a cultural change in the organization to bridge the gap between development, testing, and operations, treating their functions with equal value.

DevOps is based on the use of an iterative/incremental model of the SDLC supported by Agile principles such as team autonomy, rapid feedback, and integrated tool chains, as well as technical practices such as continuous integration (CI), continuous delivery of software, or continuous software deployment. These principles and practices allow teams to build, test, and release quality code faster through what is known as a DevOps pipeline [27]. To help achieve higher quality and faster delivery, DevOps pipelines automate manual activities wherever possible. Ideally, configuration management, compilation, software build, deployment, and testing are all included in a single, automated, repeatable deployment pipeline.

In practice, effective implementation of a pipeline means that as much testing as possible must be automated, because otherwise testing slows down code delivery and potentially allows defects to seep into production. Ideally, these automated tests should provide rapid feedback on any defects found, as well as support continuous integration, continuous delivery, and continuous deployment.

Through the continuous integration process, new code uploaded by developers to the code repository is automatically merged, compiled, and built. Ideally, component (unit) tests are also uploaded. This is definitely easier if the TDD approach is used. Automatic component tests are ran to make sure that the uploaded code passes these tests and are often part of regression test suites. In addition, static analysis of the code can be performed at this stage. Feedback to the developer is virtually instantaneous, especially in situations if code fails to compile and fails smoke tests or when anomalies are detected by the static analysis tool. Depending on the availability of test environments and the time required for testing, it may also be possible to run component integration tests at this stage. These component integration tests will be a mix of new tests for the new functionality and regression tests to ensure that the existing functionality still works as expected. The main advantage of continuous integration is that it provides developers with quick feedback if their code is flawed. When the feedback arrives too late, the developers are likely to continue development based on the flawed code, which is not efficient and requires context switching when the feedback arrives.

Continuous delivery of software can be considered an extension of continuous integration and includes another level of testing performed on a test environment that is representative of the production environment. Component integration tests, system tests and nonfunctional tests (many of which are regression tests) can be run as part of the continuous delivery process. If all these tests pass, the code is considered ready for deployment, but the decision to deploy is made by the DevOps team. The team can deploy the code automatically or manually.

Continuous deployment can be considered an extension of continuous delivery. It differs from the latter in that both the deployment decision and the deployment itself are automated. With continuous deployment, the DevOps team tries to achieve as much confidence as possible that automated testing will detect any defects that should not escape to the production environment. Organizations typically move to continuous deployment from continuous delivery when they gain sufficient

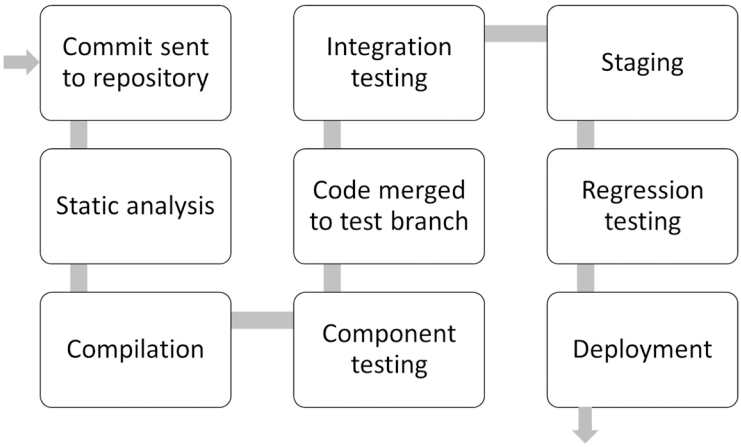

Fig. 2.12 Example of a deployment pipeline

confidence in the automated deployment pipeline. *Infrastructure as Code* (IaC) is also a common practice used by DevOps teams. It supports the automated configuration of test and production environments to support continuous delivery and continuous deployment.

An example of a deployment pipeline is shown in Fig. 2.12. After the developer submits code to the repository, an automatic process is started, which—if no problems occur along the way—leads to the automatic release of a new version of the software. Once the code is submitted, it is subjected to static analysis (e.g., it is checked whether the code is formatted correctly, whether the naming of variables follows the organization's rules, whether the code has a sufficiently low cyclomatic complexity,[2] etc.). The code is then compiled, and component tests are ran against it. After they pass, the new code is merged with the existing code base, and automatic component integration tests can be run. After they pass, the code is deployed to a pre-production environment (similar or identical to the production environment), and regression tests are run to see if the new code has caused problems elsewhere in the software. Once the regression tests pass, the code is released to the customer.

If at any point in this process something goes wrong (e.g., a negative static analysis result, a compilation error, a failed test, etc.), the process is immediately stopped, and the developers receive appropriate feedback. They can then correct the code and resubmit it to the repository by restarting the deployment pipeline.

A very big advantage of the automatic deployment pipeline, supported by the code management (versioning) process, is that at any point in the development cycle, the code is compiled, executable, and ready for release. Any change to the code automatically triggers the deployment pipeline, and developers receive immediate

[2]Cyclomatic complexity, or McCabe complexity, is a software metric developed by Thomas J. McCabe in 1976 and used to measure the complexity of a program's structure. The basis for the calculation is the number of decision points in the control flow diagram of a given program.

feedback on the level of code quality. In the event of any problem, the code reverts to the (working) version from before the changes.

From a testing perspective, the benefits of the DevOps approach are as follows:

- Fast feedback on the code quality and whether changes adversely affect the existing code.
- CI promotes a shift-left approach in testing (see Sect. 2.1.5) by encouraging developers to submit high-quality code accompanied by component tests and static analysis.
- DevOps facilitates stable test environments by promoting automated continuous integration and continuous delivery (CI/CD) processes.
- Increases the view on nonfunctional quality characteristics (e.g., performance, reliability).
- Automation through a delivery pipeline reduces the need for repetitive manual testing.
- The risk in regression is minimized due to the scale and range of automated regression tests.

However, DevOps is not without risks and challenges, in particular:

- The DevOps deployment pipeline must be defined, established, and maintained.
- Continuous integration tools must be in place and maintained.
- Test automation requires additional resources and can be difficult to establish and maintain.
- Teams sometimes become overly dependent on component testing and perform too little system and acceptance testing.

2.1.5 Shift-Left Approach

The principle of "Early testing saves time and money" (see Sect. 1.3) is sometimes called "**shift-left**" 🕮 because it is an approach in which testing is done as early as possible in a given SDLC. Shift-left usually implies that testing should be done earlier (e.g., not waiting for code implementation or component integration), but does not mean that testing later in the development cycle should be neglected.

