

**Fig. 3.4** Generic review process


specific review process to a given situation. If the required review is more formal, then there will be more tasks or elements of the process. A generic review process described in ISO/IEC 20246 is shown in Fig. 3.4.

The size of many work products means that they may be too large to be covered by a single review. In such cases, the review process is typically applied multiple times to the individual parts that make up the work product.

The Foundation Level syllabus describes five generic types of activities shown in Fig. 3.4 that can occur in the work product review process. We discuss them below.

### Planning

Planning defines the scope of work that the review will cover. Planning sets the boundaries of the review by answering questions like who, what, where, when, and why. The purpose of the review is defined (answering the “why” question), as well as which documents will be the subject of the review. The quality characteristics to be reviewed are defined (answering the “what” question). The amount of work needed to conduct the review is estimated, and the timeframe and location of certain phases of the review process, such as the review meeting, are defined (answering the “where” and “when” questions). Based on the purpose of the review, the type of review is defined, together with the roles, activities, and checklists to be used (if necessary). The people who are to participate in the review are selected, and roles are assigned (answer to the “who” question).

For **formal reviews**  such as inspection, formal entry and exit criteria are defined, and at the end of this phase, it is also verified that the entry criteria are met and that the review can proceed to the next phase.

### Review initiation

As part of the start of the review, the work product to be reviewed is sent out to the review participants (selected in the planning phase), along with all other necessary materials, e.g., checklists, statements of procedure, and defect report templates. All materials should be distributed as early as possible so that reviewers have sufficient time to complete the review.

If necessary, any explanations are given to the participants about how the review will take place and what their role is. If participants have any questions, answers and explanations are provided during the review initiation phase—before the actual review of the work product takes place—so that everyone is well aware of what they are supposed to do and knows the timeframe for these activities.

It is possible to arrange for review training during this phase. This makes sense when the participants are inexperienced in reviews and you want the review process to run smoothly and efficiently. For formal reviews, an initial meeting can also be held to explain to individual participants the scope of the review and their individual roles and responsibilities as reviewers.

The purpose of the review initiation phase is to make sure that all those involved in the review are prepared and ready to begin the review.

### **Individual review (i.e., individual preparation)**


This is the essential, central phase of any review. In this phase, substantive activities are performed, that is, the actual review of the work product by the reviewers takes place, using specific review techniques (see Sect. 3.2.6). As part of these activities, all or part of the work product is reviewed (what part is to be reviewed should be determined during planning and carefully explained to the participants in the review initiation phase). The reviewers record any comments, questions, recommendations, concerns, and relevant observations made during the review of the work product.

Of all the review phases, the individual review phase detects the largest percentage of problems. Identified problems are usually documented in a problem log, which is often supported by a defect management or review support tool.

According to the standard ISO/IEC 20246 [6], this process step is optional<sup>7</sup> and may not be performed for certain types of review, such as walkthroughs or informal reviews. However, many authors take the position that it is the most important activity in the entire review process.

### **Communication and analysis**

If a review meeting is not scheduled, the problems found are communicated directly to individuals (e.g., the author of the work product being reviewed), along with an analysis of those problems. If the review meeting is included in the review plan, the problems are reported directly to the meeting participants, or, which is probably better, they are collectively sent to all participants prior to the meeting to review them in advance, so that the review meeting will run more efficiently.

The meeting includes an analysis of problems (**anomalies** ) reported by reviewers. Since not every anomaly necessarily turns out to be a defect, all findings must be carefully reviewed at the meeting to decide whether there is a real problem (defect) or merely an apparent problem (false positive). If the anomaly turns out to be a defect, those responsible for fixing them are appointed, and parameters such as status, priority, or severity are defined. These actions are performed at a review

---

<sup>7</sup>In such situations, finding defects can take place, for example, during the so-called review meeting discussed in the next paragraph.

meeting or (if there is no such meeting) individually. The most commonly used statuses for reported anomalies are:

- Problem rejected
- Problem recorded without taking any action
- Problem to be solved by the work product author
- Problem updated as a result of further analysis
- Problem assigned to an external stakeholder

This phase also evaluates and documents the level of quality characteristics that were defined in the planning phase as those being reviewed. Finally, the conclusions of the review are evaluated against exit criteria to decide what to do with the reviewed work product. Examples of decisions include:

- Acceptance of the work product (usually when no problems or only a small number of insignificant issues are detected).
- Update the work product based on the identified problems (typical decision).
- The work product should be reviewed again (usually due to the large number of problems and the large scope of changes).
- Rejection of the work product (the rarest type of decision, but it can also happen).

### Fixing and reporting

This is the final stage of the review. It creates defect reports on detected defects that require changes. It is expected that the author of the reviewed work product will carry out defect removal during this phase. This is done by informing the relevant persons or the relevant team of the defects detected in the work product under review. Finally, when the changes are confirmed, a review report is created.

For more formal types of review, in addition, various types of measures are collected and used to test the effectiveness of the review. It is also checked against exit criteria, and the work product is accepted once it is determined that the exit criteria have been met.

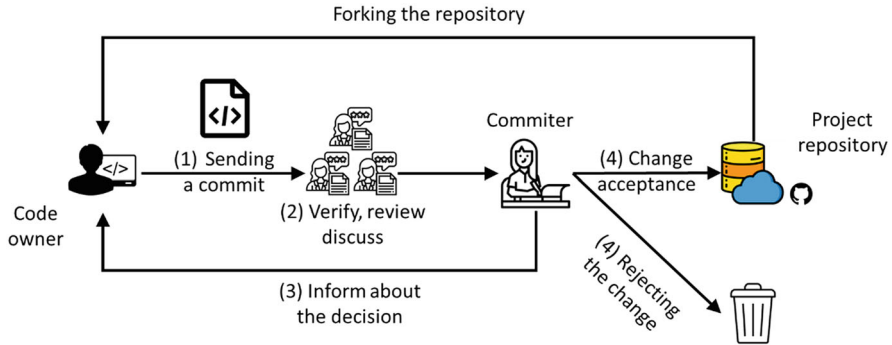
The results of a work product review can vary depending on the type of review and the degree of formalization (see Sect. 3.2.4).

**Example** A team has decided to perform a review of the so-called skill tree in an online RPG computer game they are producing. The test leader selects five people as participants in the review: a developer, a tester, and three people from outside the team, serving as external stakeholders (players). It is decided that the type of review conducted will be an individual informal review.

The team leader distributes to participants a document describing the skill tree, a checklist of characteristics to be checked, and a problem log template, shown in Table 3.2.

Each participant, at a predetermined time, reviews the document and completes the problem log form and then sends it to the team leader. The team leader merges the forms into one, removes redundant defects (possibly marking that they were found by more than one person), and sends the list of problems thus created to the





**Fig. 3.5** Modern Code Review process

### Effectiveness of review meetings

The Foundation Level syllabus says that the review meeting is one of the steps in the review process. Looking historically at how reviews have been organized, one can see that until some time ago, reviews were focused on review meetings, which were the main, substantive activity of the review process. In the past dozen years or so, the problem of the necessity and effectiveness of these meetings has been studied. These studies have concluded that, in general, these meetings do not increase significantly the effectiveness of defect detection (which is usually the most important goal of reviews). Review meetings are expensive, so they do not seem to add any particular value. In addition, it turns out that if there are too many people at a meeting (more than five), the communication overhead begins to have a negative impact on the learning and skill improvement of those attending.

On the other hand, the review meetings are an excellent opportunity to talk and exchange experiences, share knowledge, or gain consensus. Studies also show that these meetings help reduce the number of false positives (reports of anomalies that turn out not to be defects). Thus, the added value can be revealed not necessarily in the number of defects detected, but in the improvement of the entire development process and in the improvement of team members' skills [32].

### Modern Code Review (MCR)

Today, many companies use a code review process known as *Modern Code Review* (MCR). A diagram of this process is shown in Fig. 3.5. MCR is an effective quality control technique that can verify software quality and customer satisfaction by identifying defects, improving code, and speeding up the development process. It is an asynchronous and lightweight review process supported by review tools such as Gerrit. It is a lightweight version of Fagan's inspection process (see Sect. 3.2.4) and has evolved as a practice for open-source and industrial software development [33].

### ***3.2.3 Roles and Responsibilities in Reviews***

In a typical review, the following roles are distinguished (the description of the roles and the list of responsibilities are taken from the syllabus—at the end, we give some additional notes on some of the roles):

#### **Manager**

- Is responsible for scheduling the review
- Decides to conduct a review
- Designates staff and sets a budget and timeframe
- Monitors the cost-effectiveness of the review on an ongoing basis
- Executes control decisions in case of unsatisfactory results

#### **Author**

- Creates the work product under review
- Removes defects in the work product under review (if necessary)

The author is the person responsible for preparing the materials for review, although formally the materials may be distributed by the review leader. If necessary, the author may provide reviewers with technical explanations of the work product under review. It is very important that the authors understand and accept the reviewers' professional criticism of their work product and that they do not defend or deny the reviewers' comments. After all, the purpose of the review is usually to uncover as many problems as possible, and the criticism is not directed at the author (because everyone is fallible), but at the product.

However, it may happen that the reviewer's remark is not understood by the author and the author does not understand what the problem is and therefore does not know how to fix it. Then it may be necessary to communicate between the author and the reviewer to explain the reviewer's concerns in more detail.

The author can also evaluate the work of the reviewers in terms of the value of the comments they make. This type of feedback can be used in planning future reviews in the organization.

#### **Moderator (also known as facilitator)**

- Ensures the smooth running of review meetings (if they take place)
- Acts as a mediator if it is necessary to reconcile different points of view
- Ensures that a safe atmosphere of mutual trust and respect is created at the review meeting

#### **Scribe (also known as the recorder)**

- Gathers potential anomalies detected and reported as part of the individual review
- Records new potential defects found during the review meeting, as well as decisions made at the meeting (if it takes place)

The scribe should play a "transparent" role during the review meeting, i.e., they should be "invisible" to the other participants. Their role is to relieve the other

participants of the task of recording any comments made during the meeting. A review meeting requires maximum focus from the reviewers and the author, and they will work most effectively when they do not have to be distracted every time a new problem is found and there is a need to write it down.

**Reviewer**

- Conducts a review, identifying potential defects in the work product under review
- Can be a subject matter expert, a person working on the project, a stakeholder interested in the work product, and/or a person with specific technical or business experience
- Can represent different viewpoints (e.g., the viewpoint of a tester, developer, user, operator, business analyst, usability specialist)

Reviewers play a key role in the review process, as they find problems in the work product being reviewed, and this is usually the main purpose of reviews. Reviewers should be given sufficient time to prepare and to report the problems they find. Reviewers should direct their comments at the product, not the author.

**Review leader**

- Bears overall responsibility for the review process
- Decides who is to participate in the review, determines the place and date of the review, and is responsible for organizing the review meetings

**Roles vs. persons**

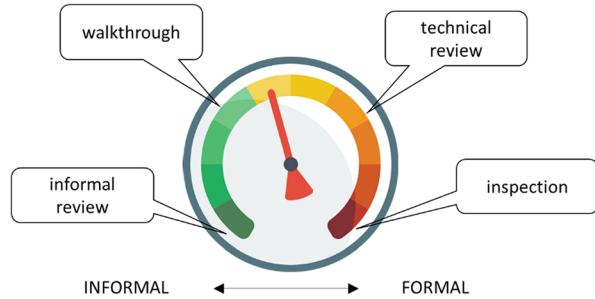
For some types of reviews, one person may perform several roles, and depending on the type of review, the activities associated with each role may also vary. In addition, with the advent of tools to assist in the review process (and in particular the recording of defects, open points, and decisions), there is often no need to appoint a scribe. A scribe will also not be needed if no review meeting is planned.

### ***3.2.4 Review Types***

There are many types of reviews with different levels of formality, from informal reviews to highly formalized reviews (see Fig. 3.6). The level of formality required depends on factors such as the SDLC model used, the maturity of the development process, the criticality and complexity of the work product being reviewed, any legal or regulatory requirements, and the need for an audit trail.

Choosing the right type of review is critical to achieving the required review objectives. The selection is based not only on the objectives but also on factors such as project needs, available resources, type of work product, risks, business domain, and organizational culture.

**Fig. 3.6** Level of formality of different types of review



The Foundation Level syllabus describes four types of reviews:

- Inspection
- Technical review
- Walkthrough
- Informal review

Inspection is a formal review. Technical review and walkthrough can range from very formalized to quite informal. The hallmark of informal reviews is that they do not follow a defined process and the information obtained through them does not need to be formally documented. Formal reviews, on the other hand, are conducted in accordance with documented procedures and with the participation of a pre-established team, and the results must be documented mandatorily.

