain key parameters – number of tests run by a tester per day, number of defects found by a tester per day, etc. – and then plugging in those parameters to predict duration and effort for key tasks or activities on your project. The tester-to-developer ratio is an example of a top-down estimation technique, in that the entire estimate is derived at the project level, while the parametric technique is bottom-up, at least when it is used to estimate individual tasks or activities.

We prefer to start by drawing on the team's wisdom to create the work-breakdown structure and a detailed bottom-up estimate. We then apply models and rules of thumb to check and adjust the estimate bottom-up and top-down using past history. This approach tends to create an estimate that is both more accurate and more defensible than either technique by itself.

Even the best estimate must be negotiated with management. Negotiating sessions exhibit amazing variety, depending on the people involved. However, there are some classic negotiating positions. It is not unusual for the test manager to try to sell the management team on the value added by the testing or to alert management to the potential problems that would result from not testing enough. It is not unusual for management to look for smart ways to accelerate the schedule or to press for equivalent coverage in less time or with fewer resources. In between these positions, you and your colleagues can reach compromise, if the parties are willing. Our experience has been that successful negotiations about estimates are those where the focus is less on winning and losing and more about figuring out how best to balance competing pressures in the realms of quality, schedule, budget and features.

In Agile development, planning poker is an example of the expert-based technique; team members estimate based on their own experience of the effort needed to deliver and test a feature. Burndown charts are an example of the metrics-based technique, since the effort being spent is captured and reported and used to feed into the team's velocity to determine the amount of work the team can do in the next iteration. This is actually monitoring of the current sprint in order to use the results to help predict and estimate the effort needed for the following sprint. The ISTQB Foundation Level Agile Tester Extension Syllabus has more on estimation of testing in Agile development.

In sequential life cycle models, a technique such as Wideband Delphi estimation is an example of expert-based estimation, since groups of engineers provide estimates which are then aggregated together.

Defect removal models are examples of metrics-based techniques. Data is gathered from previous projects about the number of defects and the time to remove them; this provides a basis for future similar projects. More information about these are given in the ISTQB Advanced Level Test Manager Syllabus.

5.3 TEST MONITORING AND CONTROL

SYLLABUS LEARNING OBJECTIVES FOR 5.3 TEST MONITORING AND CONTROL (K2)

FL-5.3.1 Recall metrics used for testing (K1)

FL-5.3.2 Summarize the purposes, contents and audiences for test reports (K2)

In this section, we'll review techniques and metrics that are commonly used for monitoring test implementation and execution. We'll focus especially on the use and interpretation of such test metrics for reporting, controlling and analyzing the test effort, including those based on defects and those based on test data. We'll also look at options for reporting test status using such metrics and other information.

As you read, remember to watch for the Glossary terms **test control**, **test monitoring**, **test progress report** and **test summary report**.

Test monitoring is concerned with gathering data and information about test activities; **test control** is using that information to guide or control the remaining testing. We will look briefly at test control in this introduction, then consider various metrics that could be gathered, and finally discuss how information should be communicated in test reports.

Test monitoring A test management activity that involves checking the status of testing activities, identifying any variances from the planned or expected status and reporting status to stakeholders.

Test control A test management task that deals with developing and applying a set of corrective actions to get a test project on track when monitoring shows a deviation from what was planned.

Projects do not always unfold as planned. In fact, any human endeavour more complicated than a family picnic is likely to vary from plan. Risks become occurrences. Stakeholder needs evolve. The world around us changes. When plans and reality diverge, we must act to bring the project back under control, and testing is no exception.

In some cases, the test findings themselves are behind the divergence; for example, suppose the quality of the test items proves unacceptably bad and delays test progress. In other cases, testing is affected by outside events; for example, testing can be delayed when the test items show up late or the test environment is unavailable. Test control is about guiding and corrective actions to try to achieve the best possible outcome for the project.