There are several best practices that illustrate how to achieve shift-left in testing, which include:

- **Use of reviews**. Reviews can be performed early in the development cycle, even before the code is written. A review of the specification often finds potential defects in the specification, such as ambiguities, incompleteness, inconsistencies, superfluous elements, or contradictions.
- **Use of continuous integration and continuous delivery** (see Sect. 2.1.4) provides rapid feedback on code quality and forces the creation of automated component tests for source code as it is submitted to the code repository.

- **Performing static analysis**. Static analysis of source code can be performed before dynamic testing or as part of an automated process, such as a DevOps deployment pipeline (see Sect. 2.1.4).
- **Performing nonfunctional testing as soon as possible**. This is a form of shift-left, since nonfunctional testing is usually done later in the software development cycle, when a complete system and a representative test environment are available.
- **Using *model-based testing***. In this approach, the tester uses or creates a model of the software, and a special tool on the basis of this model automatically creates test cases that achieve a certain coverage criterion. The model can usually be created before the implementation work begins.

Examples of the use of the shift-left approach can be seen well in some SDLC models or development practices:

- In the V model, testing activities (e.g., test design or prototyping) already take place when the corresponding development phase begins.
- In iterative models, testing is usually present in every iteration, so it starts early in the project.
- In test-driven development (a "test-first" approach), writing tests before writing code in TDD or ATDD (see Sect. 2.1.3) naturally moves testing activities early in the development cycle.

2.1.6 Retrospectives and Process Improvement

Retrospectives (also known as *lessons learned* meetings) take place at the end of a project or when a milestone is reached in a project, such as at the end of an iteration or upon test level completion. During these meetings, participants discuss what worked well (what was successful), what didn't work and can be improved, and how to make improvements and sustain successes in the future. The meetings specifically cover topics such as:

- Processes and organization (e.g., did the procedures used delay, hinder, or rather speed up and facilitate the performance of certain tasks?)
- People (e.g., did the team members have the right knowledge and skills, or are there any deficiencies in this area and the need for training?)
- Relationships (e.g., did the team work together smoothly? Was communication effective?)
- Tools (e.g., did the use of certain tools increase the effectiveness or efficiency of testing? Would acquiring a certain tool help increase effectiveness or efficiency?)
- The product that was developed and tested (e.g., are there features that can be improved? Is there a level of technical debt that should be resolved?)

Usually, however, participants are not restricted to discussing any specific areas. If there are appropriate corrective actions against the problems identified during the

retrospective, they contribute to the self-organization of the teams and the continuous improvement of development and testing. The results of the retrospective should be recorded and can be part of the test completion report (see Sect. 5.3.2).

Retrospectives can result in decisions on test-related improvements and thus contribute to improving the test process. In particular, successfully conducted retrospectives provide the following benefits for testing:

- Increasing test effectiveness (e.g., more defects detected)
- Increasing test efficiency (e.g., achieving the same goal with less effort through the use of tools)
- Improving the quality of test cases (more likely to detect defects, less likely for defects to occur in the tests themselves)
- Increasing test team satisfaction (boost morale, improve team relations, increase team effectiveness)
- Improving collaboration between developers and testers

Retrospectives can also address the testability of applications, user stories, or other requirements, functions, or system interfaces. Root cause analysis of defects can also lead to improvements in testing and development. In general, teams should implement only a few improvements at a time (e.g., in a given iteration) to enable continuous improvement at a steady pace.

The timing and organization of the meeting depend on the specific model of the software development life cycle. For example, in agile software development, business representatives and the agile team attend each retrospective as participants, while the facilitator organizes and leads the meeting. In some cases, teams may invite other participants to the meeting.

Testers should play an important role in retrospectives because they bring their unique perspective (see Sect. 1.5.3). Testing occurs in every iteration or release and contributes to the success of the project. All team members are encouraged to contribute to both testing and non-testing activities.

Retrospectives should take place in an atmosphere of mutual trust. The characteristics of a successful retrospective are the same as those of a review (see Chap. 3).

A typical outcome of a retrospective is the selection and prioritization of one (and only one) improvement action that shall be added to the next iteration backlog. The improvement action might be outward looking (product related) or inward looking (team related).

By committing to implementing one new improvement action per iteration, the team aims to achieve continuous improvement.

It is undesirable to select multiple improvement actions for the next iteration as it lowers the potential customer value of the next iteration but also since it makes it impossible to uniquely identify the (positive) impact of the improvement action.

2.2 Test Levels and Test Types

FL-2.2.1 (K2) Distinguish the different test levels
FL-2.2.2 (K2) Distinguish the different test types
FL-2.2.3 (K2) Distinguish confirmation testing from regression testing

Test levels 🕮 are groups of test activities that are organized and managed together. Each test level is an instance of the test process consisting of the activities described in Sect. 1.4, performed for software at a given development level, from individual components to complete systems or systems of systems. These levels are linked to other activities performed as part of the software development life cycle.

Test levels are characterized in particular by attributes such as:

- **Test object** 🕮
- Test objective
- Test basis
- Defects and failures
- Specific approaches and responsibilities

Testing associated with a specific characteristic of a test object is called a **test type** 🕮. The difference between test levels and test types is that test levels refer to the phases of the development life cycle and the stage of advancement in product development, while test types take into account the test objective and the reason for testing. Therefore, test levels and test types are independent of each other, which means that each test type can be performed at any test level. We will come back to this issue later.

2.2.1 Test Levels

The Foundation Level syllabus describes five typical, most common test levels. These are:

- Component testing
- Component integration testing
- System testing
- System integration testing
- Acceptance testing

However, it is important to remember that this is only an example classification. Software development organizations may use only some of these levels or have other, additional levels specific to their organization.

Each test level requires an appropriate test environment. However, there might be constraints regarding the number of available test environments. Typically, the

minimal available test environments are identified as DTAP: development environment, test environment, acceptance environment, and production environment.

For component testing, this environment is usually a unit test environment, i.e., xUnit-type frameworks (e.g., JUnit), libraries for creating so-called mock objects (e.g., EasyMock), etc. For acceptance testing, on the other hand, a test environment similar to a production environment is ideal, since a large part of field defects (i.e., those reported by users after the software is released to the customer) are defects created by the interaction of the system and the environment.