Reviews can be conducted for various purposes, but one of the main goals of any review is to find defects. Reviews can be classified according to various attributes. Table 3.4 shows the most common review types discussed in the Foundation Level syllabus, along with their corresponding attributes. Below, we will provide some more detailed information on these review types.

### Order of reviews

A work product may be subject to more than one type of review, and if several reviews of different types are conducted, their order may vary. For example, an informal review may be conducted before a technical review to ensure that the work product is ready for technical review.

### Peer reviews

All types of reviews can be implemented as peer reviews, that is, conducted by colleagues at a similar organizational level or with similar responsibilities or experience.

### Types of defects found during reviews

The types of defects found in reviews vary, depending in particular on the work product being reviewed. For examples of defects found in reviews of various work products, see Sect. 3.1.3, and for information on formal reviews, see [34].


We now turn to a detailed discussion of the four types of reviews described in the syllabus. A summary comparison of the review types discussed below is summarized in Table 3.4.



**Table 3.4** Comparison of review types

| Type of review             | Informal review  | Walkthrough  | Technical review   | Inspection   |
|----------------------------|--|--|--|--|
| Main goals                 | Detect potential defects   | Detect potential defects, improve quality, consider alternatives, evaluate compliance with standards | Obtain consensus, detect potential defects   | Detect potential defects, assess the quality of the work product, increase confidence in it, prevent similar defects from occurring in the future  |
| Potential additional goals | Generate new ideas, quickly solve simple problems                        | Exchange information, train participants, reach consensus  | Assess the quality of a work product, increase confidence in it, generate new ideas, motivate authors to improve future work products, evaluate alternatives | Motivate authors to improve future work products and the software development process, create conditions for this, reach consensus                 |
| Formal process             | None   | Optional individual preparation before the meeting   | Mandatory individual preparation before the meeting  | Formal process based on rules and checklists; mandatory individual preparation before meeting  |
| Roles                      | It can be run by an author, by an author's peer, or by a group of people | The meeting is usually chaired by the author; the presence of a scribe is required                   | The meeting should be conducted by a moderator, not by the author; the presence of a scribe is required  | Strictly defined; meeting is led by a moderator, not the author; scribe is mandatory; optional reader role   |
| Documentation of results   | Optional   | Optional defect logs and review reports are created  | In general, defect logs and review reports are created   | In general, defect logs and review reports are created   |
| Level of formalism         | Informal   | From informal to formal; can take the form of scenarios, dry runs, or simulations                    | From informal to formal; review meeting optional   | Formal; entry and exit criteria are in place; measures are collected that are used to improve the entire process, including the inspection process |
| Checklists                 | Optional   | Optional   | Optional   | Usually used   |

### Informal review


**Informal review**  does not follow any established or predefined process. Nor is the result of an informal review documented in any formal way. The main purpose of an informal review is to detect potential anomalies.

An example of an informal review might be a conversation between two developers, when one of them asks the other for help in finding a defect causing a compilation error. Another example might be a conversation between two engineers in the office kitchen over lunch about some technical problem. After an informal review, often no trace is left, and nothing is documented. In agile methodologies, this is the most common type of review.

Examples of informal review types include:

- Buddy check
- Pair review


### Walkthrough

**Walkthrough**  involves the sequential review of a work product. It is conducted by the author of the work product and may include several different objectives, such as assessing the quality of the work product, increasing confidence in the work product, improving the reviewers' understanding of the work product, reaching consensus, generating new ideas, and motivating authors to create better products and to find defects more effectively. Reviewers can do individual preparation before the walkthrough meeting, but it is not mandatory.

This type of review is also often used when the team is unable to detect the cause of a software failure. It can then take the form of so-called dry runs. A test run may involve manual simulation of the code execution. The team then analyzes the code carefully, line by line, to get a good understanding of how it works and to locate the defect causing the failure.

The walkthrough is conducted by the author of the code, as the author is usually able to explain on the fly what the code does (or at least what their intentions were in this regard when they implemented the code). When *code reviews* are conducted at review meetings, they usually take the form of a walkthrough.


### Technical review

The objectives of a **technical review** , performed by technically qualified reviewers, are to gain consensus and make decisions on a technical problem, but also to detect potential defects, assess quality and build confidence in the work product, generate new ideas, and motivate and enable authors to improve their work product.

A technical review usually takes the form of an “expert panel,” that is, a meeting at which competent people are tasked with making some—usually important and significant—design or technical decision that has a major impact on the further development of the project. For this reason, individual preparation before the meeting is mandatory, so that in the meeting itself, everyone can participate in an informed and competent manner.

In general, a review meeting in a technical review is optional. However, if it takes place, it should be led by a facilitator. Usually, the technical review ends with a report, as there should be a trail left behind—especially if important design decisions were made during the review.

### Inspection

**Inspections**  are one of the most formal review types. Their origin dates back to the 1970s, when Michael Fagan introduced them at IBM [35]. Conducting software inspections is considered a so-called best practice in software quality engineering because of the undoubted benefits it brings. Senior management has systematically emphasized the important role of peer reviews (including inspections) as a key element to ensure high final product quality [36].

Since inspections are the most formal type of review, they follow the full review process described in Sect. 3.2.2. The main goal is to achieve maximum efficiency in finding defects. Other goals are to assess quality, build confidence in the work product, and motivate and enable authors to improve their work products. A special feature of inspections is collecting metrics and using them to improve the entire software development process, including the inspection process itself. In inspections, the author must not take the role of review leader, moderator, or scribe.

Inspections follow a well-defined process based on rules and checklists. The results of inspections must be documented. An example of a checklist used during an inspection might be in the following form.

#### Checklist for requirements

##### Completeness

1. Do the specified requirements relate to the project mission in a consistent manner?
2. Do the requirements include key and critical needs of the user, operator, and support team?
3. Is each requirement a separate specification unit independent of the others or does it have clearly defined dependencies?
4. Are there no deficiencies in the requirements?
5. Are necessary requirements distinguished from optional requirements?

##### Correctness

1. Are the requirements mutually non-contradictory?
2. Are requirements tracked forward to design and code?
3. Does each requirement have a unique number?

##### Style

1. Is each requirement easy to understand and written in plain language?
2. Do the requirements make a clear distinction between editorial and functional changes?
3. Are the nomenclature and definitions of terms used consistently and uniformly?

### Inspection measurements

Although many organizations use inspections, there is little publicly available data on their effectiveness. However, based on existing information (particularly from the National Software Quality Experiment, conducted in 1992), the following conclusions can be drawn [37]:

- Source code is not tracked back to requirements, resulting in loss of “intellectual” control, imprecision of verification procedures, and difficulty in change management.
- Good code writing practices are applied in a non-rigorous and unsystematic manner, resulting in a high percentage of defects in logic, data, interfaces, and functionality.
- Application architectural designs are often created ad hoc, which drastically reduces the comprehensibility, adaptability, and maintainability of the product.
- No significant use is made of existing design patterns and programming patterns.
- The distribution of defect types detected during inspection is as follows:
  - Documentation defects—40.51%
  - Non-compliance with standards—23.20%
  - Defects in logic—7.22%
  - Functional defects—6.57%
  - Syntax defects—4.79%
  - Data defects—4.62%
  - Maintainability defects—4.09%
- On average, a team needs to spend 8–16 min to detect one defect and about 80 min to detect a high-severity defect (known as a major defect).
- Approximately 3.2 major defects and 17 less significant defects for every 1000 lines of code are found in the code under inspection.
- On average, the team is able to analyze 625 lines of code in an hour.
- On average, the team finds 4.6 defects per session.
- The ratio of inspection preparation effort to inspection effort is 0.58.
- *Defect detection rate* ranges between 80% and 90%—this means that about 80–90% of all defects detected are found during inspection.
- The return on investment (ROI calculated as the ratio of profits to costs) is 4.1.

Metrics that can be collected as part of the inspection process include:

- Time to prepare per defect
- Time to prepare per major defect
- Number of critical defects per 1000 lines of code
- Number of non-critical defects per 1000 lines of code
- Number of lines of code checked in 1 h

(continued)

- Number of defects per session
- Effort to prepare versus effort to inspect
- Number of lines checked in one session

The analysis of these metrics, especially from a number of inspections conducted, makes it possible to estimate the cost, effort, and effectiveness of this form of static testing and calculate the return on investment. Such analyses can be very useful in convincing managers to use reviews in the daily work of teams by demonstrating their effectiveness and that they result in a reduction in the overall cost of software production and maintenance.

**Example** As part of static testing techniques, the ABC organization uses two types of reviews: walkthroughs and inspections. Walkthroughs are used for verification purposes. They are conducted to gain knowledge about various aspects of the work product. A side goal of walkthroughs is:

- To achieve a consistent, unified product vision within the organization
- To obtain consensus among stakeholders on specific product features
- To obtain agreement on the engineering techniques to be used
- To ascertain the completeness and correctness of the capabilities and features of the software being developed

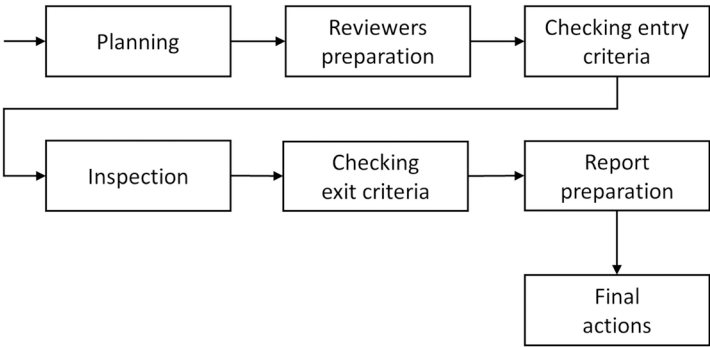
Reviews are conducted by the authors of each work product. Within each activity of the development cycle, several reviews of the same work product can be conducted. The only quantity measured is the number of walkthroughs conducted.

Inspections are carried out to meet the requirements of the team responsible for quality management. This team verifies that the work product complies with the applicable standards that the organization must follow. Formal inspections are carried out at the end of each activity of the development cycle, so-called quality gates, during which specific exit criteria for a given development phase are checked. Since inspection is a formal process, it follows a well-defined process shown in Fig. 3.7.

The inspection strictly and formally checks the work product for:

- Completeness
- Correctness
- Coherence
- Style
- Construction rules

The inspection is conducted by the moderator, and in addition, the process involves a scribe, two to five reviewers, and the author. Conducting an inspection is treated as a formal check of the exit criteria of a phase. During the inspection, product and process measurements are taken, and the entire session is reported in



**Fig. 3.7** Inspection process in an ABC organization

**Table 3.5** Comparison of walkthroughs and inspections used in the ABC organization

| Category           | Walkthrough                         | Inspection                            |
|--------------------|-------------------------------------|---------------------------------------|
| Overall objective  | Do the right job                    | Do the job right                      |
| Specific objective | Education, understanding, consensus | Defect detection, compliance checking |
| Trigger            | Author’s request                    | Phase exit criteria                   |
| Measurement        | Walkthrough instances               | Product and process measurements      |

report templates specifically defined for this purpose. Defects found during the inspection are tracked in defect management tool until they are closed.

Table 3.5 summarizes the differences between walkthroughs and inspections. For more information on reviews, see [34, 38, 39].

We will conclude this section with two examples of what a sample review of the system architecture might look like.

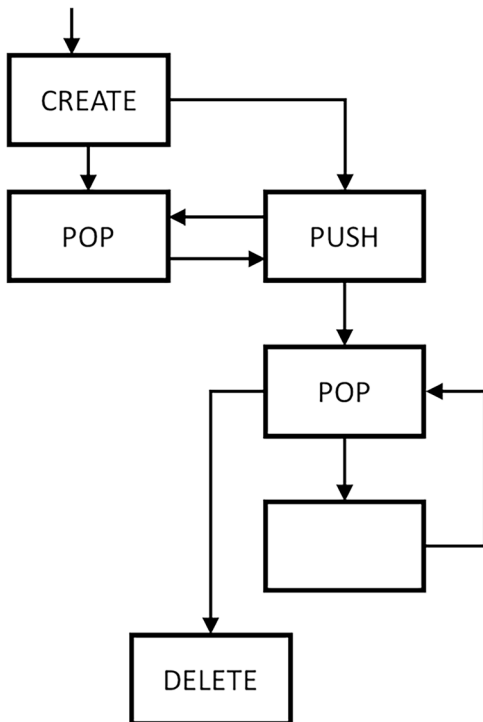
**Example** Figure 3.8 shows the design of an information flow architecture on a certain data structure used in the system, called a stack. Possible operations on the stack are:

- CREATE—creation of a stack
- PUSH—putting an item on top of the stack
- POP—pulling an item from the top of the stack
- DELETE—remove the stack structure from the system

The tester is currently in the individual preparation phase. The checklist used by the tester consists of the following items:

1. Is each PUSH and POP operation performed only after the stack has been created (CREATE)?
2. Does the system disallow the execution of POP on an empty stack?