The specific corrective or guiding actions depend, of course, on what we are trying to control. Consider the following hypothetical examples of test control actions:

- A portion of the software under test will be delivered late, after the planned test start date. Market conditions dictate that we cannot change the release date. Test control might involve re-prioritizing the tests so that we start testing against what is available now.
- For cost reasons, performance testing is normally run on weekday evenings during off-hours in the production environment. Due to unanticipated high demand for your products, the company has temporarily adopted an evening shift that keeps the production environment in use 18 hours a day, five days a week. Test control might involve rescheduling the performance tests for the weekend.
- The item being tested had previously met its entry (or exit) criteria, but since the criteria were evaluated, the test item has changed due to defects found and resulting rework. Control actions may include re-evaluating the entry (or exit) criteria.

While these examples show test control actions that affect testing, the project team might also have to take some actions that affect others on the project. For example, suppose that the test completion date is at risk due to a high number of defect fixes that fail confirmation testing in the test environment. In this case, test control might involve requiring the developers making the fixes to thoroughly retest the fixes prior to checking them in to the code repository for inclusion in a test build.

5.3.1 Metrics used in testing

The purpose of metrics in testing

Having developed our plans, defined our test strategies and approaches, and estimated the work to be done, we must now track our testing work as we carry it out. Test monitoring can serve various purposes during the project, including the following:

- Give the test team and the test manager feedback on how the testing work is going, allowing opportunities to guide and improve the testing and the project. This would be done by collecting time and cost data about progress versus the planned schedule and budget.
- Provide the project team with visibility about the test results and the quality of the test object.
- Measure the status of the testing, test coverage and test items against the exit criteria to determine whether the test work is done, and to assess the effectiveness of the test activities with respect to the objectives.
- Gather data for use in estimating future test efforts, including the adequacy of the test approach.

Especially for small projects, the test manager or a delegated person can gather test progress monitoring information manually using documents, spreadsheets and simple databases. When working with large teams, distributed projects and long-term test efforts, we find that the efficiency and consistency of data collection is aided by the use of automated tools (see Chapter 6).

Common test metrics

We will show some examples of using metrics in testing and conclude this section with a list of other common test metrics.

In Figure 5.1, columns A and B show the test ID and the test case or test suite name. The state of the test case is shown in column C ('Warn' indicates a test that resulted in a minor failure). Column D shows the tested configuration, where the codes A, B and C correspond to test environments described in detail in the test plan. Columns E and F show the defect (or bug) ID number (from the defect tracking database) and the risk priority number of the defect (ranging from 1, the highest risk, to 25, the least risky). Column G shows the initials of the tester who ran the test. Columns H to L capture data for each test related to dates, effort and duration (in hours). We have metrics for planned and actual effort and dates completed which would allow us to summarize progress against the planned schedule and budget. This spreadsheet can also be summarized in terms of the percentage of tests which have been run and the percentage of tests which have passed and failed.