2.2.1.1 Component Testing

Objectives of Component Testing

Component testing 🕮, also known as unit testing or program testing, focuses on modules that can be tested *separately*. The goals of this type of testing include:

- Component risk mitigation
- Checking the compatibility of the functional and nonfunctional behavior of the component with the design and specifications
- Building confidence in the quality of the component
- Component defect detection
- Preventing defects from reaching higher levels of testing

In some cases—especially in incremental and iterative SDLC models (e.g., in the agile projects), where code changes are continuous—automated component regression testing is a key component to ensure that changes made have not caused other existing components to malfunction.

Component tests are most often performed in *isolation* from the rest of the system (this depends primarily on the SDLC model and the specific system), in which case it may be necessary to use service virtualization, test harnesses, or mock objects (e.g., stubs or drivers). This isolation means that component tests should be able to be executed in any order, and their results should be the same regardless of the execution order.

Mock objects are used in component tests to allow actual execution of batches of code that refer to other objects (e.g., a database) that do not yet—at the time of component testing—exist. This is a common situation, since component tests are usually executed early in the development life cycle and many system components may not yet be ready. The mock objects enable the code to execute and thus test its own functionality—note that we are not interested here in the communication or interaction of the component under test with the mock object (e.g., a mocked database) itself. Testing such communication is only handled by component integration testing.

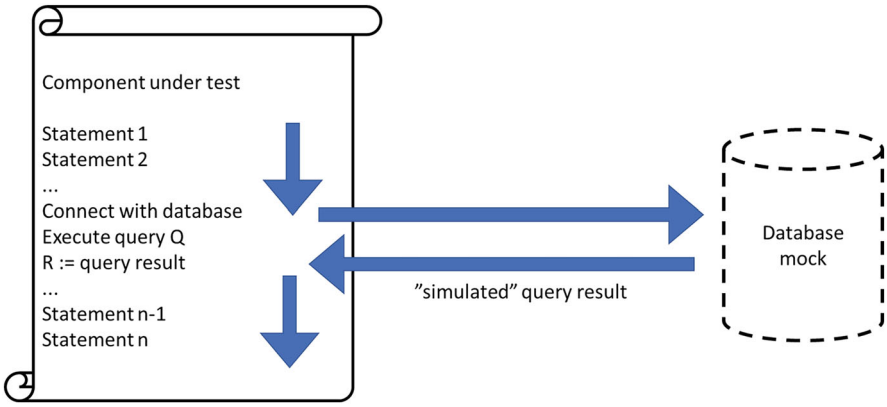

Fig. 2.13 Using a dummy object

Let's consider the example in Fig. 2.13. Let's assume that we are testing a script that at some point has to send a query to the database, receive the result of that query, and process it in a certain way. In the component test, we will not be interested in the script's communication with the database, whether the query was properly transmitted, understood, and the result correctly delivered to the script. Here we are only interested in the functionality of the component itself—whether it does what it should do according to the specification. Thus, a substitute for a database could even always return the same query result, as it is not the form of the result that is important here but what the script does with the received query.

Due to the caller-callee relationship, two types of mock objects can be distinguished: *stub* and *driver*. In principle, these objects perform the same role, i.e., they simulate the operation of other objects that do not yet exist. The only difference is whether the dummy object *is called* by our component under test or whether the mock object *calls* our component under test. In the first case, the implementation of such a mock object is called a stub, in the second one—a driver. This difference is shown symbolically in Fig. 2.14. When we use both stubs and drivers for testing, such an environment is called a **test harness**.

Component testing may cover functionality (e.g., correctness of calculations) but also nonfunctional characteristics of the software (e.g., performance, reliability), as well as structural properties (e.g., statement or decision testing).

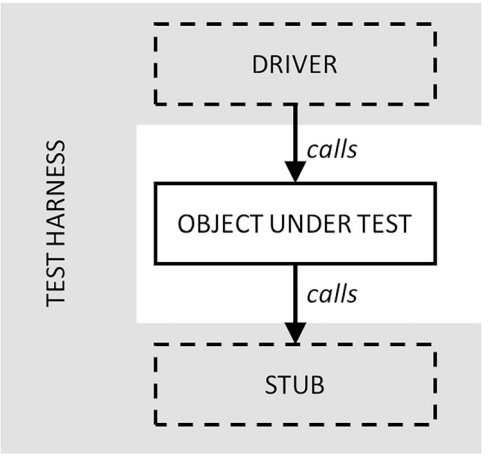

Fig. 2.14 Stub, driver and test harness

Details on Component Testing

Test basis

Examples of work products that can be used as a test basis for component testing include:

- Detailed design
- Code
- Data model
- Component specifications

Test objects

Typical test objects for component testing include:

- Modules, units, or components
- Code and data structures
- Classes or methods (in object-oriented programming)
- Database components

Typical defects and failures

Examples of common defects and failures detected by component testing are:

- Incorrect functionality (e.g., inconsistent with project specifications)
- Data flow problems
- Incorrect code and logic

(continued)

Defects are usually fixed as soon as they are detected, often without formal defect management. Thus, information from component testing (e.g., how many times a developer corrected a piece of code, what type of defects were detected, how they were fixed, how long the fix took) is typically not collected. However, it should be noted that by reporting defects, developers provide important information for root cause analysis and process improvement.

Specific approaches and responsibilities

Component testing is usually performed by the developer—the author of the code, as it always requires access to the code under test. Therefore, developers can alternate between creating components and detecting/removing defects. Developers often write and execute tests after writing the code for a particular component. However, in some situations—especially in agile software development—automated test cases for component testing can also be created before the application code is written (see Sect. 2.1.3).

2.2.1.2 Component Integration Testing and System Integration Testing

Component integration testing and system integration testing in principle are very similar, so these two test levels will be discussed together.

Component integration testing 🕮 focuses on interactions and interfaces between the integrated components. A component here can be understood as a class, file, function, procedure, package, etc. Tests of this type are performed after component testing and are usually automated. In iterative software development, component integration tests are usually part of the continuous integration process.

System integration testing 🕮 focuses on interactions and interfaces between systems, packages, and microservices. This type of testing may also involve interactions with interfaces provided by external organizations (e.g., *web service* providers). In this case, the software development organization does not control the external interfaces, which can create a whole range of testing problems (related, e.g., to fixing defects in the code created by the external organization that block testing, or to preparing test environments). System integration testing can take place after system testing or in parallel with ongoing system testing (this applies to both sequential and incremental SDLCs).