**Fig. 3.8** Stack operations system flow



3. Does the system prohibit deleting the stack (DELETE) when it is non-empty?
4. Will the number of stack elements never exceed 10?

The tester uses a checklist-based review technique and checks which properties on the list are satisfied and which are not.

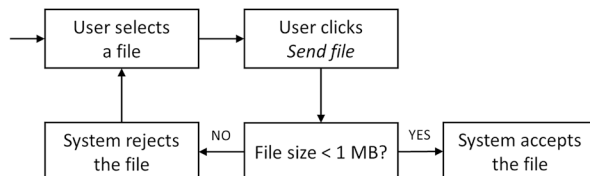
Point 1 is satisfied—every PUSH and POP statement is created after the CREATE operation, and no PUSH or POP statement can be executed after the DELETE operation. An inquisitive tester will notice, however, that the specification does not say whether the stack is empty when it is created or whether it contains, for example, some default initial elements. It is worth pointing out to the document's author that they should make this clear in the specification.

Point 2 is not fulfilled. After the CREATE operation, the POP operation (on an empty stack) can be performed immediately.

Point 3 is fulfilled. After each PUSH operation, the next operation must be a POP operation. Thus, only one element can be stored on the stack, which will be removed from the stack in the following step. At this point, the tester may point out that since there can be at most one element on the stack, why use a complicated stack structure when you could use a simpler structure (like a variable or a register) that allows us to store only one element?

Point 4 is fulfilled—this follows from the analysis from point 3.

**Fig. 3.9** Business process model



As you can see from the example above, the reviewer did not limit themselves to “check off” individual items on the list, but also noted some shortcomings in the specification. This is a valuable review feedback that can serve to improve the work product (specification) by its author.

**Example** An architect has asked a developer to conduct an informal peer review. The review concerns a process model created by the architect that implements the following business requirement: “A user can upload any file smaller than 1 GB through a web form. If the file size is less than this value, the system accepts it, otherwise it rejects the file.”

The reviewed model is shown in Fig. 3.9.

The developer has received from the architect both a requirement and a diagram showing the process. The developer checks that the diagram corresponds to the requirement and notices that there is a minor, but very significant, typo when deciding whether the system should accept or reject the file: the condition says “*File size < 1 MB*” but should read “*File size < 1 GB*.” The developer informs the architect of the noticed typo, and the architect corrects the design.

### 3.2.5 Success Factors for Reviews

The key to a successful review is the careful selection of the review type and the review techniques used. In addition, a number of other factors should be taken into account that may affect the result. Success factors include, but are not limited to, the following factors:

- Each review has clear objectives defined during planning that can serve as measurable exit criteria.
- The types of reviews used are conducive to achieving the stated goals and are appropriate to the type and level of software work products and to the participants.
- Various review techniques (such as checklist-based or role-based review) are used to effectively identify defects present in a work product.
- Checklists used are up-to-date and address the main risks.
- Large documents are written and reviewed in batches, so that authors get quick and frequent feedback on defects (which is part of quality control).



- Participants have sufficient time to prepare for the review; reviews are scheduled in advance.
- Authors are given feedback from reviewers, so they can improve their work product and the quality of their work.
- Management supports the review process (e.g., by designating sufficient time for review activities in the project schedule).
- Reviews are considered a natural part of organizational culture to promote learning and product and process improvement.
- The review involves people whose participation is conducive to achieving its goals—for example, people with different skills or viewpoints who will potentially use the document in the course of their work.
- Testers are recognized as important participants in the review, and the knowledge they gain about the work product enables them to prepare more effective tests in advance.
- Participants allocate sufficient time to take part in the review and show due attention to detail.
- Reviews are conducted on small pieces so that reviewers do not lose focus during the individual review and/or review meeting (if held).
- Detected defects are acknowledged, confirmed, and dealt with objectively.
- Review meetings are moderated in the right way so that participants do not waste time on unnecessary activities.
- The review takes place in an atmosphere of mutual trust, and its results are not used to evaluate participants.
- Participants avoid gestures and behaviors that could indicate boredom, irritation, or hostility toward other participants.
- Due training was provided to participants, especially for the more formal types of reviews (such as inspections).
- An atmosphere conducive to expanding knowledge and improving processes has been created.

**Example** A review leader was asked to organize a code review of component X (in the form of a walkthrough) for a group of developers. The review leader conducted the inspection, inviting the author of component X and a team of testers to participate.

This scenario lacked both organizational and people-related success factors:

- The review leader was asked to organize a walkthrough, but organized an inspection, which is not the best type of review for a code review. Thus, the organizational success factor was not met (choosing the appropriate review type to achieve the given objectives, and to suit the type of work product, the review participants, the project needs and context.)
- The review leader was asked to organize a code review for developers, but—except for the author—invited only testers. Such a code review will be ineffective. Thus, the success factor of a personnel nature has not been met (the review involves people whose participation is conducive to achieving its goals.)

### 3.2.6 (\*) *Review Techniques*

The Foundation Level syllabus in the previous version (3.1) described five different techniques that a reviewer can use as part of an individual review activity, that is, individual preparation for review. This activity can, of course, be performed as part of all types of review, in particular all four types of review described in Sect. 3.2.3. The purpose of these techniques is to detect defects in the reviewed work product. The five techniques mentioned are:

- Ad hoc review
- Checklist-based review
- Scenarios and dry runs
- Role-based review
- Perspective-based reading

#### **Ad hoc review**

This is the traditional approach to defect detection by reviewers. The process is completely unstructured. Each reviewer is tasked with detecting as many defects of all possible types as possible. The effectiveness of such a review is strongly dependent on the skills of the individual reviewers. It usually leads to the detection of the same (usually obvious or trivial) problems by many different reviewers. During an ad hoc review, reviewers receive little (or no) guidance on how to perform the task. Participants often read the work product sequentially, identifying and documenting the problems found on the fly.

#### **Checklist-based review**

A checklist-based review is a more structured technique than an ad hoc review. A good practice for this type of review is for different reviewers to receive different checklists. This will increase potential product coverage and detect more defects. It also reduces the risk of more people detecting the same problem multiple times, which is a waste of time and resources. The most important advantage of the checklist-based technique is the systematic coverage of the most common types of defects.

There is a risk with this approach, that reviewers may strictly limit themselves only to the issues in the checklist and ignore other potential problems in the work product under review. Reviewers should therefore be aware that they have more responsibility than just blindly following checklist items.

Checklists should be selected according to the type of work product and the purpose of the team. So a checklist for the use of a requirements review will be quite different from, for example, a checklist for the use of security testing or interface usability testing. The checklist may even be specific to the particular method used to produce the work product. For example, a checklist used for testing banking-related products may include statutory regulations imposed on banks, while one used in the automotive industry may in turn be based on the ISO 26262 standard [40].

A common problem that arises when using this approach is that checklists become longer and longer over time and therefore less handy to use. A typical

checklist should contain no more than approx. 10 items and should be reviewed and updated regularly. Updates should especially address those problems, failures, and defects that we ourselves have detected in our product. Such a checklist, based on our experience, will be far better (in terms of defect detection efficiency) than, for example, a standard checklist found on the Internet.

A common approach is to use risk analysis and expand the checklist to include identified risks with the highest severity level. This allows the greatest risks to be directly checked when using a checklist approach.

An example of a checklist-based review is presented in Sect. 3.2.4 (a stack example).

### **Scenarios and dry runs**

The scenario-based approach works well when the requirements, the design, or the tests themselves are documented in an appropriate “scenario” format, such as use cases. In this approach, reviewers conduct so-called dry runs on the work product, checking that the product’s functionality is described correctly and that common exceptions caused by product misbehavior are handled appropriately.

There is a risk that viewers will follow defined scenarios too closely. In such a situation, they can easily overlook some defects, such as missing functionality in the product.

As with the checklist approach, scenarios can be expanded to include aspects arising from the risk analysis to ensure that the most important and frequently used scenarios are explored most thoroughly.

### **Role-based review**

Role-based review is a technique in which reviewers evaluate a work product from the perspective of particular stakeholder roles. Typical roles include specific types of end users (experienced, inexperienced, elderly, children, etc.) or specific roles in the organization (user, administrator, system administrator, performance tester, etc.).

Often, roles are modeled by means of a so-called persona. A persona is a fictional but concrete character that symbolizes a certain type of user. By making this persona concrete (specifying, e.g., its gender, age, and interests), it is easier for the team to “get into” the type of user. An example of a persona is shown in Fig. 3.10.

### **Perspective-based reading**

The perspective-based reading technique is described in the literature as one of the most effective individual review methods [41]. It is based on the fact that different reviewers take different stakeholder perspectives when reviewing a work product and review the product from that angle. By considering multiple views of the product, reviewers are able to discover a greater number of potential problems. At the same time, by assigning specific perspectives to reviewers, each reviewer is able to review the product thoroughly and in detail, and since each corresponds to a different perspective, the risk of detecting duplicate defects is low.

Perspective-based reading is not only limited to taking different points of view. It also requires reviewers to try to use the work product under review to generate a first prototype of the product to see if it is possible to do so based on the information



### Benjamin Thompson

|                          |   |
|--------------------------|---|
| <b>Position:</b>         | Mobile applications tester                      |
| <b>Age:</b>              | 28 yo   |
| <b>City:</b>             | New York  |
| <b>Hobbies:</b>          | movies, FPS games, travelling, new technologies |
| <b>Character traits:</b> | vigorous, impatient, curious                    |

**Fig. 3.10** Example of a persona

available in the reviewed work product. For example, testers will try to generate draft versions of acceptance tests if they conduct a perspective-based review of the requirements specification to verify that the specification contains all the necessary information. In addition, perspective-based reading often uses checklists.

Typical perspectives a viewer might take are:

- User
- Business analyst
- Designer
- Tester
- System administrator
- Marketing and promotions manager
- Technical support engineer

If the technique is limited to the end user's point of view, a perspective-based reading technique can also be used, distinguishing between types of system users, such as:

- End user
- System administrator
- Technical support engineer

A key factor in the success of perspective-based reading is to take into account and balance the viewpoints of the various stakeholders in a way that is appropriate to the risk. Which viewpoints we adopt should depend on the context. For example:

- If the document being reviewed is requirements, typical perspectives will be user, designer, and tester.
- If the system is related to a highly regulated area (e.g., aviation, banking, medical systems), the regulator's point of view should definitely be included in the review.
- If the system is going to be used for a long time, the point of view of the system maintenance team should be adopted.

**Role-based review vs. perspective-based reading**

Perspective-based reading is similar to the previously discussed role-based review technique. The two approaches are sometimes equated in the literature, but they differ. The difference can perhaps be explained most simply as follows:

- In a role-based review, the “types” of users change, but the tasks to be performed remain the same (e.g., different types of bank customers).
- In perspective-based reading, the “perspectives” of the stakeholders or, in the case of the users themselves, the types of tasks to be performed (e.g., end user, administrator, business analyst, tester) change.

**Example** The team uses a perspective-based reading technique. The person reviewing the document—the requirements specification—takes the perspective of the tester and uses the following procedure to read the specification.

**READING PROCEDURE.** For each requirement, create a test or set of tests to ensure that the implementation meets the requirement. Use a standard test approach and test criteria and techniques to build the test suite. When creating tests for each requirement, answer the following questions:

1. Do you have enough information to identify the test item and the testing criteria? Are you able to create reasonable test cases for each item based on these criteria?
2. Is there another requirement for which you could generate a similar test case, but with a different expected result?
3. Are you sure that the test you generated will provide the right values in the right units?
4. Are there other interpretations of this requirement that a developer could adopt, based on how the requirement is defined? Would it affect your testing?
5. Is the requirement reasonable and rational in terms of your knowledge of the application and what is included in the overall system description?

**Sample Questions****Question 3.1**

(FL-3.1.1, K1)

Your organization has just prepared an official document describing how reviews should be performed in the organization.

Can this document be reviewed?

- A. Yes, because any human-understandable document can be static tested.
- B. No, because we would have to apply the rules described in this document to itself during the review.
- C. No, because the document is not a work product of either the testing process or the development process.
- D. No, because reviews can only be done for specifications and source code.

Choose one answer.