Figure 5.1 might show a snapshot of test progress during the test execution period, or perhaps even at test closure if it were deemed acceptable to skip some of the tests.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1		**System Test Case Summary**											
2			**Cycle One**										
3													
4	**Test**			**System**	**Bug**	**Bug**	**Run**	**Plan**	**Act**	**Plan**	**Actual**	**Test**	
5	**ID**	**Test Suite/Case**	**Status**	**Config**	**ID**	**RPN**	**By**	**Date**	**Date**	**Effort**	**Effort**	**Duration**	**Comment**
6													
7	1.000	*Functionality*											
8	1.001	File	Fail	A	701	1	LTW	1/8	1/8	4	6	6	
9	1.002	Edit	Fail	A	709	1	LTW	1/9	1/10	4	8	8	
10					710	5							
11					718	3							
12					722	4							
13	1.003	Font	Pass	B			IHB	1/10	1/10	4	4	4	
14	1.004	Tables	Warn	B	708	15	IHB	1/8	1/9	4	5	5	
15	1.005	Printing	Skip					1/10		4			Out of runway
16		**Suite Summary**						**1/10**	**1/10**	**20**	23	23	
17													
18	2.000	*Performance/Stress*											
19	2.001	Solaris Server	Warn	A,B,C	701	1	EM	1/10	1/13	4	8	24	Replan 1/11
20	2.002	NT Server	Fail	A,B,C	724	2	EM	1/11	1/14	4	4	24	Replan 1/12
21					713	2							
22					725	1							
23	2.003	Linux Server	Skip					1/12		4			Out of runway
24		**Suite Summary**						**1/12**	**1/14**	**12**	**12**	**48**	
25													
26	3.000	*Error Handling/Recovery*											
27	3.001	Corrupt File	Fail	A	701	1	LTW	1/8	1/9	4	8	8	
28					706	2							
29					707	4							
30					709	1							
31					710	5							
32					713	2							
33	3.002	Server Crash	Fail	A	712	6	LTW	1/9	1/10	4	6	6	
34					713	2							
35					717	1							
36		**Suite Summary**						**1/9**	**1/10**	**8**	**14**	**14**	
37													
38	4.000	*Localization*											
39	4.001	Spanish	Skip										
40	4.002	French	Skip										
41	4.003	Japanese	Skip										
42	4.004	Chinese	Skip										
43		**Suite Summary**						**1/0**	**1/0**	**0**	**0**	**0**	

FIGURE 5.1 Test case summary worksheet

During the analysis, design and implementation of the tests, such a worksheet would show the state of the tests in terms of their state of development.

In addition to test case status, it is also common to monitor test progress during the test execution period by looking at the number of defects found and fixed. Figure 5.2 shows a graph that plots the total number of defects opened and closed over the course of the test execution so far.

It also shows the planned test period end date and the planned number of defects that will be found. Ideally, as the project approaches the planned end date, the total number of defects opened will settle in at the predicted number and the total number of defects closed will converge with the total number opened. These two outcomes tell us that we have found enough defects to feel comfortable that we have finished, that we have no reason to think many more defects are lurking in the product and that all known defects have been resolved.

Charts such as Figure 5.2 can also be used to show failure rates or defect density. When reliability is a key concern, we might be more concerned with the frequency with which failures are observed than with how many defects are causing the failures. In organizations that are looking to produce ultra reliable software, they may plot the