Integration Testing Goals

Typical **integration testing** 🕮 objectives are:

- Mitigating risks arising from component/system interactions
- Checking the compliance of functional and nonfunctional behavior of component/system interfaces with the design and specifications
- Building confidence in the quality of the interfaces used by the components/systems
- Detecting defects in interfaces and communication protocols
- Preventing defects from reaching higher levels of testing

Details on Integration Testing

Test basis

The test basis of integration testing will be any kind of documentation that describes the interaction or cooperation of individual components or systems. Examples of work products that can be used as a test basis for integration testing include:

- Software and system design
- Sequence diagrams
- Interface/communication protocol specifications
- Use cases
- Architecture at the component and system level
- Workflows
- Definitions of external interfaces

Mentioning use cases as the test basis for integration testing may seem strange. However, it will cease to be so if we realize that use cases describe *interactions* between so-called actors, that is, most often between a user and a system or between two systems. Thus, use case scenarios are well suited to be the basis for integration testing.

Test objects

Typical test objects include:

- Application programming interfaces (APIs) that provide communication between components
- Interfaces that provide communication between systems (e.g., system-to-system, system-to-database, microservice-to-microservice, etc.)
- Communication protocols between components and systems

Typical defects and failures

Examples of common defects and failures detected in component integration testing are:

- Incorrect or missing data or incorrect data coding
- Incorrect sequencing or incorrect synchronization of interface calls
- Incompatible interfaces
- Communication errors between components
- Failure to handle or incorrect handling of communication errors between components
- Incorrect assumptions about the meaning, units, or boundaries of data transferred between components

Examples of common defects and failures detected in systems integration testing are:

(continued)

- Inconsistent message structures sent between systems
- Incorrect or missing data or incorrect data coding
- Incompatible interfaces
- Inter-system communication errors
- Failure to handle or improper handling of inter-system communication errors
- Incorrect assumptions about the meaning, units, or boundaries of data transferred between systems
- Failure to comply with mandatory security regulations

Specific approaches and responsibilities

Component integration tests and system integration tests should focus on the integration itself. For example, when integrating component A with component B, tests should focus on the communication between these modules, rather than on the functionality of each of them (this functionality should previously be the subject of component testing). Similarly, in the case of integrating system X with system Y, the testing should focus on the communication between these systems, and not on the functionality of each of them (this functionality should be the subject of system testing). At integration test level, functional, nonfunctional and structural testing can be used.

Component integration testing is often the responsibility of developers, while system integration testing (due to its high-level nature) is usually the responsibility of testers. Whenever possible, testers performing systems integration testing should be familiar with the system architecture, and their comments should be taken into account at the integration planning stage.

The test environment, especially for system integration testing, should, as far as possible, reflect the specifics of the target or production environment. This is because a very large number of failures arise from the system's interaction with the environment in which the system is seated. A test environment identical or similar to the target environment makes it possible to detect most of these failures and the defects that cause them during the testing phase, before the software is released to the customer.

As with component testing, there are situations where automatic integration regression testing allows you to be sure that the changes made have not caused the existing interfaces to malfunction.

Integration strategies

Planning integration tests and integration strategies prior to building components or systems enables those modules or systems to be produced in a way that maximizes testing efficiency. There are a number of approaches to integration testing strategies. The old Foundation Level syllabus (v3.1) lists several of them. They mainly deal with component integration. These are:

(continued)

- **Strategies based on the system architecture**, namely, *top-down* and *bottom-up strategies*. In a top-down strategy, the integration of a core component with the components it calls is tested first, followed by the integration of those components with the components they call and so on. Thus, integration occurs by "levels" of recess of components in the hierarchy of their calls. In a bottom-up strategy, the direction of integration is the opposite: we start from the bottom and sequentially include components lying at higher levels, calling the components already called. Using a system architecture-based integration strategy, stubs (descending strategy) and drivers (ascending strategy) are often used, because in general the called or calling modules are not yet written.
- **Functional task-based strategies**. Sometimes, for certain reasons, it is important for us to first test the integration of a group of components that together are responsible for some functionality that is important to us at the moment. Thus, the strategy is to first test the integration of the components that make up that functionality and then test the integration of the other modules.
- **Strategy based on sequences of transaction processing**. If we are primarily interested in testing, for example, the path of information flow through the system (e.g., the path from a *specific input to* a specific *output, to achieve the system observability as soon as possible*), we first carry out integration tests of the components lying on this path. Next, we conduct integration tests of the remaining components.
- **Strategy based on other aspects of the system**. The last two examples of integration strategies consisted of determining the order of integration tests due to some criteria that were important to us. One can imagine many other such strategies, such as those based on criticality of components or frequency of use of inter-component communications.

Example. Let us see with an example how the above strategies would look like in practice. Let us assume that our system has the architecture shown in Fig. 2.15. If two components are connected with a line, it means that the component lying above calls the one lying below (so A is the main component—it calls B, F, and G; component B calls C and D, etc.). In addition, suppose that B, C, D, and E form a single functionality, concerning the input handling. Component E is responsible for taking data from the user and component H for reporting the results.

In a top-down strategy, the order of integration testing will be as follows:

1. Integration of A-B, A-F, A-G (if B, F, G are not yet ready, use their stubs).
2. Integration of B-C, B-D, F-H, G-H (it may be necessary to use stubs for C, D, and H).
3. Integration of D-E, D-I, H-I (it may be necessary to use stubs for E and I).

(continued)

The bottom-up strategy is analogous; only the order is reversed:

1. Integration of D-E, D-I, H-I (it may need to use drivers for D and H)
2. Integration of B-C, B-D, F-H, G-H (it may be necessary to use drivers for B, F, and G)
3. Integration of A-B, A-F, A-G (it may be necessary to use a driver for A)

In a functionality-based strategy—assuming that input handling functionality is a priority for us—an example order of integration might be this:

1. B-C, B-D, D-E integration (input handling functionality)
2. Integration A-B, D-I (tests of integration of input handling with the environment)
3. Integration A-F, A-G, F-H, G-H, H-I (other connections)

In turn, a strategy based on the sequence of the transaction being processed can take the following form:

1. Integration of E-D, D-B, B-A, A-F, F-H (input-output path)
2. Integration of A-G, B-C, G-H, D-I, H-I (other connections)

Incremental integration methods—such as those outlined above—allow, at the time of failure, the defect to be localized quickly. If, in our example, we use a top-down strategy and the failure occurs at the time of F-H testing, we can be almost certain that the defect relates to F-H communication and is therefore localized in one of these two components, rather than in A or B, for example.