**Question 3.2**

(FL-3.1.2, K2)

While analyzing the code, the tester noticed that the cyclomatic complexity of one of the code components was very high. The tester passed this information to the developers, who refactored the code, making it more readable and testable. This scenario shows the benefit of using:

- A. Dynamic testing.
- B. Static testing.
- C. Test management.
- D. Formal test technique.

Choose one answer.

**Question 3.3**

(FL-3.1.3, K2)

What is the difference between static and dynamic techniques, due to the purpose of these techniques?

- A. Static techniques directly detect failures, while dynamic techniques directly detect defects.
- B. Static techniques are typically used early in the SDLC, while dynamic techniques are typically used in later phases of the SDLC.
- C. There is no difference, as both types of techniques aim to detect defects as early as possible.
- D. Static techniques usually require programming skills, while dynamic techniques usually do not.

Choose one answer.

**Question 3.4**

(FL-3.2.1, K1)

Consider the following statements about early feedback.

- i. Early feedback gives developers more time to produce new features of the system, as they spend less time on modifying the features planned in a given iteration.
- ii. Early feedback allows agile teams to deliver the features with the greatest business value first, as the customer's attention remains focused on the features most important from the customer's point of view.
- iii. Early feedback reduces the overall cost of testing because it reduces the amount of time testers need to test the system.
- iv. Early feedback increases the likelihood that the system produced will be close to customers' expectations, since they have the opportunity to make changes during each iteration.

Which of these statements are true?

- A. (i) and (iv) are true; (ii) and (iii) are false.
- B. (ii) and (iii) are true; (i) and (iv) are false.
- C. (ii) and (iv) are true; (i) and (iii) are false.
- D. (i) and (iii) are true; (ii) and (iv) are false.

Choose one answer.

**Question 3.5**

(FL-3.2.2, K2)

Which of the following is part of the review initiation phase?

- A. Selecting who will participate in the review.
- B. Collecting metrics.
- C. Answering participants' questions about scope, goals, process, roles, and work products.
- D. Identifying the scope of work, including the type of review and the documents (or parts thereof) that are the subject of the review and the quality characteristics to be assessed.

Choose one answer.

**Question 3.6**

(FL-3.2.3, K1)

Which role is responsible for the smooth running of the review meeting?

- A. Review leader.
- B. Moderator.
- C. Author.
- D. Reviewer.

Choose one answer.

**Question 3.7**

(FL-3.2.4, K2)

Over the past few days, the team has been trying to find the cause of a strange system failure. Since they could not figure out the cause, the team decided to “simulate the computer” by manually executing the code line by line to understand what exactly the program was doing and thus discover the cause of the software’s strange behavior.

What type of review will be most appropriate for this activity?

- A. Inspection.
- B. Informal review.
- C. Technical review.
- D. Walkthrough.

Choose one answer.

**Question 3.8**

(FL-3.2.5, K1)

Inspection will be better carried out if:

- A. Its main purpose will be defined as “evaluation of alternatives”.
- B. The manager will attend review meetings.
- C. Reviewers will be trained to conduct reviews.
- D. Metrics will not be collected throughout the review process.

Choose one answer.



# Chapter 4 Test Analysis and Design



## Keywords

### Acceptance criteria

the criteria that a component or system must satisfy in order to be accepted by a user, customer, or other authorized entity. *References:* ISO 24765.

### Acceptance test-driven development

a collaboration-based test-first approach that defines acceptance tests in the stakeholders' domain language. *Abbreviation:* ATDD.

### Black-box testing technique

a test technique based on an analysis of the specification of a component or system. *Synonyms:* black-box test design technique, specification-based technique.

### Boundary value analysis

a black-box test technique in which test cases are designed based on boundary values.

### Branch coverage

the coverage of branches in a control flow graph.

### Checklist-based testing

an experience-based test technique in which test cases are designed to exercise the items of a checklist.

### Collaboration-based test approach

an approach to testing that focuses on defect avoidance by collaborating among stakeholders.

### Coverage

the degree to which specified coverage items are exercised by a test suite, expressed as a percentage. *Synonyms:* test coverage.

|  |   |
|--|---|
| <b>Coverage item</b>                   | an attribute or combination of attributes derived from one or more test conditions by using a test technique.   |
| <b>Decision table testing</b>          | a black-box test technique in which test cases are designed to exercise the combinations of conditions and the resulting actions shown in a decision table.   |
| <b>Equivalence partitioning</b>        | a black-box test technique in which test conditions are equivalence partitions exercised by one representative member of each partition. <i>After</i> ISO 29119-1. <i>Synonyms</i> : partition testing. |
| <b>Error guessing</b>                  | 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. <i>References</i> : ISO 29119-1.                             |
| <b>Experience-based test technique</b> | a test technique based on the tester's experience, knowledge, and intuition. <i>Synonyms</i> : experience-based test design technique, experience-based technique.                                      |
| <b>Exploratory testing</b>             | an approach to testing in which the testers dynamically design and execute tests based on their knowledge, exploration of the test item, and the results of previous tests. <i>After</i> ISO 29119-1.   |
| <b>State transition testing</b>        | a black-box test technique in which test cases are designed to exercise elements of a state transition model. <i>Synonyms</i> : finite state testing.   |
| <b>Statement coverage</b>              | the coverage of executable statements.  |
| <b>Test technique</b>                  | a procedure used to define test conditions, design test cases, and specify test data. <i>Synonyms</i> : test design technique.  |
| <b>White-box test technique</b>        | a test technique only based on the internal structure of a component or system. <i>Synonyms</i> : white-box test design technique, structure-based technique.   |

## 4.1 Test Techniques Overview


FL-4.1.1 (K2) Distinguish black-box, white-box and experience-based test techniques.

In this section, we describe:

- How to choose the right test technique
- What types of test techniques are described in the Foundation Level syllabus
- What are the common features of each group of test techniques

According to the definition from the *Merriam-Webster* dictionary, a technique (from Gr. technē (τέχνη)—art, craftsmanship, mastery, skill) is “a body of technical methods (as in a craft or in scientific research)” or “a method of accomplishing a desired aim.”<sup>1</sup>

In the context of software testing, this activity will consist of using certain methods to derive (produce) test conditions, test cases, and test data based on the scientific basis of software knowledge.

Let’s say right away that every **test technique**  is strictly defined, i.e., it is formal in the sense of the principles that constitute it (although in itself, as such, formal need not be, e.g., error guessing or exploratory testing). This is because behind every test technique is what Glenford Myers calls an “error hypothesis” [10]. In the context of the ISTQB® vocabulary [42], perhaps a more appropriate name would be “defect hypothesis,” but for historical reasons, we will stick with the original name (anyway, test techniques also detect defects from specific programming errors).

This error hypothesis, or, as it is sometimes called, error theory, is the scientific basis for the construction of each technique. When discussing each of them in the following subsections, we will pay particular attention to what types of problems the technique is capable of detecting. Test techniques are therefore independent of the specific technologies, tools used, or type of software under test. Each technique is built around an abstract problem related to a specific error hypothesis and designed to detect that specific type of error or defect.

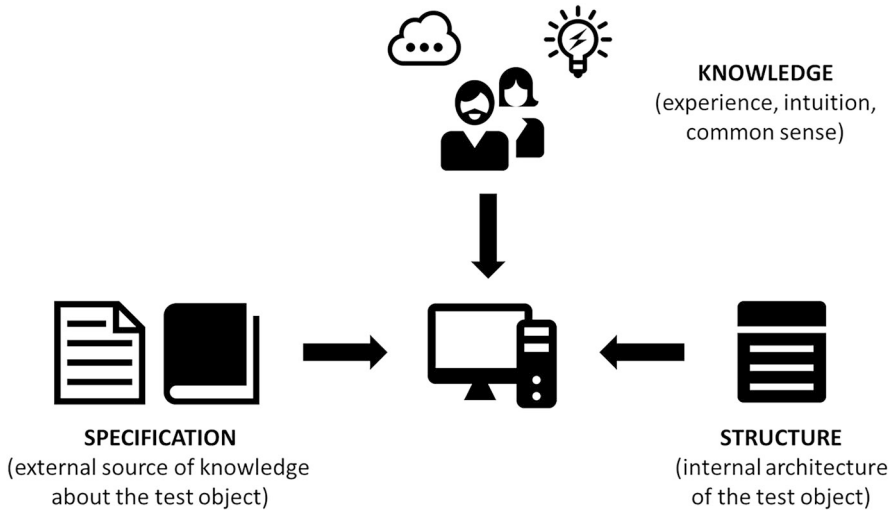
The Foundation Level syllabus discusses nine test techniques, grouping them into three categories:

- Black-box test techniques (four techniques)
- White-box test techniques (two techniques)
- Experience-based test techniques (three techniques)

This division is carried out regarding the *source of knowledge* about the test object. Figure 4.1 justifies schematically the reason for such a division of techniques. Before software is developed, it must be *designed*. The role of design documentation can be fulfilled by requirements specification, use cases, or business process

---

<sup>1</sup><https://www.merriam-webster.com/dictionary/technique>



**Fig. 4.1** Sources of knowledge about test object as basis for test technique categorization

description. On their basis, software is created, which has a specific *structure*. It is, for example, code, menu structure, business process flow structure, or another form of architecture description. In addition to these formal sources of knowledge, there are also less formal ones, related directly to the people working on software development. These people have *knowledge*, *experience*, and *intuition*, from which valuable tests can also be derived.

We will now discuss the different categories of testing techniques in more detail.

### Black-box Test Techniques

**Black-box test techniques** 📖 are also known as behavioral techniques or specification-based techniques. They use knowledge external to the test object about how it should behave. This knowledge can be, for example, requirements specification, use case description, user stories (in agile projects), and business processes. All of these models describe the desired behavior of the system, both functional and nonfunctional, without referring to its internal design.

The advantage of using black-box techniques is that the aforementioned documents generally exist long before the implementation of a component or system begins. This means that testing activities (e.g., test analysis and test design) can begin long before code development. This allows us to use black-box test techniques not only during dynamic testing but also at the test design stage, as a kind of static testing method. Black-box techniques can be used in both functional and nonfunctional testing.


### White-Box Testing Techniques

**White-box test techniques** 📖 are also called structured or structure-based techniques. The basis for white-box test design is the *internal* structure of the test object.

Most often, this structure is the source code, but it can also be another, more high-level model of the system or module architecture. For example, at the integration test level, such a model can be the so-called call graph, and at the system test level, information flow in the business process.

An attentive reader may notice that since, when testing a program, we refer to the program itself (e.g., to the source code), we arrive at a paradoxical situation in which the program becomes its own oracle, i.e., it adjudicates for itself how it should behave. In fact, this is an apparent problem. Knowledge of the program's internal structure is used only to design tests that cover specific elements of this model (e.g., code statements, decisions in the code, paths in the program). In contrast, for each test, its expected output must be determined on the basis of knowledge external to the system under test, for example, on the basis of specifications or common sense. Code can never be an oracle for itself.


### Experience-Based Testing Techniques

The group of **experience-based test techniques**  differs from the other two groups in that it is not based on any formal document: design, requirements, code, etc. This is because experience-based test techniques use somewhat more “soft” sources of knowledge about the system under test. These sources are knowledge, intuition, experience, knowledge of defects found in previous versions of the system or similar applications, etc. Thus, they refer directly to the qualities and skills of the testers themselves, rather than “objective” knowledge about the system under test.

Techniques can leverage not only the experience of testers but also of all other project stakeholders. Examples include using the expertise of developers, end users, customers, architects, business analysts, and the project manager. Experience-based test techniques are often used in combination with black-box and white-box test techniques. This gives testers a chance to detect problems that are easily overlooked by using more formal test techniques, which are unable to take into account all the nuances of the project or the context in which the software is being developed.

Table 4.1 shows the basic differences between the three groups of test techniques mentioned above.

As previously mentioned, the Foundation Level syllabus introduces nine test techniques, knowledge of which is mandatory for the exam. These include four black-box techniques, two white-box techniques, and three experience-based techniques. A summary of these techniques is shown in Fig. 4.2. The syllabus does not describe these techniques in detail nor give examples of their practical use, although the ability to apply them correctly is one of the most important qualities of a good tester. Therefore, in the following subsections, we will discuss them in *great detail*, highlighting the important aspects of each technique. To make the material more accessible, we will illustrate the use of these techniques with a lot of examples.

There are, of course, many other test techniques. The syllabus discusses only the most popular and widely used ones. Test techniques and corresponding **coverage**  measures are described in the international standard ISO/IEC/IEEE 29119-4 [4]. More information on test techniques is also provided in [16, 43–48].