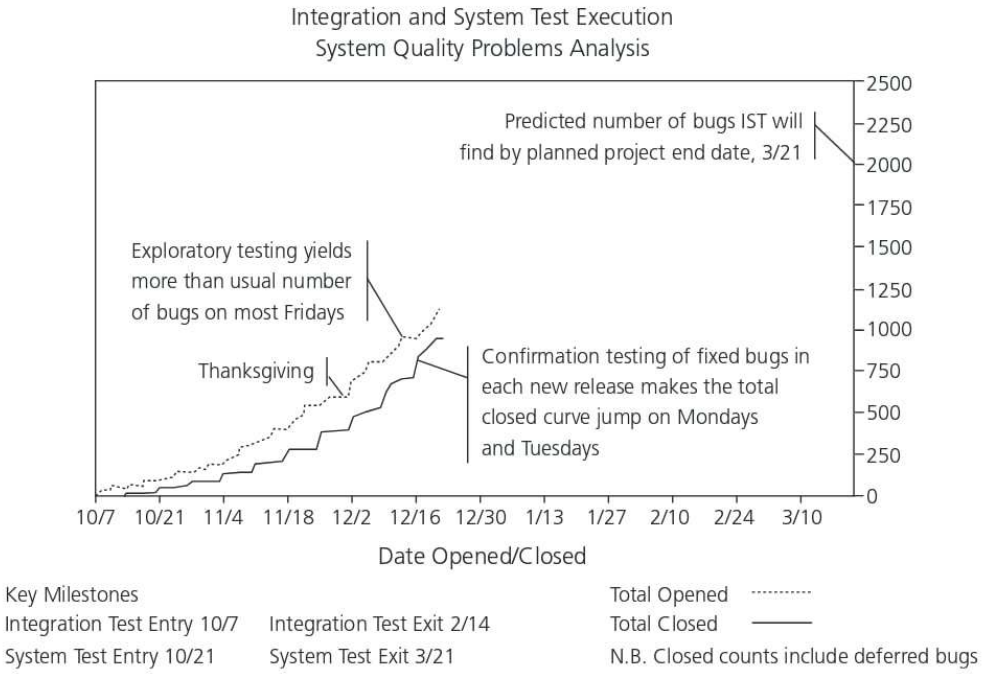


FIGURE 5.2 Total defects opened and closed chart

number of unresolved defects normalized by the size of the product, either in thousands of source lines of code (KSLOC), function points (FP) or some other metric of code size. Once the number of unresolved defects falls below some predefined threshold – for example, three per million lines of code – then the product may be deemed to have met the defect density exit criteria.

Measuring test progress based on defects found and fixed is common and useful, if used with care. Avoid using defect metrics alone, as it is possible to achieve a flat defect find rate and to fix all the known defects by stopping any further testing, by deliberately impeding the reporting of defects and by allowing developers to reject, cancel or close defect reports without any independent review.

That said, test progress monitoring techniques vary considerably depending on the preferences of the testers and stakeholders, the needs and goals of the project, regulatory requirements, time and money constraints and other factors.

Common metrics for test progress monitoring include:

- The percentage of planned work done in test case preparation (or percentage of planned test cases implemented).
- The percentage of planned work done in test environment preparation.
- Metrics relating to test execution, such as the number of test cases run and not run, the number of test cases passed or failed, or/or the number of test conditions passed or failed.
- Metrics relating to defects, such as defect density, defects found and fixed, failure rate and confirmation test results.
- The extent of test coverage achieved, measured against requirements, user stories, acceptance criteria, risks, code, configurations or other areas of interest.

- The status of the testing (including analysis, design and implementation) compared to various test milestones, that is, task completion, resource allocation and usage, and effort.
- The economics of testing, such as the costs and benefits of continuing test execution in terms of finding the next defect or running the next test.

As a complementary monitoring technique, you might assess the subjective level of confidence the testers have in the test items. However, avoid making important decisions based on subjective assessments alone, as people's impressions have a way of being inaccurate and coloured by bias.

5.3.2 Purpose, contents and audiences for test reports

The purpose of test reports

Test monitoring is about gathering detailed test data; reporting test progress is about effectively communicating our findings to other project stakeholders. As with test progress monitoring, in practice there is wide variability observed in how people report test progress, with the variations driven by the preferences of the testers and stakeholders, the needs and goals of the project, regulatory requirements, time and money constraints and limitations of the tools available for test progress reporting. Often variations or summaries of the metrics used for test progress monitoring, such as Figure 5.1 and Figure 5.2, are used for test progress reporting, too. Regardless of the specific metrics, charts and reports used, test progress reporting is about helping project stakeholders understand the results of a test period, especially as it relates to key project goals and whether (or when) exit criteria were satisfied.

In addition to notifying project stakeholders about test results, test progress reporting is often about enlightening and influencing them. This involves analyzing the information and metrics available to support conclusions, recommendations and decisions about how to guide the project forward or to take other actions. For example, we might estimate the number of defects remaining to be discovered, present the costs and benefits of delaying a release date to allow for further testing, assess the remaining product and project risks and offer an opinion on the confidence the stakeholders should have in the quality of the system under test.

You should think about test progress reporting during test planning, since you will often need to collect specific metrics during and at the end of a test period to generate the test progress reports in an effective and efficient fashion. The specific data you will want to gather will depend on your specific reports, but common considerations include the following:

- How will you assess the adequacy of the test objectives for a given test level and whether those objectives were achieved?
- How will you assess the adequacy of the test approaches taken and whether they support the achievement of the project's testing goals?
- How will you assess the effectiveness of the testing with respect to these objectives and approaches?