Big bang integration, which involves integrating all components or systems at once, is not recommended, because when a failure occurs, we usually have no idea what it might have been caused by. Risk analysis of the most complex interfaces can also help guide integration testing accordingly.

The broader the scope of integration, the more difficult it is to pinpoint a specific component or system with defects, which can lead to increased risk and longer time to diagnose problems. This is one reason why the *Continuous Integration* (CI) method of integrating software component by component (e.g., functional integration) is commonly used. Automated regression testing, which should be done at multiple test levels whenever possible, is often a part of continuous integration.

2.2.1.3 System Testing

Objectives of System Testing

System testing 🕮 level is the typical level of a tester's activity (component testing and integration testing are usually done by the developer and acceptance testing by the customer). System testing focuses on the behavior and capabilities of an entire,

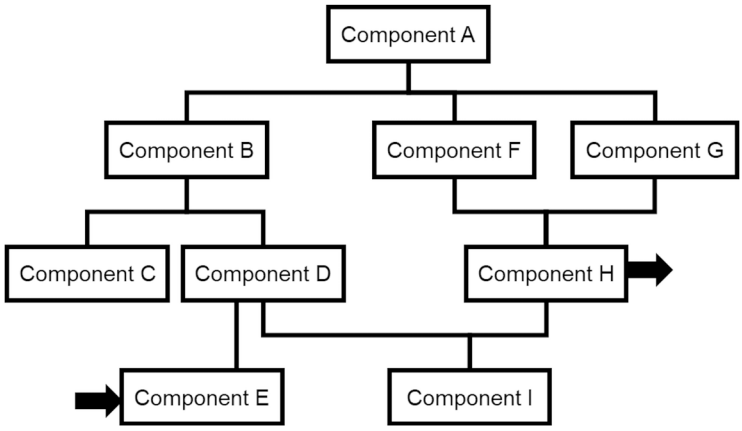

Fig. 2.15 Example of a system architecture

already integrated system or product, often taking into account the entirety of the tasks it can perform and the nonfunctional behaviors it exhibits while performing those tasks. The goals of system testing are primarily:

- Reducing the risk of system malfunction
- Checking the compliance of the functional and nonfunctional behavior of the system with the design and specifications
- Checking the completeness of the system and the correctness of its operation
- Building confidence in the quality of the system as a whole
- Detecting defects in the system
- Preventing defects from reaching the acceptance testing level or production

For some systems, the purpose of system testing may also be to verify data quality. As in the case of component or system integration testing, automated system regression testing provides assurance that changes made have not caused legacy functions or overall functionality to malfunction. System testing often results in information on the basis of which stakeholders make decisions to transfer the system to production use. In addition, system testing may be necessary to meet the requirements of applicable regulations or norms/standards.

As with the system integration test level, the test environment should, as far as possible, reflect the specifics of the target or production environment.

At the system test level, most feedback is generated from the test process. This includes the number and type of defects found in the program, the time taken to fix them, when the defect was introduced, etc.

Typically, system testing focuses on the verification of the system, by comparing the behavior of the system as described in the requirements with the actual behavior. System testing confirms that “the system was built right.”

Details on System Testing

Test basis

System testing refers to a fully integrated system, so the test basis should refer precisely to the system as a whole. Examples of work products that can be used as a test basis for system testing include:

- Specifications of requirements (functional and nonfunctional) for the system and software
- Risk analysis reports
- Use cases
- User stories and epics
- Models of system behavior
- State diagrams
- System operation statements and user manuals

Test objects

Typical test objects for system testing include:

- System under test (the system as a whole)
- System configuration and configuration data

Typical defects and failures

Examples of common defects and failures detected by system testing include:

- Incorrect calculations
- Incorrect or unexpected functional or nonfunctional behavior of the system
- Incorrect control flows and/or data flows in the system
- Problems with the correct and complete performance of overall functional tasks
- Problems with the proper operation of the system in a production environment
- Inconsistency of system operation with the descriptions contained in the system's statements and user manuals

Specific approaches and responsibilities

System testing should focus on the overall behavior of the system as a whole at both functional and nonfunctional aspects. System testing often uses test techniques (see Chap. 4) that are most appropriate for the particular aspect(s) of the system under test. For example, a decision table can be used to test whether functional behavior is consistent with the description of the business rules.

System testing is usually performed by independent testers. Defects in the specifications (e.g., missing user stories or incorrectly expressed business requirements) can lead to misunderstandings or disagreements about the

(continued)

expected behavior of the system. As a result, false-positive or false-negative results can occur (see Table 1.2), resulting in wasted time or reduced defect detection performance, respectively. For this reason, among others—in order to reduce the occurrence of the above situations—testers should be involved in reviews, refinement of user stories, and other static testing activities as early as possible in the SDLC.

Some nonfunctional system tests may require that they be conducted in a representative, production environment. Examples of such tests might be:

- Performance tests (e.g., the environment should reflect the actual network throughput, and components should not be replaced with mock objects, as this may disrupt, for example, the reporting of a module's response time)
- Security testing (e.g., security attacks will depend on the specific system configuration and environment in which the system is located)
- Usability testing (e.g., final interface usability tests should be conducted on the actual interface, not on a mock of this interface)

2.2.1.4 Acceptance Testing

Acceptance Testing Objectives

Acceptance testing 🕮—like system testing—usually focuses on the behavior and capabilities of the entire system or product. However, it is conducted from the perspective of the user, not the development team. Thus, it has the character of validation rather than verification.

The objectives of acceptance testing are primarily:

- Building user confidence in the system
- Checking the completeness of the system and its proper operation from the point of view of achieving business objectives

Acceptance testing can produce information to assess the readiness of the system for deployment and use by the customer (user). Defects may also be detected during acceptance testing, but their detection is most often not the main test objective. Rather, at this stage, we are concerned with the *validation of the system*, i.e., checking whether the system actually meets the customer's business needs. Acceptance testing confirms that "the right system was built."

In some cases, finding a large number of defects at the acceptance testing level may even be considered a significant project risk. After all, if the customer is supposed to check whether the software solves their problem and the system "crashes every other click," such tests are rather pointless—they are only a sign that some fundamental errors were made in the quality control process, through which a lot of defects slipped through all the manufacturing phases until the software is released to the customer.

In addition, acceptance testing may be necessary to meet requirements under applicable laws or standards/regulations.