**Table 4.1** Comparison of test technique categories

| Criterion of comparison                                       | Black-box  | White-box  | Experience-based  |
|---|--|--|---|
| Source for deriving test conditions, test cases and test data | Test basis external to the test object: requirements, specifications, use cases, user stories      | Test basis describing the internal structure of the test object: code, architecture, detailed design; specifications are often used to describe the expected results | Knowledge, experience, intuition of testers, developers, users, and other stakeholders      |
| Type of problems detected                                     | Discrepancies between requirements (declared behavior) and their implementation                    | Problems associated with control flow or data flow   | Depends on the person performing the tests or, for example, the checklist used in the tests |
| Coverage  | Measured against the tested elements of the test basis and the technique applied to the test basis | Measured against tested elements of the structure, such as code or interfaces  | Not defined   |

| TEST TECHNIQUES               |  |                         |
|-------------------------------|--|-------------------------|
| BLACK-BOX                     | WHITE-BOX                                | EXPERIENCE-BASED        |
| equivalence partitioning (EP) | statement testing and statement coverage | error guessing          |
| boundary value analysis (BVA) | branch testing and branch coverage       | exploratory testing     |
| decision table testing        |  | checklist-based testing |
| state transition testing      |  |                         |

**Fig. 4.2** Testing techniques as described in the Foundation Level syllabus

**Selecting the Test Technique**

Many factors influence the choice of test techniques. These can be divided into three main groups:

- Formal factors (e.g., documentation, law and regulations in force, customer contract provisions, processes in place in the organization, test objectives, SDLC model used).

- Product factors (e.g., software, its complexity, importance of various quality characteristics, risks, expected types of defects, expected use of the software).
- Project factors (e.g., available time, budget, resources, tools, skills, knowledge and experience of testers).

**Example** You are working on a project involving the development of a university library application. In your organization, there is an obligation to test the applications being developed, while the test strategy (see Sect. 5.1.1) in place in your project enforces—due to contracts signed with the client—the need to conduct testing using formal test techniques so that the design, implementation, execution, and results of the tests can be documented. The project is being conducted under the V-model.

The requirements have been collected and presented in the form of a requirements specification. Currently, the architects of the system are working on the design of the business logic of the module responsible for verification of how many books a user can borrow according to this user type (student, employee) and overdue fines. This is a key component to ensure proper implementation of the library's rules and regulations, so the risk impact of its malfunction is very high.

You have 3 days to analyze and design manual tests. Test automation is not expected due to the fact that regression testing will not play a big role in this project. In addition, the organization does not have specialized tools or the technical skills to create automated test scripts.

From the above description, it is clear that in the context of the component responsible for applying the book lending rules:

- We need to conduct tests (this follows directly from the documentation and contract).
- We need to design and document these tests (ditto).
- Testing should focus on business logic, so a sensible choice would be to use appropriate test techniques for this aspect of the software.
- The risk impact associated with this component is high, so it makes sense to use methods that check the business logic very carefully.
- Testing will be done manually (lack of tools and experience, low role of regression testing, which is usually a natural candidate for automation).

As can be seen from the above example, the decision on the choice of technique, the level of detail of analysis, the choice of test design, implementation, and execution is influenced by many formal, product, and project factors.

There is no perfect, universal test technique. Each technique has its own advantages and disadvantages. Each will perform better in one situation and worse in another. For example, in the case of the abovementioned situation with a component

**Table 4.2** Examples of test techniques with different levels of formalization

| Degree of formalization | Example  |
|-------------------------|--|
| Very low                | Unplanned, undocumented execution of error guessing, without saving test results   |
| Low                     | Conducting exploratory tests with a test charter   |
| Medium                  | Using decision table testing to test business logic; test cases are documented in test case specifications, and test results are logged  |
| Large                   | Using the state machine model and state transition testing technique to test program behavior; test design (state machine), test cases (in the form of test scripts), and test results (logs) are documented |

that verifies the rules for book lending in a library, one of the black-box testing techniques—decision table testing (see Sect. 4.2.3)—seems to be a good choice, because it tests the business logic and this is the functionality we care most about testing. On the other hand, there is no point in using, for example, white-box test techniques that are based on the internal structure of the program (source code), since the testing problem here is business logic testing (the aforementioned techniques are discussed in detail later in this chapter).

A good practice, often used by experienced testers, is to combine techniques. For example, using decision tables in the above problem, the tester can on occasion apply the boundary value analysis technique (see Sect. 4.2.2) and check business rules for such values (e.g., the maximum possible number of books a student can borrow).

The use of test techniques can also be considered in the context of test levels. Some techniques are more universal; hence, they are naturally present at all test levels—from component testing to acceptance testing. Some other techniques, on the other hand, have a somewhat more limited scope of application; for example, white-box techniques are most often applied at the component test level (e.g., developer unit tests). Of course, one can imagine examples of using this technique at the acceptance test level, but in practice, such situations are rather rare.

The degree of formalization of test development using test techniques can range from very informal to very formal. In Table 4.2, we give some examples of the varying degree of formalization.

## 4.2 Black-Box Test Techniques


- FL-4.2.1 (K3) Use equivalence partitioning to derive test cases.
- FL-4.2.2 (K3) Use boundary value analysis to derive test cases.
- FL-4.2.3 (K3) Use decision table testing to derive test cases.
- FL-4.2.4 (K3) Use state transition testing to derive test cases.



In this section, in addition to discussing the four black-box testing techniques described in the syllabus, we discuss an additional fifth one—use case-based testing (Sect. 4.2.5). This is an optional, non-examinable material. This technique was present in an older version of the syllabus. We believe that it is so important and widely used that it is worth discussing, even as super-compulsory content in relation to the syllabus.

### 4.2.1 Equivalence Partitioning (EP)

One of the seven principles of testing presented in Sect. 1.3 says that “exhaustive testing is impossible.” This is quite obvious: the number of possible combinations of input data is virtually infinite, while the tester has the ability to perform only a finite, and very small, number of tests.

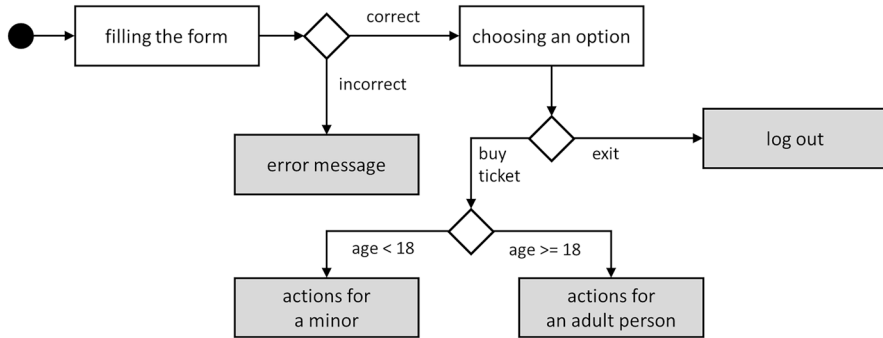
The **equivalence partitioning**  technique attempts to overcome the principle of impossibility of thorough testing. There are usually an infinite number of possible inputs, but the number of various *expected behaviors* of a program on these inputs is typically finite, especially if we consider specific behaviors related to a strictly defined aspect of the application’s operation. Let us look at some examples.

**Example** The system sets the amount of tax according to income. The tax can be 0%, 19%, 33%, or 45%. There are potentially infinitely many possible values for income but only four possible types of decision made by the system.

**Example** A user fills out a web form and then wants to print it. The printout can be one or both sides. There are potentially infinite ways to fill out a form, especially if it is complex. However, what we want to test in this case is only two types of behavior: correct one-sided printing and correct two-sided printing.

**Example** The user fills out the form, including the “age” field. The user can then buy a ticket, with the procedure being different depending on the user’s age. The exact flowchart of the process is shown in Fig. 4.3. There are many possibilities for filling out the form, but only four possible actions: error message, buy ticket by an adult, buy ticket by a minor, and logout.

So, as you can see, it is usually possible to reduce the testing behavior of a program to a *finite* number of variants. The equivalence partitioning method divides a given domain into subsets called partitions (or equivalence partitions) such that for every two elements from one partition, we have identical program behavior. For example, if the system assigns a discount to people under 18, then all numbers less than 18 will form one equivalence partition, corresponding to the assignment of the discount. In this way, from the tester’s point of view, values belonging to the same partition *are treated in a same way*. Hence, each element of a given partition is an equally good choice to test.



**Fig. 4.3** Example of a process describing possible scenarios of system use

### Application

The EP technique is versatile. It can be used in virtually any situation, at any test level, and in any test type. This is because it boils down to the division of possible data into groups. It is worth mentioning that the technique can be applied not only to input domains but also to output domains and internal domains (i.e., those related to variables that are neither directly given in the input nor returned in the output).

The domain that we divide into equivalence partitions does not have to be a numerical domain. It can in fact be *any nonempty set*. Here are some examples of domains to which the equivalence partitioning technique can be applied:

- A set of natural numbers (e.g., partition into even and odd numbers)
- A collection of words (e.g., partition by word length, one-letter, two-letter, etc.)
- Collections relating to time (e.g., partition by year of birth, by month in a given year, etc.)
- A collection of operating system types: {Windows, Linux, macOS} (e.g., partition into single-element partitions, {Windows}, {Linux}, {macOS})

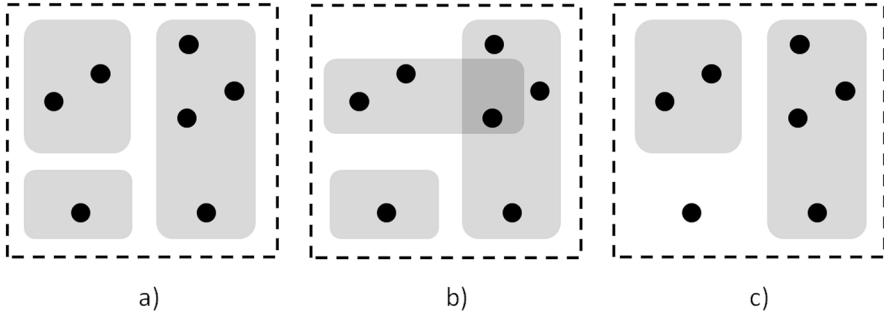
### Partitioning correctness

It is very important that the partitioning we make is correct, which means that:

- *Each* element of the domain belongs to *exactly one* equivalence partition
- No equivalence partition is empty

Figure 4.4 illustrates the issue of correctness of partitioning. The black dots indicate the elements of the domain, and the gray rectangles indicate the equivalence partitions. Figure 4.4a shows the correct equivalence partitioning, which meets both of the above conditions. Figure 4.4b shows an incorrect partitioning—one element belongs to two equivalence partitions, which violates the correctness conditions. Figure 4.4c shows another incorrect partitioning—one of the elements of the domain has not been assigned to any equivalence partition.

The issue of partitioning correctness seems quite obvious, but in practice, it is not a trivial problem. In real applications, domains and partitions may be very complex



**Fig. 4.4** Examples of correct (a) and incorrect (b, c) domain partitioning

and have a very complicated structure. It often happens that for a partitioning we have done, some correctness conditions are violated.

**Example** Consider the classic testing problem described by Myers [10]. The program takes three non-negative integers,  $a$ ,  $b$ ,  $c$ , as input and outputs the type of triangle that can be constructed from segments of the given lengths. Possible answers of the program are “equilateral triangle,” “isosceles triangle,” “scalene triangle,” and “not a triangle.” Suppose we want to apply the EP technique to the input domain (i.e., to the set of all possible triples  $(a, b, c)$  of non-negative integers), regarding the output domain (i.e., the type of triangle). It seems natural to divide the input domain into the four partitions corresponding to the four possible outputs mentioned earlier. However, upon closer inspection, it turns out that every equilateral triangle is also an isosceles triangle, so we are dealing with an equivalence partition, which is a proper subset of another partition. We need to correct our partitioning, by defining the following partitions:

- Inputs representing equilateral triangles
- Inputs representing isosceles triangles, *that are not equilateral*
- Inputs representing scalene triangles
- Inputs falling into the “not a triangle” category

Now, this partitioning is correct: each triple of numbers given to the input corresponds to exactly one of the above four possible partitions.

**Example** We want to classify the set of all possible finite sequences of numbers regarding their ordering. A natural equivalence partitioning of such data into equivalence partitions could look like this: ascending sequences, descending sequences, and unordered sequences. But note that a one-element sequence is both an ascending and a descending sequence. In addition, the question arises as to where to classify the empty string (containing 0 elements). Therefore, the correct partitioning must take into account these “edge cases,” and the partitions can look like this:

- Partition 1: the empty sequence
- Partition 2: all one-element sequences
- Partition 3: all ascending sequences with at least two elements
- Partition 4: all descending sequences with at least two elements
- Partition 5: all unordered sequences with more than two elements

Partitions that contain “normal,” “correct” values, i.e., values expected (accepted) by the system, are called *valid partitions*. Partitions that contain values that the component or system should reject (e.g., data with incorrect syntax, exceeding acceptable ranges, etc.) are called *invalid partitions*.