For example, if you are doing risk-based testing, one main test objective is to subject the important product risks to the appropriate extent of testing. Table 5.1 shows an example of a chart that would allow you to report your test coverage and

TABLE 5.1 Risk coverage by defects and tests

	Unresolved defects		Test cases to be run		
Product risk areas	**Number**	**%**	**Planned**	**Actual**	**%**
Performance, load, reliability	304	28	3,843	1,512	39
Robustness, operations, security	234	21	1,032	432	42
Functionality, data, dates	224	20	4,744	2,043	43
Use cases, user interfaces, localization	160	15	498	318	64
Interfaces	93	8	193	153	79
Compatibility	71	6	1,787	939	53
Other	21	2	760	306	40
	1,107	*100*	*12,857*	*5,703*	*44*

unresolved defects against the main product risk areas you identified in your risk analysis. If you are doing requirements-based testing, you could measure coverage in terms of requirements or functional areas instead of risks.

The content of test reports

Test progress report (test status report)
A test report produced at regular intervals about the progress of test activities against a baseline, risks and alternatives requiring a decision.

Test summary report (test report)
A test report that provides an evaluation of the corresponding test items against exit criteria.

The Syllabus discusses two types of test report: a **test progress report** and a **test summary report**. Both reports contain a lot of information in common; the main difference is that test progress reports are used at regular intervals throughout the project, during test activities, and would result in test control actions; the test summary report is in a sense the final test progress report for the whole project, and is prepared at the end of a test activity or test level. For example, a test summary report could also be used in a retrospective.

Because test progress reports are reporting on more dynamic information, there are some things which are included in this report that are no longer relevant for a test summary report, including:

- The current status of the test activities and progress against the test plan.
- Factors that are currently impeding the progress of the testing.
- The testing planned for the next period, to be including in the next test progress report.
- The quality of the test object as currently assessed by the testing.

When exit criteria are met, the test manager issues the test summary report. Such a report, created either at a key milestone or at the end of a test level, describes the results of a given test level. In addition to including the kind of charts and tables shown earlier (but now at the end of the time period being monitored), you might discuss important events (especially problematic ones) that occurred during testing, the objectives of testing and whether they were achieved, the test approach, strategies

followed, how well they worked and the overall effectiveness of the test effort. The test summary report draws on the test progress reports over the duration of the project or test level, particularly the last one.

The contents of both test progress reports and test summary reports may include the following:

- A summary of the testing performed.
- Information about what occurred during the period the report covers.
- Deviations from plan, with regard to the schedule, duration or effort of test activities.
- The status of the testing and of product (test object) quality with respect to the exit criteria or definition of done.
- The factors that have blocked or continue to block progress.
- Relevant metrics, such as those relating to defects, test cases, test coverage, test activity progress, and resource consumption, as described in Section 5.3.1.
- Residual risks (see Section 5.5).
- Reusable test work products produced.

Not every test report will contain all this information (for example a quick software update), and some may include additional information (for example for a complex project with many stakeholders or under legal and regulatory requirements). The content of the report depends on the project, organizational requirements and the software development life cycle. For example, in Agile development, test progress reporting may be incorporated into task boards, defect summaries and burndown charts, which may be discussed during a daily stand-up meeting. For more about test reports in Agile projects, see the ISTQB-AT Foundation Level Agile Tester Extension Syllabus.

The audience for test reports and the effect on the report

In addition to the factors already mentioned that influence the content of a test report (either progress or summary), each test report should be tailored for the intended audience of the report. For example, a test summary report which is intended for the CEO and other high-level management should be short and concise, with overview summaries of the main points. It may contain a summary of defects of high and other priority levels and the percentage of tests passed, but information about budget and schedule would be of more importance to them. A test progress report which is intended for testers and developers would include detailed information about defect types and trends, such as we saw in Figure 5.2.