The most common forms of acceptance testing are:

- User Acceptance Testing (UAT)
- Operational Acceptance Testing (OAT)
- Acceptance testing for contractual and legal compliance

Acceptance testing can also be divided by where it is performed:

- Alpha testing—testing by the customer at the producer's site, on a test environment
- Beta testing (also known as field testing)—testing by the customers at their own target environments

The various forms of acceptance testing are described in the following subsections.

User Acceptance Testing
User acceptance testing takes place in a (simulated) production environment. The main goal is to build confidence that the system will enable users to meet their needs, the requirements placed on it, and perform the business processes with a minimum of problems, cost, and risk.

Operational Acceptance Testing
System acceptance testing by operators or administrators usually takes place in a (simulated) production environment. Testing focuses on operational aspects and may include:

- Testing backup and recovery mechanisms
- Installing, uninstalling and updating software
- Failure recovery
- User management
- Maintenance activities
- Data loading and migration activities
- Checking for security vulnerabilities
- Performing performance tests

The main objective of operational acceptance testing is to gain confidence that operators or system administrators will be able to ensure that users will be able to operate the system correctly in a production environment, even under exceptional and difficult conditions.

Acceptance Testing of Contractual and Legal Compliance
Acceptance testing of compliance with the contract is carried out in accordance with the acceptance criteria written in the contract for the contracted software development. These acceptance criteria should be specified when the parties agree on the content of the contract. This type of acceptance testing is often performed by users or independent testers.

Table 2.2 DO-178C standard requirements for test coverage by risk level

Criticality level	Type of failure	Coverage requirements
A	Catastrophic	Full MC/DC (modified condition/decision) coverage required
B	Hazardous/ severe	Full decision coverage required
C	Major	Low-level requirements coverage required; testing of proper data handling and control; full statement coverage required
D	Minor	High-level requirements coverage required
E	No effect	No requirements

Legal compliance acceptance testing is performed in the context of applicable legislation, such as laws, regulations, or safety standards. This type of acceptance testing is often performed by users or independent testers, and the results can be observed or audited by regulators. The main objective of acceptance testing for contractual and legal compliance is to obtain assurance that compliance with requirements under applicable contracts or regulations has been achieved. A good example of a standard that imposes specific coverage criteria is the DO-178C standard applicable to aviation software. In this standard, software is classified by risk level (levels A through E), and then for each classification, the requirements for meeting specific coverage criteria are explicitly defined (see Table 2.2). Coverage criteria: MC/DC, decision, and statement coverage appearing in this standard refer to specific white-box coverage criteria. The latter, as well as the branch coverage criterion—similar to decision coverage—are discussed in the Foundation Level syllabus (see Sects. 4.3.1 and 4.3.2). The MC/DC criterion is discussed in the advanced level syllabus—Technical Test Analyst.

Alpha and Beta Testing

Developers of so-called commercial of-the-shelf (COTS), that is, software for general sale, often want feedback from potential or existing customers before the software hits the market. Alpha and beta tests are used for this purpose.

Alpha testing is performed on the premises of the software development organization, but instead of the development team, potential or current customers and/or operators or independent testers perform the tests. Beta testing, on the other hand, is performed by current or potential customers at their own locations. Beta testing may or may not be preceded by alpha testing.

One of the test objectives of alpha and beta testing is to build the confidence of potential and existing customers and/or operators that they can use the system under normal conditions, in a production environment, to achieve their goals with minimal effort, cost, and risk. Another purpose may be to detect errors related to the conditions and environment(s) in which the system will be used, especially when such conditions are difficult for the project team to reproduce. If beta testing is conducted on a representative group of users, then because it is conducted in the

environments used by individual testers, the tests will cover in a representative way the environments in which the system will operate. For example, if 80% of users have Windows 11 and 20% have Windows 10, then in a random, representative group of beta testers, about 80% of them should be testing a product installed under Windows 11 and about 20% under Windows 10.

Details on Acceptance Testing
Test basis

Examples of work products that can be used as a test basis for any type of acceptance testing include:

- Business processes
- User requirements or business requirements
- Regulations, agreements, norms, and standards
- Use cases
- System documentation or user manuals
- Installation procedures
- Risk analysis reports

In addition, the test basis from which test cases are derived for operational acceptance testing can be the following work products:

- Backup and restoration procedures
- Failure recovery procedures
- Operational documentation
- Implementation and installation documentation
- Performance assumptions
- Norms, standards, or regulations in the field of security

Test objects

Typical test objects of any acceptance test type include:

- System under test
- System configuration and configuration data
- Business processes performed on a fully integrated system
- Backup systems and *hot site* replacement centers (for testing business continuity and failure recovery mechanisms)
- Processes related to operational use and maintenance
- Forms
- Reports
- Existing and converted production data

Typical defects and failures

(continued)

Examples of common defects detected by various forms of acceptance testing include:

- System workflows that are incompatible with business or user requirements
- Incorrectly implemented business rules
- Failure of the system to meet contractual or legal requirements
- Nonfunctional failures, such as security vulnerabilities, insufficient performance under heavy load, or malfunctioning on a supported platform

Specific approaches and responsibilities

Acceptance testing often rests with customers, business users, product owners, or system operators, but other stakeholders may also be involved in the process. Acceptance testing is often considered the last level of the sequential software development cycle, but it can also take place at other stages, for example:

- Acceptance testing of software for general sale can take place during installation or integration.
- Acceptance testing of a new functional enhancement can take place before system testing begins.

In iterative SDLC models, project teams may use various forms of acceptance testing at the end of each iteration, such as tests focused on verification that new functionality meets acceptance criteria or tests focused on validation of new functionality against user needs. In addition, at the end of each iteration, alpha and beta testing, as well as user acceptance testing, production acceptance testing, and contractual and legal compliance acceptance testing, may be performed after the completion of each iteration or a series of iterations.

Example A company develops web-based software for the Polish IRS office to enter data from PIT-11 tax forms by an official to generate a final PIT-37 tax form for a particular taxpayer.

An example of a component test would be to test the correctness of a data entry form, for example, in terms of how currency fields will behave when a character value or an amount with incorrect accuracy is entered into them.

An example of a component integration test would be the interaction between the system's login function and a function that grants certain permissions to a user.

An example of a system test would be for the tester to run through the entire data entry process of several PIT-11 forms and then verify that the system generates the correct PIT-37.

An example of a systems integration test would be to test the correctness of pulling taxpayer data from the PESEL (personal identification number used in Poland) database based on the taxpayer's PESEL number entered by the user.