In our last example, the five partitions we have identified are valid, because each of them contains correct data expected by the system (perhaps the case of the empty string is controversial—if the specification does not explicitly mention *non-empty* strings, then partition 1 may be considered an invalid partition). In addition to these identified partitions, we can distinguish, for example, an invalid partition consisting of elements that are not numeric strings (e.g., they contain alphanumeric characters).

The definitions of valid and invalid partitions, and consequently valid and invalid values, can be understood in different ways, so if we are going to use these terms, it is worth defining them precisely. For example, valid values can be understood in at least two ways:

- As those that should be processed by the system
- As those for which the specification defines their processing

Similarly, incorrect values can be understood:

- As those that should be ignored or rejected by the system
- As those for which the specification does not define their processing

**Example** In the user registration form, the system expects a valid e-mail address to be entered in the “e-mail” field. When the user enters the character string “abc@def@ghi” there, the system rejects this input as invalid, informing the user with the message “Incorrect e-mail.” In this situation, the input “abc@def@ghi” can be treated by some as an invalid value (because the system rejected it, registration will take place only if the correct e-mail address is provided) and by others as a value coming from the valid partition (because the system is prepared for this type of situation, the proof of this is the error message; so it is in a sense an “expected” value and, therefore, coming from the valid partition).

### Further Division of Partitions into Subpartitions

An important advantage of the EP technique is that if you divide any (even all) partitions into subpartitions, you will still have equivalence partitioning. The testing for such a more fragmented domain will be more accurate. The tester can decide that certain partitions should be further subdivided for certain reasons. This is a natural approach that takes advantage of the hierarchical nature of partitions. Consider the following example.


**Example** PESEL is the personal citizen identification number used by Polish government. Each Polish citizen has a unique PESEL number assigned. It is an 11-digit number, in which first six digits encode the birth date, the last one is the check digit, and the evenness of the penultimate digit encodes gender—male or female.<sup>2</sup> The system takes the PESEL number from the user and, depending on whether the person is of age or not, takes the appropriate action. We want to perform equivalence partitioning for all possible sequences of digits. As a first step, we can divide them into valid and invalid numbers, that is, numbers that represent valid PESEL numbers and everything else. Next, we can consider each of these partitions in terms of further possible division. Correct PESEL numbers should undoubtedly be divided—according to the specification—into numbers corresponding to adults and minors. A further division of each of these partitions could, for example, take into account the appropriate coding of the month number (e.g., for those born between 1900 and 1999, the month is coded normally; for those born between 2000 and 2099, 20 is added to the month number (so, e.g., February is coded as 22); for 2100 to 2199, 40 is added; for 2200 to 2299, 60 is added; and for 1800 to 1899, 80 is added). In turn, the PESEL number may be incorrect for various reasons, such as structural (e.g., wrong length) or substantive (e.g., check number inconsistency). An example of the process of dividing partitions into subpartitions is shown in Fig. 4.5.

Let us note that people born between 2000 and 2099 can be both adults and minors (assuming that we make the division in 2024), so we divided this time frame into 2000–2006 and 2007–2099. We assume, for testing purposes, that people born after 2024 (which is obviously impossible in 2024) are classified as minors. In testing, we will only be concerned with checking whether PESEL numbers with the correct month coding are accepted, and we want to check this for all such codings defined in the PESEL specification (1800–1899, 1900–1999, 2000–2099, 2100–2199, 2200–2299). Finally, as a result of the hierarchical division of partitions, we divided the domain related to PESEL numbers into 12 equivalence partitions:

- Three valid partitions for adults (1800–1899, 1900–1999, 2000–2006)
- Three valid partitions for minors (2007–2099, 2100–2199, 2200–2299)
- Three partitions invalid due to the syntax (structure) of the PESEL number
- Three partitions invalid due to the semantics (meaning) of the PESEL number

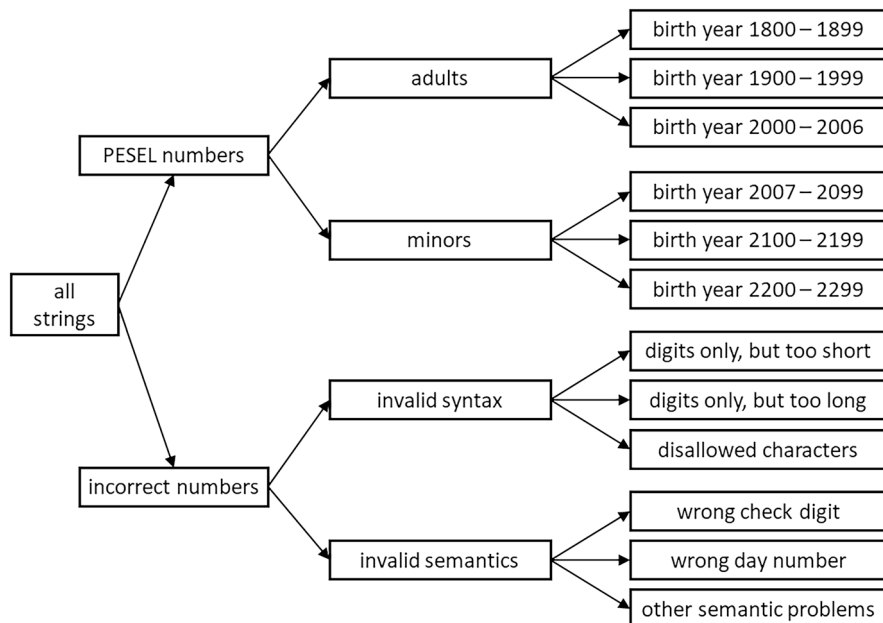
Notice that we could go further with the division. For example, each valid equivalence partition could be divided into two partitions, encoding “male” and “female” genders.

### Deriving Test Cases

In the case of the EP technique, the **coverage items**  are equivalence partitions. The minimum set of test cases to ensure 100% coverage is one that covers every equivalence partition, i.e., for each identified partition, there is a test case containing

---

<sup>2</sup>The PESEL system is quite old; it was introduced in the 1970s and therefore did not take into account genders other than male and female.



**Fig. 4.5** Example of hierarchical equivalence partitioning

a value from that partition. In the one-dimensional case (one domain and one division), the minimum number of test cases should therefore be as many as the equivalence partitions we have identified. In the multidimensional case (more than one domain), the matter gets a bit more complicated, since the number of test cases will depend, for example, on the way we treat the combinations of invalid partitions, as well as on possible dependencies or constraints between values and partitions from different domains. Coverage is measured as the number of equivalence partitions tested using at least one value divided by the total number of equivalence partitions defined and is usually expressed as a percentage.

**Example** Let us consider the PESEL verification system again. Assume that the partitioning looks like in Fig. 4.5 and that we have defined the following test cases:

- PESEL of an adult born in 1898
- PESEL of an adult born in 1999
- PESEL = “1234” (too short)

This set of three test cases covers 3 of the 12 identified equivalence partitions (see Fig. 4.5), so it achieves coverage of  $3/12 = 1/4 = 25\%$ .

### Covering Multiple Equivalence Partitions Simultaneously: Defect Masking

A slightly different approach is needed in a situation where our tests are to simultaneously cover equivalence partitions derived from more than one domain. In such

a situation, it is good practice not to create test cases that will cover two or more invalid partitions. This has to do with so-called defect masking. The recommended strategy is as follows:

1. First, create the smallest possible number of test cases composed only of test data from valid partitions, which will cover all valid partitions from all domains.
2. Then, for each uncovered invalid partition, create a separate test case in which data from that partition will occur, and all other data will come from the valid partitions.

Consider the following example to illustrate this approach.

**Example** The system gives a grade to a student on the basis of two data: the number of points for exercises (0–50) and for the exam (0–50). The student passes the course if the total score exceeds 50. Thus, the input data of the test case will consist of two parts: points for exercises and points for the exam. Let us assume that we have distinguished the following domains and their partitions:

Domain of variable A = “exercise points”:

- (A1) invalid partition: numbers less than 0
- (A2) valid partition: numbers from 0 to 50
- (A3) invalid partition: numbers greater than 50

Domain of variable B = “exam points”:

- (B1) invalid partition: numbers less than 0
- (B2) valid partition: numbers from 0 to 50
- (B3) invalid partition: numbers greater than 50

We want to cover all equivalence partitions of the two domains, A and B. The valid partitions are A2 and B2. All others (A1, A3, B1, B3) are invalid partitions. Each test case contains as input data some number of exercise points and some number of exam points, so a single test case always covers one partition from domain A and one from domain B. Following the strategy described above, we cover the valid partitions first. Since we have only one valid partition in each domain, one test case is enough, e.g.:

TC1: A = 25, B = 30 (covers valid partition A2 and valid partition B2).

There are two invalid partitions left to cover from A and two from B. So we need four more test cases in which these partitions will be tested individually (the other partition in a given test case must be the valid partition):

TC2: A = -8, B = 35 (covers invalid partition A1; additionally covers B2),

TC3: A = 48, B = -11 (covers invalid partition B1; additionally covers A2),

TC4: A = 64, B = 4 (covers invalid partition A3; additionally covers B2),

TC5: A = 12, B = 154 (covers invalid partition B3; additionally covers A2).

If we were testing a situation in which both values came from invalid partitions, the phenomenon of *defect masking* could occur. Suppose the system verifies that a

student has passed a subject through the following procedure (described in pseudocode):

```

INPUT: ExercisesPoints, ExamPoints
IF (ExercisesPoints + ExamPoints > 50) THEN
    RETURN "Course passed."
ELSE
    RETURN "Course failed."

```

Now consider the following test case:

TC6:  $A = -28$ ,  $B = 105$  (covers invalid partitions A1 and B3).

In this situation, the total score will be  $-28 + 105 = 77$ , and therefore, the system will return a result of "Course passed," despite the fact that both inputs are incorrect!

### "Each Choice" Coverage

The "each choice" coverage is applied to the multidimensional case, i.e., the case in which there is more than one domain, and each test case covers one partition from the distribution of each domain. This coverage is one of the simplest (and weakest) coverage types applied to the multidimensional case. It requires that each partition of each domain be tested at least once. In practice, using this method, the tester tries to make the next test case cover as many previously uncovered coverage items as possible.

**Example** We are testing a COTS product for general sale. This means that we need to test it in different environments. The program works with different operating systems and different browsers. It is therefore necessary to test the operation of different browsers under different operating systems. Let us assume that we have the following four browsers to test:

- Google Chrome (GC)
- Firefox (F)
- Safari (S)
- Opera (O)

and the following three operating systems:

- Windows (W)
- Linux (L)
- iOS

For simplicity, we do not include specific versions of operating systems or browsers. Each browser type forms a one-element equivalence partition, resulting in four partitions: {GC}, {F}, {S}, and {O}. Each operating system, in turn, creates a one-element equivalence partition, making a total of three partitions: {W}, {L}, and {iOS}.



According to the “each choice” coverage criterion, for each identified partition, there must be a test case covering a value from that partition. In our example, the test cases are represented by a pair of test data (browser type, operating system).

In our example, four test cases are sufficient to meet the coverage criterion, for example:

TC1: (GC, W)

TC2: (F, L)

TC3: (S, iOS)

TC4: (O, W)

### Types of Problems Detected

The EP technique identifies problems resulting from faulty data processing, i.e., resulting from errors in the domain model.

#### Defining the Domain Is Key

The EP technique is always applied to *a specific* domain that models the problem. Sometimes, the choice of the domain is obvious, but sometimes, the matter can be a bit more complicated. Consider the following fairly typical example.

**Example.** The thermostat turns off the heating when the temperature (calculated in full degrees) exceeds 21 degrees and turns on when the temperature drops below 18 degrees. Design test cases using equivalence partitioning.

How to apply the EP technique to this problem? How many equivalence partitions will there be to check? That depends on *what specific property* of the system we want to check. We can approach our problem in at least three ways:

Method 1. Analyzing only the domain, we notice that for values 22 and above, the system turns off the heating; for values 17 and below, it turns on; and for values between 18 and 21, it takes no action. Therefore, we have three equivalence partitions for the “temperature” domain, up to 17 degrees, from 18 to 21 degrees, and above 21 degrees (see Fig. 4.6a), and to cover them, we need three test cases, e.g.:

- Temperature = 15 (expected output: heating on).
- Temperature = 20 (expected output: heating state does not change against the previous state).
- Temperature = 22 (expected output: heating off).