The standard ISO/IEC/IEEE 291119-3 [2013] gives structures and examples of test progress reports and what they call test completion reports (test summary reports).

5.4 CONFIGURATION MANAGEMENT

SYLLABUS LEARNING OBJECTIVE FOR 5.4 CONFIGURATION MANAGEMENT (K2)

FL-5.4.1 Summarize how configuration management supports testing (K2)

In this brief section, we'll look at how configuration management relates to and supports testing. The only Glossary term is **configuration management**.

Configuration management A discipline applying technical and administrative direction and surveillance to identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements.

Configuration management is a topic that often perplexes new practitioners, but, if you ever have the bad luck to work as a tester on a project where this critical activity is handled poorly, you will never forget how important it is. Briefly put, configuration management is in part about determining clearly what the items are that make up the software or system. These items include source code, test scripts, third-party software (including tools that support testing), hardware, data and both development and test documentation. Configuration management is also about making sure that these items are managed carefully, thoroughly and attentively throughout the entire project and product life cycle.

Configuration management has a number of important implications for testing. For one thing, it allows the testers to manage their testware and test results using the same configuration management mechanisms, as if they were as valuable as the source code and documentation for the system itself – which of course they are.

For another thing, configuration management supports the build process, which is essential for delivery of a test release into the test environment. It is critical to have a solid, reliable way of delivering test items that work and are the proper version.

Last but not least, configuration management allows us to map what is being tested to the underlying files and components that make it up. This is absolutely critical. For example, when we report defects, we need to report them *against* something, something which is configuration controlled or version controlled. If it is not clear what we found the defect in, the developers will have a very tough time of finding the defect in order to fix it. For the kind of test reports discussed earlier to have any meaning, we must be able to trace the test results back to what exactly we tested.

Configuration management for testing may involve ensuring the following:

- All test items of the test object are uniquely identified, version controlled, tracked for changes and related to each other, that is, what is being tested.
- All items of testware are uniquely identified, version controlled, tracked for changes, related to each other and related to a version of the test item(s) so that traceability can be maintained throughout the test process.
- All identified work products and software items are referenced unambiguously in test documentation.

Ideally, when testers receive an organized, version-controlled test release from a change-managed source code repository, it is accompanied by release notes which contain all the information shown.

While our description was brief, configuration management is a topic that is as complex as test environment management. Advanced planning is critical to making this work. During the project planning stage, and perhaps as part of your own test plan, make sure that configuration management procedures and tools are selected. As the project proceeds, the configuration process and mechanisms must be implemented, and the key interfaces to the rest of the development process should be documented. Come test execution time, this will allow you and the rest of the project team to avoid nasty surprises like testing the wrong software, receiving uninstallable builds and reporting unreproducible defects against versions of code that do not exist anywhere but in the test environment.

5.5 RISKS AND TESTING

SYLLABUS LEARNING OBJECTIVES FOR 5.5 RISK AND TESTING (K2)

FL-5.5.1 **Define risk level by using likelihood and impact (K1)**

FL-5.5.2 **Distinguish between project and product risks (K2)**

FL-5.5.3 **Describe, by using examples, how product risk analysis may influence the thoroughness and scope of testing (K2)**

This section covers a topic that we believe is critical to testing: risk. Let's look closely at risks, the possible problems that might endanger the objectives of the project stakeholders. We will discuss how to determine the level of risk using likelihood and impact. We will see that there are risks related to the product and risks related to the project, and look at typical risks in both categories. Finally, and most important, we'll look at various ways that risk analysis and risk management can help us plot a course for solid testing

As you read this section, make sure to attend carefully to the Glossary terms **product risk**, **project risk**, **risk**, **risk level** and **risk-based testing**.

5.5.1 Definition of risk

Risk A factor that could result in future negative consequences.

