Table 2.7 White-box testing at different test levels

Test level	Example structure to be covered
Component	Source code (e.g., statements, branches, decisions, paths)
Component integration	Call graph (i.e., events of calling one function by another), a set of API functions
System	Business process model
System integration	Call graph at the level of services offered by communicating systems
Acceptance	Tree model of the menu structure

the number of executable statements covered by the tests divided by all executable statements in a given code.

The coverage metric defined as above takes values between 0 and 1 (because both A and B are positive and A is always less or equal to B). The more common form of presentation of this metric is its percentage form:

$$\text{coverage} = (A/B)^{*}\ 100\%.$$

For example, if we covered 5 out of 20 executable code statements with our tests, then we achieved the coverage 5 / 20 = 0.25 or, in other words, (5 / 20) * 100% = 25%.

At the component testing level, code coverage is determined by the percentage of a component's code that has been tested. This value can be measured in terms of various aspects of the code (coverage elements), such as the percentage of executable statements or the percentage of decision results tested in a given module. The above types of coverage are collectively called code coverage. At the component integration testing level, white-box testing can be done on the basis of system architecture (e.g., interfaces between modules), and structural coverage can be measured in terms of the percentage of interfaces tested. Table 2.7 shows examples of structures that can be used to perform white-box testing at different test levels.

Special skills or special knowledge may be needed to design and execute white-box tests, such as knowledge of code construction (enabling, e.g., the use of tools to measure code coverage), knowledge of how to store data (enabling, e.g., the evaluation of database queries), or how to use tools to measure coverage and correctly interpret the results they generate.

It's essential to highlight that achieving an arbitrarily high level of coverage does not guarantee an arbitrarily high quality. Even 100% coverage of all executable statements does not ensure that every executable statement actually performs as is expected. It's a means of measuring test completeness, rather than the quality of the system under test.

2.2.2.4 Black-Box Testing

Black-box testing 🕮 is based on specifications (i.e., information external to the test object). The main purpose of black-box testing is to verify that the system's behavior conforms to the behavior described in the specification.

For more information on black-box testing, see Sect. 4.2, where we discuss black-box test techniques in detail.

2.2.2.5 Test Levels vs. Test Types

Test levels and test types are independent of each other. Although in practice it will be the case that certain test types tend to occur at certain specific test levels, in general, any test type can be run at any test level. This is shown by the following example.

Example We are testing a web application of a book selling e-shop. Registered users, after logging in, have access to a catalog with a search function (including search by title, author, book category, price range). They can add selected items to the shopping cart and then proceed to payment. Some samples of functional, nonfunctional, black-box, and white-box tests are described below (for the sake of simplicity, we only give brief descriptions of the tests, not describing the test cases in detail).

Examples of functional (and black-box) tests:

- Validate user registration in the system.
- Check if it is possible to register a user with an existing name in the system.
- Check if it is possible to register a user with an existing e-mail address in the system.
- Check the standard purchase and correct payment.
- Check if when a book is put in the shopping cart, that copy is blocked in the system and other users cannot add it to their carts.
- Check that after a certain, fixed period of time, if there is no payment, the session expires and all the books in the shopping cart go back to the store and become available to other users.

Examples of nonfunctional tests:

- Check the response time after querying a catalog of 1 million items.
- SQL injection vulnerability test[3] via search box.
- Check the response time of the system in the situation of simultaneous execution of a query by a thousand users.

[3] SQL injection is an attack on a website or web application in which SQL language code is added to a field in a form to gain access to an account or change data.

Examples of white-box tests:

- Check the statement coverage of the customer registration component (intention: to check all paths and possible exceptions and errors that the system should handle).
- Check the performance of all elements of the main menu (covering the structure of the web page).
- Test all the interface functions related to the e-payment system.

Each of the test types listed above can be performed at any test level. To illustrate this, the following are examples of functional, nonfunctional, black-box, and white-box tests that are performed at all test levels for a banking application. The first group is functional (and black-box) tests:

- For component testing, tests are designed to reflect how a module should calculate compound interest.
- For component integration testing, tests are designed to reflect how account information captured in the user interface is transferred to the business logic layer.
- For system testing, tests are being designed to reflect how account holders can apply for an overdraft line of credit.
- For systems integration testing, tests are designed to reflect how a given system checks the creditworthiness of an account holder using an external microservice.
- For acceptance testing, tests are designed to reflect how a bank employee approves or rejects a loan application.