Method 2. Note that in the temperature range 18–21, heating can be both on and off. Thus, we consider a domain composed of pairs (temperature, heating state). In this situation, we have four possibilities:

- Temperature <18, heating on
- Temperature 18–21, heating on

(continued)

- Temperature 18–21, heating off
- Temperature >21, heating off

So we have four situations to cover (see Fig. 4.6b). The situation is now a bit more complicated than before, because before test execution, we need to force the appropriate heating condition for temperatures of 18–21. The test cases covering four above mentioned partitions could therefore look like this:

- TC1: temperature = 15 (expected output, heating on).
- TC2: temperature = 18 after rising from 17 (expected output, heating is still on).
- TC3: temperature = 21 after dropping from 22 (expected output, heating is still off).
- TC4: temperature = 22 (expected output, heating off).

Method 3: We can consider not static pairs (temperature, heating state) but *transitions* between such pairs. Then our domain will describe possible transitions between pairs (temperature, heating state) and will consist of six possible elements (in the following list of domain elements, the numbers denote temperatures, and OFF and ON denote heating off and heating on, respectively):

- (17, ON) → (18, ON)
- (18, ON) → (17, ON)
- (18, OFF) → (17, ON)
- (21, ON) → (22, OFF)
- (21, OFF) → (22, OFF)
- (22, OFF) → (21, OFF)

This situation is shown in Fig. 4.6c. So we need six test cases to simulate these transitions, e.g., for the first element, the initial configuration of the system is 17 degrees, and the heating state is on. The test is to increase the temperature to 18 degrees and see if the heating is turned off.

Note that we could add four more transitions to our list, involving staying in the same domain, if we don't want to limit ourselves to changes involving transitions between different temperature ranges that form equivalence partitions of the "temperature" domain:

- (17, ON) → (16, ON)
- (19, ON) → (20, ON)
- (19, OFF) → (18, OFF)
- (22, OFF) → (23, OFF)

So, as you can see from the above example, the test objective significantly affects the form of the domain. In our case, the domain was modeled in three

(continued)


different ways: as a set of numbers representing temperatures, as a set of pairs (temperature, heating state), and as a set of transitions between such pairs. Applying the EP technique to each of these resulted in verifying a significantly different aspect of the system. Therefore, we always need to know exactly what is the form of the domain we will be working on, before applying this technique.

Note also that sometimes, the need to specify precisely the *expected* test result allows us to detect errors or ambiguities in the specification. In the above method 1, for data from partition 18–21, it is not very clear what the expected result is supposed to be: whether the thermostat is supposed to be on or off. This may indicate that the problem we try to solve is ill posed.

At the end of the analysis of this example, let us note one more very important thing. Namely, for this particular problem, the equivalence partitioning technique is not a good choice (the problems we encountered when applying it illustrate this very well). This is because the EP focuses on detecting problems in the implementation of the (static) domain, and the fundamental issue in the testing problem we are considering here is verification of the *behavior* of the thermostat. It seems that a more appropriate technique would be state transition testing (see Sect. 4.2.4).

## 4.2.2 Boundary Value Analysis (BVA)

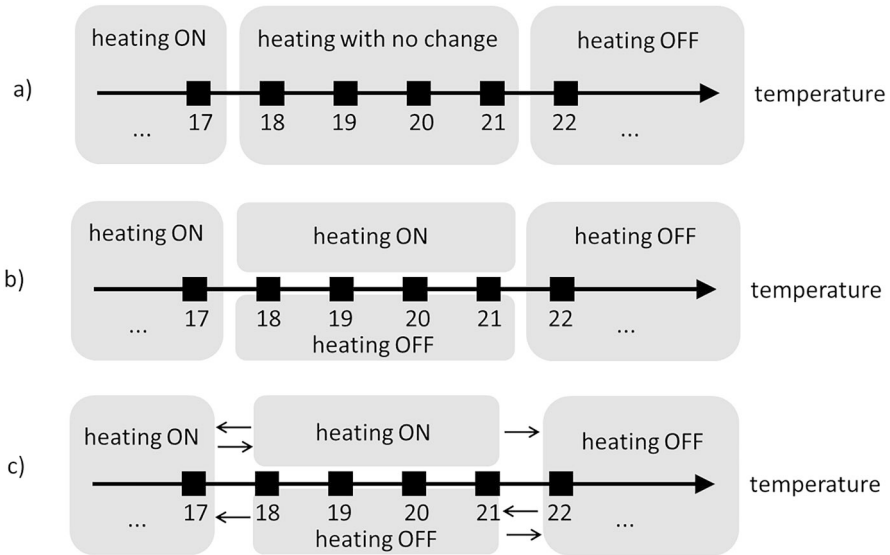
### BVA as an Extension of the EP

**Boundary value analysis**  is a technique built on the equivalence partitioning technique. So its versatility and the type of problems detected will be similar to the latter. The difference with equivalence partitioning is that in BVA, we select very specific elements of equivalence partitions for testing, namely, those that lie on *the boundaries* of these partitions.

### Boundary Values: For Which Domains Is the BVA Applicable?

A boundary value of a partition is the smallest or the largest element of that partition. Speaking of “smallest” and “largest” makes sense only if we define an *order* relation on the elements of the domain. Thus, the BVA technique can be applied only to domains whose elements are ordered in terms of some *order relation* (e.g., to sets of numbers, with a “<” relation). Examples of such domains are sets of natural, integer or real numbers, values relating to time or date, etc.

Boundary values are always defined for a specific equivalence partition. For example, if the “age” domain, containing the natural numbers between 1 and 120, is divided into two equivalence partitions, {1, 2, ..., 18} (child) and {19, 20, ..., 120} (adult), then the boundary values of the “child” partition are 1 (the smallest) and 18 (the largest). The boundary values for the “adult” partition are 19 and 120.



**Fig. 4.6** Three different approaches to selecting a domain for equivalence partitions

In addition, we must assume one more thing: the considered equivalence partitions cannot have “gaps,” i.e., formally speaking, if two values  $a$  and  $b$  such that  $a < b$  belong to the same equivalence partition, then all intermediate elements  $c$ , i.e., such that  $a < c < b$ , must also belong to the same partition. Why? Consider the previously mentioned example but drop the element 4 from the “child” partition. This partition has the form  $\{1, 2, 3, 5, 6, \dots, 18\}$ . Its extreme values are, of course, still 1 and 18, but the question arises: aren’t the values 3 and 5 also “boundaries” of this partition? After all, both are adjacent to an element that does not belong to the partition! In this case, then, we would have to split our partition into two,  $\{1, 2, 3\}$  and  $\{5, 6, \dots, 18\}$ , and the value of 4 would be a separate partition. So, the domain would be partitioned into four partitions:  $\{1, 2, 3\}$ ,  $\{4\}$ ,  $\{5, 6, \dots, 18\}$ , and  $\{19, \dots, 120\}$ .

### Deriving Test Cases with BVA

The general procedure for deriving test cases using the BVA technique is as follows:

1. Identify the domain you want to analyze.
2. Perform equivalence partitioning of this domain into equivalence partitions.
3. For each equivalence partition identified, determine its boundary values (note—sometimes the analysis can be limited only to certain partitions and may not have to take into account all the equivalence partitions determined).
4. For each boundary value, determine the coverage items (the elements to be tested) for this boundary value.

The first two steps simply apply the EP technique. In step 3, we identify the boundaries of the partitions. In step 4, based on the identified boundary values, we

determine the elements to be used in test cases as test input data. In the BVA technique, the boundary values (test conditions) are not necessarily the same as the elements to be selected for testing (coverage items). This will depend on which partitions we are considering and on the chosen variant of the BVA method. This is because there are two main variants of the BVA—the so-called 2-value BVA and 3-value BVA. Let us discuss them in detail.

### 2-Value BVA

In the 2-value BVA [10, 43], for each identified boundary value, that value and its nearest neighbor *not belonging* to the partition to which the boundary value belongs are selected for testing. For example, if we consider boundaries of the partition  $P = \{1, 2, 3, 4, 5, 6\}$  in the integer domain (i.e., the values 1 and 6), we select for testing:

- Value of 1 (as a boundary value of  $P$ )
- Value of 0 (as the nearest neighbor of 1 not belonging to  $P$ )
- Value of 6 (as a boundary value of  $P$ )
- Value of 7 (as the nearest neighbor of 6 not belonging to  $P$ )

### 3-Value BVA

In the 3-value BVA [45, 49] for each identified boundary value, we take that value and *both* of its neighbors for testing, regardless of which partitions they belong to. In the example above, we would select for testing:

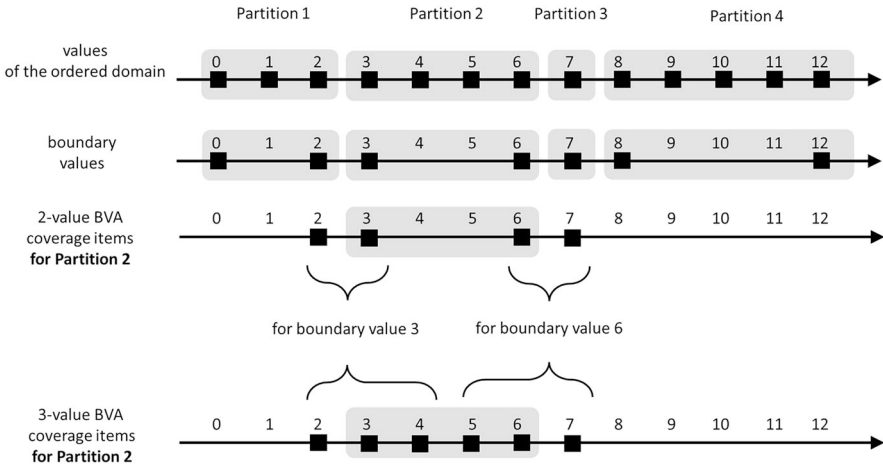
- Value of 1 (as a boundary value of  $P$ )
- Value of 0 (as the left neighbor of 1)
- Value of 2 (as the right neighbor of 1)
- Value of 6 (as a boundary value of  $P$ )
- Value of 5 (as the left neighbor of 6)
- Value of 7 (as the right neighbor of 6)

### Comparison of 2-Value and 3-Value BVA

Schematically, the principle of determining the boundary values, together with associated coverage items, is shown in Fig. 4.7. A special case may be a one-element partition, but the principle works analogously. The single element of this partition is both its smallest and largest element. In the example in the figure, partition 3 contains only the value 7. In the case of the 2-value BVA, the values taken for testing would be as follows:

- A value of 7 as the *minimum* boundary value and a value of 6 as an out-of-partition neighbor
- A value of 7 as the *maximum* boundary value and a value of 8 as an out-of-partition neighbor

So we would take the set of values 6, 7, and 8 for testing. Similarly, for the 3-value BVA, the values for testing are 6, 7, and 8 determined by 7 as the minimum value and the same values (6, 7, 8) determined by 7 as the maximum value. Thus, for testing, we would take a set of the same values as in the 2-value BVA: 6, 7, and



**Fig. 4.7** Determination of boundaries and coverage items in the two- and three-point versions of the AWB method

8. This is the case only for single-element partitions. For partitions with at least two elements, the 3-value BVA will always give us more elements to test than the 2-value BVA.

Let us now consider some examples of the application of the BVA technique.

**Example** The system assigns a ticket discount to passengers under 18 (children discount) and over 65 (senior discount). A passenger between the ages of 18 and 65 is not entitled to a discount. We want to check the correctness of the allocation of the discount. Let us carry out the four-step procedure described above to identify the coverage items.

Step 1. Identify the domain. The variable to be analyzed is the age of the passenger, which is a non-negative integer. The domain is therefore of the form  $\{0, 1, 2, 3, \dots\}$ .

Step 2. Identify equivalence partitions. From the specification, we identify the following equivalence partitions:

- P1: age eligible for children discount  $\{0, 1, 2, \dots, 17\}$
- P2: age eligible for a regular ticket  $\{18, 19, \dots, 64, 65\}$
- P3: age eligible for senior discount  $\{66, 67, 68, \dots\}$

Step 3. Identify boundary values:

- The boundary values of P1 are 0 and 17.
- The boundary values of P2 are 18 and 65.
- The boundary value of P3 is 66 (in this example, P3 does not have the largest value).