Risk is a word we all use loosely, but what exactly is risk? Simply put, it is the possibility of a negative or undesirable outcome. In the future, a risk has some likelihood between 0% and 100%; it is a possibility, not a certainty. In the past, however, either the risk has materialized and become an outcome or issue or it has not; the likelihood of a risk in the past is either 0% or 100%.

Risk level (risk exposure) The qualitative or quantitative measure of a risk defined by impact and likelihood.

The likelihood of a risk becoming an outcome is one factor to consider when thinking about the **risk level** associated with its possible negative consequences. The more likely the outcome is, the worse the risk. However, likelihood is not the only consideration.

For example, most people are likely to catch a cold in the course of their lives, usually more than once. The typical healthy individual suffers no serious consequences. Therefore the overall level of risk associated with colds is low for this person. But the risk of a cold for an elderly person with breathing difficulties would be high. The potential consequences or impact is an important consideration affecting the level of risk, too.

Remember that in Chapter 1 we discussed how system context, and especially the risk associated with failures, influences testing. Here, we'll get into more detail about the concept of risks, how they influence testing and specific ways to manage risk.

5.5.2 Product and project risks

We can classify risks into project risks (factors relating to the way the work is carried out, that is, the test project) and product risks (factors relating to what is produced by the work, that is, the thing we are testing). We will look at product risks first.

Product risks

Product risk A risk impacting the quality of a product.

You can think of a **product risk** as the possibility that the system or software might fail to satisfy some reasonable customer, user or stakeholder expectation. Unsatisfactory software might omit some key function that the customers specified, the users required or the stakeholders were promised. It might be unreliable and frequently fail to behave normally. It might fail in ways that cause financial or other damage to a user or the company that user works for. It might have problems with data integrity and data quality, such as data migration issues, data conversion problems, data transport problems and violations of data standards such as field and referential integrity. It might have problems related to a particular quality characteristic. The problems might not involve functionality, but rather security, reliability, usability, maintainability or performance. This type of quality attribute-related risk is sometimes referred to as a 'quality risk'. Generally, the software could fail to perform its intended functions, dissatisfying users and customers. If you can write a test for it, it's a product risk.

Here are some example product risks:

- Software might not perform its intended functions according to the specification.
- Software might not perform its intended functions according to user, customer, and/or stakeholder needs or expectations (which is usually different from the first!).
- A particular computation may be performed incorrectly in some circumstances.
- A loop control structure may be coded incorrectly.
- Response times may be inadequate for a high-performance transaction processing system.
- User experience (UX) feedback might not meet product expectations.

Project risks

Project risk A risk that impacts project success.

We just discussed risks to product quality. However, testing is an activity like the rest of the project and thus it is subject to risks that endanger the project. To deal with the **project risks** that apply to testing, we can use the same concepts we apply to identifying, prioritizing and managing product risks, which we will discuss in Section 5.3.3.

Remembering that a risk is the possibility of a negative outcome, what project risks affect testing? There are direct risks such as the late delivery of the test items to the test team or availability issues with the test environment. There are also indirect risks such as excessive delays in repairing defects found in testing or problems with getting professional system administration support for the test environment.

Of course, these are merely four examples of project risks; many others can apply to your testing effort. To discover these risks, ask yourself and other project participants and stakeholders: 'What could go wrong on the project to delay or invalidate the test plan, the test strategy and the test estimate? What are unacceptable outcomes of testing or in testing? What are the likelihoods and impacts of each of these risks?' This process is very much like the risk analysis process for products. Checklists and examples can help you identify test project risks [Black 2004].

The Syllabus gives a good list of project risks in different categories. We list them here along with some examples and ways to deal with them.