The next group contains examples of nonfunctional tests:

- For component testing, performance tests are designed that evaluate the number of CPU cycles required to perform complex calculations of the total amount of interest.
- For component integration testing, security tests are designed to detect security vulnerabilities related to buffer overflows of data passed from the user interface to the business logic layer.
- For system testing, portability tests are designed to verify that the presentation layer works across all supported browsers and mobile devices.
- For system integration testing, reliability tests are designed to assess the resilience of the system in the event of a lack of response from the microservice used to check creditworthiness.
- For acceptance testing, usability tests are designed to evaluate the accessibility features for people with disabilities, applied to the interface for processing loans on the bank's side.

The next group contains examples of white-box tests:

- For component testing, tests are designed with the objective of ensuring full statement coverage of code instructions and decisions in all modules that perform financial calculations.

- For component integration testing, tests are designed to see how each screen of the browser display interface passes data to the next screen and to the business logic layer.
- For system testing, tests are designed to ensure coverage of possible sequences of web pages displayed when applying for a line of credit.
- For systems integration testing, tests are designed that check all possible types of queries sent to the microservice used for credit checking.
- For acceptance testing, tests are designed to cover all supported structures and value ranges for financial data files used in interbank transfers.

This section provides examples of all test types performed at all levels, but not all software needs to include every test type at every level. What is important is that the appropriate tests are executed at each level—this is especially true for the earliest level at which a test type occurs.

2.2.3 Confirmation Testing and Regression Testing

Once changes have been made to the system to fix defects or add or modify functionality, tests should be conducted to confirm that the changes actually fixed the defect or correctly implemented the relevant functionality, without causing any unforeseen adverse consequences. The Foundation Level syllabus distinguishes between two types of such tests: confirmation tests (also called re-tests) and regression tests.

Confirmation Testing

Confirmation testing 🕮 (also known as re-testing) is performed in connection with the occurrence of a failure and the repair of the associated defect. The role of confirmation testing is to verify that the defect has indeed been repaired (see Fig. 2.17). Typically, a retest is the same test that revealed the failure, performed again, after the failure has been repaired, but sometimes the confirmation test can be a different test—for example, if the reported defect was a lack of functionality. The absolute minimum when performing confirmation tests is to execute the steps that caused the failure earlier. If the test is passed, we can assume that the defect has been fixed.

Since you never know when a failure will occur and when it will be repaired, the execution of retests cannot be planned in advance. However, you should allocate sufficient time for confirmation tests during planning (see Chap. 5).

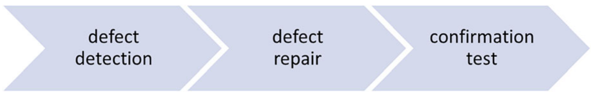

Fig. 2.17 Confirmation testing

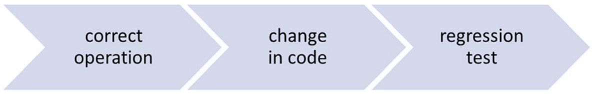

Fig. 2.18 Regression testing

Regression Testing

Regression testing 🕮 checks whether a change (fix or modification) made to one component of the code will inadvertently affect the behavior of other parts of the code in the same component, in other components of the same system, or even in other systems (see Fig. 2.18). In addition, changes to the environment, such as the introduction of a new version of the operating system or database management system, must be taken into account. The above unintended side effects are called *regressions* (from the word "regress," meaning a deterioration of something), and regression testing involves re-executing tests to detect these regressions.

Since regression tests are most often performed when another functionality is added or at the end of each iteration, these tests can usually—unlike confirmation tests—be perfectly scheduled. Since regression tests should be performed frequently and also evolve rather slowly, they are natural candidates for automation. Automating regression tests saves time and effort, as testers can devote the saved time to other activities, such as creating and executing new tests. Automation of this type of testing should start early in the project.