Step 4. Identify the values to be tested. In case we want to apply the 2-value BVA for all equivalence partitions, in fact, all the identified boundary values should be taken for testing, and that is enough. This is because we have a kind of symmetry between any two adjacent boundary values of two partitions. For example, 65 as a boundary value of P2 has a neighbor 66 from outside the partition, which is also a boundary value of P3, and its neighbor from another partition is 65. Therefore, using the 2-value BVA, for testing, we select all the identified boundary values: 0, 17, 18, 65, and 66. We assume here that the value  $-1$  is infeasible.

In the case of the 3-value BVA, we choose for testing 0, 1, 16, 17, 18, 19, 64, 65, 66, and 67 because:

- For the boundary value 0, we take this value and its neighbor: 0 and 1 (we assume that the value  $-1$  is infeasible).
- For the boundary value 17, we take this value and its neighbors: 16, 17, and 18.
- For the boundary value 18, we take this value and its neighbors: 17, 18, and 19.
- For the boundary value 65, we take this value and its neighbors: 64, 65, and 66.
- For the boundary value 66, we take this value and its neighbors: 65, 66, and 67.

Of course, some values are repeated, but we take each value for testing only once. If only the middle partition (“normal ticket”) were subject to our analysis, we would be limited to identifying the boundaries of this partition only, so the values 18 and 65. So in the 2-value BVA, we would choose the values 17, 18, 65, and 66 for testing, and in the 3-value BVA, we would choose the values 17, 18, 19, 64, 65, and 66.

**Example** At some place in the code, a decision is made, depending on the value of the integer variable  $x$ . The decision should be of the form:

```
IF  $x < 16$  THEN
    Perform ACTION 1
ELSE
    Perform ACTION 2
```

In this case, there are two equivalence partitions: the first one contains values that cause the execution of “ACTION 1,” which are integers less than 16; the second one is its complement, i.e., the values greater than or equal to 16. They cause the execution of “ACTION 2.” The first partition’s boundary is 15, and the second partition’s boundary is 16. The variable  $x$  has neither the smallest nor the largest value, so the first partition has no minimum boundary value, and the second partition has no maximum boundary value. Thus, in the 2-value BVA, we will choose 15 and 16 for testing, and in the 3-value BVA, we will choose 14, 15, 16, and 17.

In practice, usually the globally largest/smallest values exist—these are, for example, the ranges accepted by fields of the corresponding type (e.g., a field accepts a number consisting of up to five digits) or the values of corresponding variables. For example, a variable of type `int` (integer) in C++ has a range from  $-2^{31}$  to  $2^{31} - 1$ , i.e., from  $-2,147,483,648$  to  $2,147,483,647$ . A good tester will therefore check not only the partition boundaries set by *the specification* but also the boundaries set by *the architecture* of the solution under testing, e.g., the range of variable values.

### Careful Determination of Boundary Values

In the case of the BVA, one must be very careful about how partition boundaries are defined. Suppose we are working in the domain of natural numbers. Then, for example, the formulation “the partition contains elements not less than 7” means that the smallest number belonging to this partition is 7. In turn, the formulation “the partition contains elements greater than 7” means that the smallest value from this partition is 8. Similarly, “at most 65” means numbers up to and including 65, and the condition “ $x < 65$ ” means that the largest value satisfying this inequality is 64. The phrase “the partition contains the values ranging from  $x$  to  $y$ ” means that the partition contains the values from  $x$  to  $y$  *including  $x$  and  $y$* . Those with doubts about this are encouraged to consider the problem: on what days is the store open, on which hangs a piece of paper stating: “the store is open from Monday to Friday?” Is the store open 5 days a week from Monday to Friday or only from Tuesday through Thursday?

### Application

The BVA method is very simple yet extremely effective in detecting specific types of defects. The reason for this is that developers often make what are called “errors by one,” incorrectly implementing equivalence partition boundaries. Here are two examples of typical developer mistakes resulting in defects detectable by the BVA:

- The developer should have implemented the condition “if  $x < 10$ ” (strict inequality) but mistakenly implemented it as “if  $x \leq 10$ ” (weak inequality).
- The developer should initialize the loop control variable (iterator) as “for  $i=0$  to 10” but mistakenly assumed that the elements of the array are indexed from 1 and wrote it as “for  $i=1$  to 10”.

In the first case, the equivalence partitions, according to the specification, should look like this:  $\{..., 8, 9\}$ ,  $\{10, 11, ...\}$ . However, the incorrect implementation divided the domain (due to the realized control flow) as follows:  $\{..., 9, 10\}$ ,  $\{11, 12, ...\}$ . Analyzing this example with the BVA, we see that the boundary values (valid, according to the specification) are 9 and 10. If we use the 2-value BVA, we take these very values for testing. In the case of the value 10, according to the specification, the program should take the path corresponding to the “false” branch in the decision “if  $x < 10$ ,” but in the incorrect implementation, the decision “ $x \leq 10$ ” is true, so the control of the program will take the wrong path. There is a good chance of detecting this defect with test case that uses 10 as the test data value for  $x$ .

The 3-point BVA is stronger than the 2-point BVA, i.e., there are situations in which the 2-value BVA has no chance of detecting a failure, while the 3-value BVA can trigger a failure. Consider the following example.

**Example** A developer implements a component for controlling the temperature in a laboratory room. The temperature—measured in full degrees—must not exceed  $10^{\circ}$  C. When this threshold is exceeded, the cooling mechanism is activated. The business logic is as follows:



```
IF the temperature does not exceed 10°C THEN
    Do nothing
ELSE
    Turn on cooling
```

The developer has implemented the correct decision IF  $x \leq 10$  THEN ... incorrectly, as IF  $x == 10$  THEN ... . According to the specification, the domain partitioning should be {..., 9, 10}, {11, 12, ...}. However, according to the incorrect implementation, the partitioning is as follows, {..., 8, 9}, {10}, {11, 12, ...}, with the middle partition giving the truth value and the left and right partitions giving the false value of the decision in the incorrect implementation.

If we use the 2-value BVA, we identify the values 10 and 11 as boundary values and test only them. For the value 10 both in the expected and in the incorrect implementation, the decision is true: for it is true that  $10 \leq 10$  and that  $10 == 10$ . On the other hand, for the value 11, both decisions are false: for it is not true that  $11 \leq 10$ , and it is not true that  $11 == 10$ . Thus, neither of these two tests will be able to detect the defect—the control flow of the program (containing the incorrect implementation of the decision!) in both cases will go along the correct path—the one that results from the expected implementation! This is because for 10 °C, the program will not perform any action, and for 11 °C, it will turn on cooling. Thus, the 2-value BVA will not be able to detect a defect in the code.

However, if we use the 3-value BVA, we will have to take four values for testing: 9, 10, 11, and 12. In particular, the value of 9 differentiates execution paths according to correct and incorrect implementation: in correct implementation, the decision  $9 \leq 10$  is true, and in incorrect implementation, the decision  $9 == 10$  is false. Thus, there is a chance that for  $x = 9$ , we will detect incorrect program operation due to the selection of the wrong control flow path. After all, according to the specification, for 9 °C, the program is supposed to take no action but will turn on cooling. The above considerations are summarized in Table 4.3.

### Values Outside the Domain

Finally, let us consider one more problem. In the example with the assignment of a ticket discount, we indicated that for a boundary value of 0, we do not take its neighbor -1 for testing, because we assumed that it is infeasible, i.e., the user is not able to enter incorrect, negative values. In practice, however, this does not always have to be the case. If, for example, the user enters an age value into a form field from the keyboard, they can, of course, enter a negative value. Whether or not to test extreme, out-of-domain boundary values depends on whether the interface allows it. If it does, the tester should of course test for them.

### Coverage

The coverage items in the BVA are the boundary values of the equivalence partitions (in the 2-value BVA) or the boundary values along with all their neighboring values (in the 3-value BVA). For 2-value BVA, the coverage is therefore defined as the quotient of the number of tested boundary values and the total number of identified boundary values. For the 3-value BVA, it is the quotient of the number of tested


**Table 4.3** Differences between 2-value and 3-value BVA

|  | 2-Value BVA  |                    | 3-Value BVA                              |                   |                    |                    |
|--|--|--------------------|--|-------------------|--------------------|--------------------|
|  | Identified boundary values (and values for testing): |                    |  |                   |                    |                    |
|  | x = 10   | x = 11             | x = 9                                    | x = 10            | x = 11             | x = 12             |
|  | Test results   |                    |  |                   |                    |                    |
| Specification:<br>x ≤ 10               | (10 ≤ 10)<br>TRUE                                    | (11 ≤ 10)<br>FALSE | (9 ≤ 10)<br>TRUE                         | (10 ≤ 10)<br>TRUE | (11 ≤ 10)<br>FALSE | (12 ≤ 10)<br>FALSE |
| Incorrect<br>implementation:<br>x = 10 | (10 = 10)<br>TRUE                                    | (11 = 10)<br>FALSE | (9 = 10)<br>FALSE                        | (10 = 10)<br>TRUE | (11 = 10)<br>FALSE | (12 = 10)<br>FALSE |
| Result:                                | Test<br>passed                                       | Test<br>passed     | <b>Test<br/>failed</b>                   | Test<br>passed    | Test<br>passed     | Test<br>passed     |
| Interpretation:                        | Both tests passed, defect<br>not detected            |                    | There is a test that detected the defect |                   |                    |                    |

boundary values with their neighbors and the total number of identified boundary values and their neighbors. This is because in the 3-value BVA, we take not only boundary values for testing but also values from inside the partitions. For example, for the partition {2, 3, 4, 5, 6}, the values taken for testing in the 3-point BVA will be, in particular, the numbers 3 and 5, which are not the boundary values for this partition.

4.2.3 Decision Table Testing

Application

**Decision table testing**  is a technique that is used to verify the correctness of *business rules* implementations. A business rule usually takes the form of a logical implication:

**IF** (condition) **THEN** (action) ,

where both *condition* and *action* can be composed of more than one factor. Then we are dealing with a *combination* of conditions or actions. Here are some examples of business rules:

- **IF** (customerAge < 18) **THEN** (assign a ticket discount);
- **IF** (a+b>c>0 AND a+c>b>0 AND b+c>a>0) **THEN** (you can build a triangle with sides of length a, b, c);
- **IF** (monthlySalary > 10000) **THEN** (grant bank loan AND offer a gold card).

Decision tables allow us to systematically test the correctness of the implementation of combinations of conditions. This is one of the so-called combinatorial

**Table 4.4** Example decision table

|                           | Business rules 1–8 |     |     |     |     |     |     |    |
|---------------------------|--------------------|-----|-----|-----|-----|-----|-----|----|
|                           | 1                  | 2   | 3   | 4   | 5   | 6   | 7   | 8  |
| <b>Conditions</b>         |                    |     |     |     |     |     |     |    |
| Has a loyalty card?       | YES                | YES | YES | YES | NO  | NO  | NO  | NO |
| Total amount > \$1000?    | YES                | YES | NO  | NO  | YES | YES | NO  | NO |
| Shopping in last 30 days? | YES                | NO  | YES | NO  | YES | NO  | YES | NO |
| <b>Actions</b>            |                    |     |     |     |     |     |     |    |
| Granted discount          | 10%                | 5%  | 5%  | 0%  | 0%  | 0%  | 0%  | 0% |

techniques. For more information on these techniques, see the Advanced Level—Test Analyst syllabus [28].

### Construction of the Decision Table

We will describe the construction of the decision table using an example (see Table 4.4).

The decision table consists of two parts, describing conditions (upper part) and actions (lower part), respectively. The individual columns describe business rules. Table 4.4 describes the rules for assigning a discount on purchases depending on three factors describing the customer in question:

- Does the customer have a loyalty card? (YES or NO)
- Does the total amount of purchases exceed \$1000? (YES or NO)
- Has the customer made purchases in the last 30 days? (YES or NO)

Based on the answers to these questions, a discount is assigned: 0%, 5%, or 10%.

**Example** A customer has a loyalty card and has so far made purchases of \$1250, and the last purchases took place 5 days ago. This situation corresponds to rule 1 (has a loyalty card, total amount >\$1000, shopping in the last 30 days). So, the system should assign a 10% discount to this customer.

### Deriving Test Cases from the Decision Table

The process of creating specific test cases using decision tables can be presented in the following five steps.

**Step 1.** Identify all possible single conditions (from the test basis, e.g., based on specifications, customer conversations, so-called common sense), and list them in the consecutive rows in the upper part of the table. If needed, split compound conditions into single conditions. Conditions usually appear in specifications as sentence fragments preceded by words such as “if,” “in the event that,” etc.

**Step 2.** Identify all corresponding actions that can occur in the system and that are dependent on these conditions (also derived from the test basis), and list them in the consecutive lines in the lower part of the table. Actions usually appear in specifications as sentence fragments preceded by words such as “then,” “in this case,” “the system should,” etc.