Project issues:

- Delays may occur in delivery, task completion or satisfaction of exit criteria or definition of done. For example, logistics or product quality problems may block tests: These can be mitigated through careful planning, good defect triage and management, and robust test design.
- Inaccurate estimates, reallocation of funds to higher priority projects, or general cost cutting across the organization may result in inadequate funding.
- Late changes may result in substantial rework. For example, excessive change to the product may invalidate test results or require updates to test cases, expected results and environments. These can be mitigated through good change control processes, robust test design and lightweight test documentation. When severe defects occur, transference of the risk by escalation to management is often in order.

Organizational issues:

- Skills, training and staff may not be sufficient. For example, shortages of people, skills or training may lead to problems with communicating and responding to test results, unrealistic expectations of what testing can achieve, and needlessly complex project or team organization.
- Personnel issues may cause conflict and problems, affecting work. For example, if two people do not get along, working against each other rather than toward a common goal (it happens), this is demoralizing to the whole team as well as degrading the quality of everyone's work.
- Users, business staff or subject matter experts may not be available due to conflicting business priorities.

Political issues:

- Testers may not communicate their needs and/or the test results adequately. For example, high-level managers may not realize the need for ongoing support, time and effort for test automation after an initial start, if testers and automators do not communicate the need for this.
- Developers and/or testers may fail to follow up on information found in testing and reviews (for example not improving development and testing practices).
- There may be an improper attitude toward, or expectations of, testing (for example not appreciating the value of finding defects during testing).

Technical issues:

- Requirements or user stories may not be clear enough or well enough defined. For example, ambiguous, conflicting or unprioritized requirements, an excessively large number of requirements given other project constraints, high system complexity and quality problems with the design, the code or the tests mean that the system will take longer to develop and may not meet customer expectations. The best cure for this is clear unambiguous requirements.
- The requirements may not be met, given existing constraints. For example, a requirement may want the app to be able to check the user's bank balance to see if they can pay for what they are ordering, but the bank software does not allow it.

- The test environment may not be ready on time or may not be adequate. For example, insufficient or unrealistic test environments may yield misleading results, including false positives and false negatives. One option is to transfer the risks to management by explaining the limits on test results obtained in limited environments. Mitigation, sometimes complete alleviation, can be achieved by outsourcing tests such as performance tests that are particularly sensitive to proper test environments, or by using virtualization.
- Data conversion, migration planning and their tool support may be late.
- Weaknesses in the development process may impact the consistency or quality of project work products such as design, code, configurations, test data and test cases. For example, test items may not install in the test environment. This can be mitigated through smoke (or acceptance) testing prior to starting other testing or as part of a nightly build or continuous integration. Having a defined uninstall process is a good contingency plan.
- Poor defect management and similar problems may result in accumulated defects and other technical debt.

Supplier issues:

- A third party may fail to deliver a necessary product or service or go bankrupt.
- Contractual issues may cause problems to the project. For example, problems with underlying platforms or hardware, failure to consider testing issues in the contract, or failure to properly respond to the issues when they arise can quickly add up to serious delays as well as time-consuming negotiations with a supplier. The best way to mitigate this is to ensure that the contract is solid to begin with, and have the technical details reviewed by testers or a test manager.

Project risks do not just affect testing of course, they also affect development. In some organizations, project managers are responsible for dealing with all project risks, but sometimes test managers are given responsibility for test-related project risks (as well as product risks).

There may be other risks that apply to your project and not all projects are subject to the same risks. See Chapter 2 of Black [2009], Chapters 6 and 7 of Black [2004] and Chapter 3 of Craig and Jaskiel [2002] for a discussion on managing project risks during testing and in the test plan.

Finally, don't forget that test items can also have risks associated with them. For example, there is a risk that the test plan will omit tests for a functional area or that the test cases do not exercise the critical areas of the system.

5.5.3 Risk-based testing and product quality

Risk management

Dealing with risks within an organization is known as risk management, and testing is one way of managing aspects of risk. For any risk, product or project, you have four typical options:

- **Mitigate:** Take steps in advance to reduce the likelihood (and possibly the impact) of the risk.
- **Contingency:** Have a plan in place to reduce the impact should the risk become an outcome.