An example of an acceptance (beta) test would be the validation of the system by tax office officials at the office, on their own computers, and in the production environment. Validation can check the substantive correctness (e.g., the program's compliance with the law) but also usability issues (e.g., whether interfaces are intelligible) or how well the system integrates with the tax office's IT infrastructure.

2.2.2 Test Types

A test type is a group of dynamic test activities performed to test specific characteristics of a software system (or part of it) according to specific test objectives. Thus, unlike test levels, test type is not related to phases in a development process, but to the objective we have in mind when performing a given test.

Test objectives may include:

- Evaluation of functional quality characteristics such as completeness, correctness, and adequacy
- Evaluation of nonfunctional quality characteristics, including parameters such as reliability, performance, safety, compatibility, or usability
- Determining whether the structure or architecture of a component or system is correct, complete, and in accordance with specifications

The Foundation Level syllabus distinguishes four basic test types:

- Functional testing
- Nonfunctional testing
- White-box testing
- Black-box testing

However, it should be noted that many more test types can be distinguished. An example would be, for example, the so-called smoke tests, the purpose of which is to verify the correctness of the basic functionality of the system, before starting to perform more detailed tests.

We will now discuss each of the above four test types.

2.2.2.1 Functional Testing

Functional testing 🕮 of a system involves the execution of tests that evaluate the functions that the system should perform. Functional testing thus tests "what" the system should do. Functional requirements can be described in work products such as business requirements specifications, user stories, use cases, or system requirements specifications, but they also happen to exist in undocumented form.

The main test objective of functional testing is to verify functional *correctness*, functional *appropriateness*, and functional *completeness*. These are the three

Table 2.3 Test levels vs. functional testing

Test level	Sample test basis	Sample test
Component	Component specifications	Verification of the correctness of the component's calculation of the tax amount
Component integration	Component diagrams (architecture design)	Verification of the data transfer between the login component and the authorization component
System	User stories, use cases	Verification of the correct implementation of the business process "pay tax"
System integration	Specification of the system-database interface	Verification of the correctness of sending a query to the database and receiving its result
Acceptance	User manual	Validate that the description of the implementation of functionality contained in the manual is consistent with the actual business process realized by the system

sub-characteristics of functionality defined in the ISO/IEC/IEEE 25010 quality model [5].

Functional testing is the test type most commonly associated with testing, because it is common to equate testing with checking that the system does what it is supposed to do for the user. However, it should not be forgotten that other test types, including those discussed in the following sections, are equally important.

Functional testing can (and should) be performed at all test levels (e.g., tests on components can be based on component specifications), but with the caveat that tests performed at each level focus on different issues (see Sect. 2.2.1). Table 2.3 gives an example of activities performed by functional testing at different test levels for the aforementioned tax office data entry system.

Functional testing takes into account software behavior, which is most often described in documents external to the system: requirements specifications, user stories, etc. Consequently, functional testing typically uses black-box techniques to derive test conditions and test cases checking the functionality of a component or system (see Sect. 4.2), but is not limited to these techniques.

The thoroughness of functional testing can be measured by functional coverage. The term "functional coverage" refers to the degree to which a specific type of functional item has been tested, expressed as a percentage of items of a given type covered by testing. By tracking the relationship between test cases, their results, and functional requirements, for example, it is possible to calculate what percentage of requirements have been covered by testing and as a result identify any gaps in coverage.

Example Table 2.4 shows the traceability between test cases and functional requirements. Each test case has been assigned its priority (related to the risk it covers) on a scale from 1 (minor) to 5 (critical).

Now let us assume that all six test cases have been performed. Let us also assume that tests 1, 2, and 4 pass and 3, 5, and 6 fail. Assume the following definition of requirement coverage: the coverage of requirement R is the sum of the priorities of

Table 2.4 Traceability between requirements and test cases

Test \ Requirement	Req 1	Req 2	Req 3
Test1 (priority 2)	X		
Test2 (priority 5)	X		X
Test3 (priority 1)		X	X
Test4 (priority 1)		X	
Test5 (priority 4)		X	
Test6 (priority 2)			X

the test cases associated with R that pass, relative to the sum of the priorities of all test cases associated with R.

For example, associated with requirement Req 1 are test cases 1 and 2 with a priority sum of 2 + 5 = 7. Since both tests are passed, the coverage of requirement Req 1 is 7/7 = 100%. Similarly, we can calculate the requirement coverage for all the requirements:

$$\text{Req 1 : coverage} = (2+5)/(2+5) = 7/7 = 100\%$$

$$\text{Req 2 : coverage} = (1)/(1+1+4) = 1/6 = 16.6\%$$

$$\text{Req 3 : coverage} = (5)/(5+1+2) = 5/8 = 62.5\%$$

Of course, what exactly the result of requirements test coverage will look like depends on the definition of coverage adopted. We could just as well, for example, define it as the ratio of tests passed (related to requirement W) to all tests related to requirement W. Then the coverage of requirements Req 1, Req 2, and Req 3 would be, respectively, 2 / 2 = 100%, 1 / 3 = 33.3%, and 1 / 3 = 33.3%.

Special skills or expertise may be needed to design and execute functional tests, such as knowledge of the specific business problem being solved by the software (e.g., geological modeling software for the oil and gas industry) or the specific role the software performs (e.g., a computer game that provides interactive entertainment). This is because functional testing is concerned with the *functions* that the system provides, and these are usually embedded in a specific business context. Therefore, for example, testers of financial applications should at least know the basics of finance.

2.2.2.2 Nonfunctional Testing

The purpose of **nonfunctional testing** 🕮 is to evaluate the characteristics of systems and software, such as usability, performance, and security. A classification of software quality characteristics is provided in the ISO/IEC 25010 standard (Systems and Software Quality Requirements and Evaluation (SQuaRE) [5]). Nonfunctional testing checks "how" a system behaves.

Figure 2.16 shows the quality model according to ISO/IEC 25010. It distinguishes eight quality characteristics (gray rectangles) and assigns to each of them

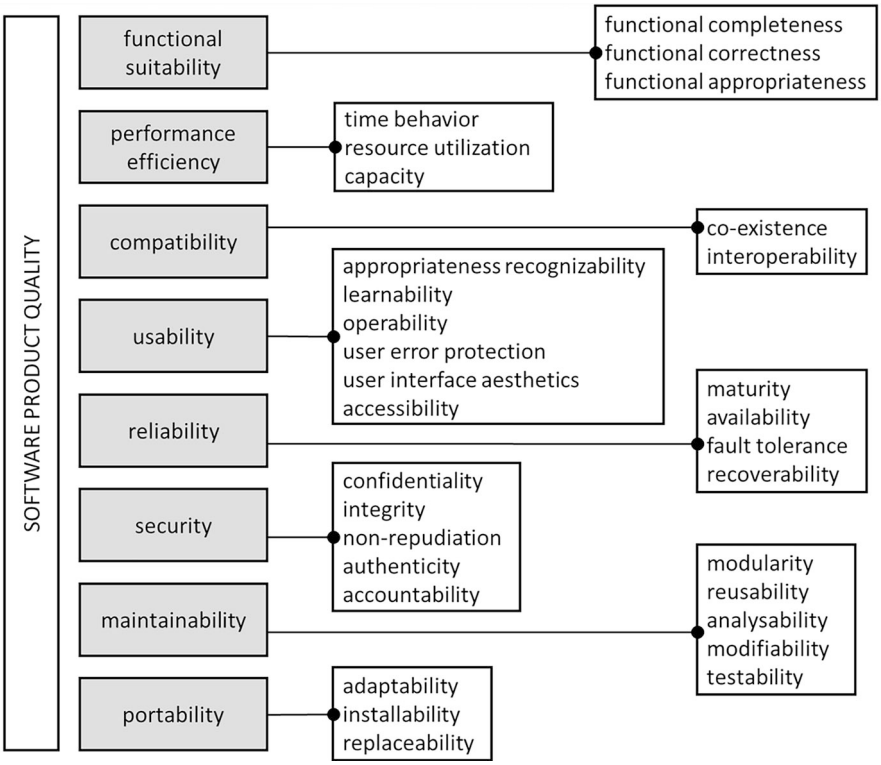

Fig. 2.16 ISO/IEC 25010—System and software quality models (SQuaRE)

corresponding sub-characteristics. From the point of view of the Foundation Level syllabus, nonfunctional characteristics are all but functional suitability. Quality models, such as ISO/IEC 25010, can be helpful in determining which nonfunctional parameters of our system should be tested.

Contrary to the common misconception, nonfunctional testing can and often should be performed at all test levels. Moreover, it should be done at the earliest possible stage, because detecting nonfunctional defects too late can be a very big threat to the success of the project. Many nonfunctional tests are derived from functional tests as they use the same functional tests, but check that while performing the function, a nonfunctional constraint is satisfied (e.g., checking that a function performs within a specified time or a function can be ported to a new platform).

Table 2.5 shows examples of nonfunctional tests of the aforementioned system for data entry by tax offices, which can be performed at each test level.

Black-box test techniques can be used to derive test conditions and test cases for nonfunctional testing (see Sect. 4.2). An example is the use of boundary value analysis to define stress conditions for performance testing.

The diligence of nonfunctional testing can be measured by nonfunctional coverage. The term "nonfunctional coverage" means the degree to which a specific type of

Table 2.5 Test levels vs. nonfunctional tests

Test level	Sample test
Component	Verify that the component can be easily replaced by another with equivalent functionality (portability—replaceability)
System integration	Verify that the communication between the client system and the central server are properly secured and unaffected by possible hacking attacks (security—confidentiality)
System	Verify that the system generates a summary report quickly enough based on 100,000 tax forms (performance efficiency—time behavior)
Acceptance	Validate that the system is adapted to the needs of the visually impaired users (usability—accessibility)

nonfunctional item has been tested expressed as a percentage of items of a given type covered by tests. By tracking the relationship between tests and the types of devices supported by a mobile application, for example, it is possible to calculate what percentage of devices were covered by compatibility testing, and consequently identify any gaps in coverage.

SMART Goals for Nonfunctional Testing
The results of nonfunctional tests must be precise, i.e., they should be expressible in terms of some well-defined metrics. Often, the acronym SMART is used to specify the hallmarks of properly defined requirements and tests: specific, measurable, attainable, realistic, and time-bound. This is because during test execution, we need to make a comparison between the actual result and the expected result and state *unequivocally* whether the test was passed or failed. In Table 2.6, we give examples of metrics for various quality characteristics according to the ISO/IEC 25010 model.

Special skills or special knowledge—such as knowledge of vulnerabilities specific to a particular project or technology (e.g., security vulnerabilities associated with certain programming languages) or a particular group of users (e.g., user profiles for healthcare management systems)—may be needed to design and execute nonfunctional tests. That is why it is common practice, for example, to hire third-party organizations that specialize in conducting tests of a specific type, such as security or performance testing.

For detailed information on nonfunctional testing of quality characteristics, see the syllabi Test Analyst [28], Technical Test Analyst [29], Security Tester [30], and other specialized ISTQB® syllabi.

Table 2.6 Example metrics for nonfunctional characteristics

Characteristics	Metrics	Model	Description
Reliability	Failure density per number of tests	M = A/B	A = number of failures detected B = number of tests executed
	Availability	M = A/(A + B)	A = mean time to failure (MTTF) B = mean time to repair (MTTR)
Usability	Ease of learning the function	M = T	T = learning time
	Availability of help	M = A/B	A = the number of tasks for which there is a valid help topic B = total number of tasks tested
	Attractiveness of interaction	questionnaire	Metrics measured by questionnaire after using software
Performance	Response time	M = T1 − T2	T1 = task completion time T2 = completion time for issuing a task request
	Throughput	M = A	A = number of tasks completed in a fixed unit of time
Maintainability	Failure analysis capability	M = 1 − A/B	A = number of failures, the cause of which is still unknown B = number of all observed failures
Portability	Possibility of interoperability	M = A/T	A = number of unexpected problems or failures during parallel use of other software T = total time during which other software was used in parallel

2.2.2.3 White-Box Testing

In the case of **white-box testing** 🕮, tests are derived on the basis of the internal structure or implementation of a given system. The internal structure can include code, architecture, workflows, and/or data flows within the system (see Sect. 4.3, where we describe in detail the white-box coverage techniques and criteria applicable to the Foundation Level exam).

The thoroughness of white-box testing can be measured by structural coverage. The term "structural coverage" means the degree to which a specific type of structural item has been tested expressed as the percentage of items of a given type covered by the tests. The general coverage metric is thus expressed by the formula:

$$\text{coverage} = A/B,$$

where A = number of structural elements covered by tests and B = number of all identified structural elements. An example is the statement coverage metric, which is