A common practical problem with regression tests is that their number grows very quickly. After a certain point, they take long enough to execute overnight, with results waiting for testers the next morning. Sometimes, however, a set of regression tests is not even able to execute overnight. In this case, a decision must be made to optimize the regression test execution. Possible solutions include:

- **Prioritization**—perform the most important, relevant tests first; those that fail to execute are less important, so there is little risk of a major failure going unnoticed.
- **Reduction**—remove some tests from the set of regression suite; the criterion for removal can, for example, be based on the test's ability to detect failures—if a test detects failures quite often, it should definitely stay, while if a test has never detected any failures, it can be removed from the set of regression tests after some time.
- **Mixed strategy**—the most important tests are executed on a scheduled basis, while the part that is unable to perform due to time considerations is executed less frequently, such as every other iteration.

Confirmation testing and regression testing can be performed at all test levels, especially in iterative and incremental SDLC models, new functionality, changes to existing functionality, and refactoring result in frequent code changes that entail appropriate testing. Given the continuous evolution of the system, confirmation testing and regression testing are very important, especially for systems related to the IoT, where individual objects (e.g., devices) are frequently updated or replaced.

Example While testing a book e-store (see previous section), a tester triggered a failure: the system allowed a user to register with the same email address that another previously registered user already has. Let us call "test A" the test that detected this issue. The tester filed a defect report and took care of other activities. Specifically, he ran test B (attempting to pay for a standard order), which passed. In the next iteration, a new component was implemented to report payment operations. After these changes, the tester performed test B again to see if the component affected the payment functionality itself. Moments later, the tester received a message that the defect he had reported earlier had been fixed. The tester executed test A again, which this time passed. The tester closed the defect.

In the above example, test A and test B were executed twice, in order: A, B, B, A.

- The first execution of test A and the first execution of test B do not apply to either regression testing or confirmation testing, as these tests were performed for the first time.
- The second execution of test B is an example of regression testing, since the intention of the tester is to check whether the implementation of a new component has broken something in the payment module.
- The second execution of test A is an example of confirmation testing, since the intention of the tester is to verify that test A, which caused a failure earlier, will now, after correction, pass.

Change-related tests can be performed at all test levels. Referring to the banking application example from Sect. 2.2.2.4, we can distinguish the following examples of change-related tests:

- For component testing, automated regression testing is designed on a per-component basis (these tests will then be included in the continuous integration framework).
- For component integration testing, tests are designed to confirm the effectiveness of fixes related to defects in interfaces as such fixes are placed in the code repository.
- For system testing, all testing for a given workflow is performed again if any of the screens covered by that workflow has changed.
- For system integration testing, testing of the application that works with the microservice for credit checking is performed daily (as part of the ongoing implementation of this microservice).
- For acceptance testing, all previously failed tests are performed again after the defect detected in acceptance testing is fixed.

2.3 Maintenance Testing

FL-2.3.1 (K2) Summarize maintenance testing and its triggers

The moment a system is released to a customer is not the end, but only an intermediate state of product development. Although many IT managers do not realize it, a lot of empirical software engineering studies show that the cost of system maintenance is usually significantly higher than the cost of developing it. This is due to a simple fact: it usually takes much longer to use software than to develop it, and during its use, many different situations can occur that require changes or corrections to the software. Examples of such situations are:

- The need for a patch to the system due to the discovery of an attack related to security vulnerability
- Adding, removing, or modifying program features
- Fixing the defect that caused the failure reported by the user
- The need to archive data due to software decommissioning
- The need to move the system to a new environment, caused, for example, by upgrading the version of the operating system

Once deployed in a production environment, the software or system therefore requires further maintenance. Various types of changes, such as those described above, are virtually inevitable. Moreover, maintenance is necessary to maintain or improve the required nonfunctional quality characteristics of the software or system throughout its life cycle—especially in terms of performance, compatibility, reliability, security, and portability.

It is good to remember that maintenance tests are performed *after* the software is released, during its operational use. Tests performed *before* the software is released to the customer cannot be considered maintenance tests. Maintenance, by definition, applies to a product already in use.

After any change is made in the maintenance phase, **maintenance testing** should be performed. The purpose of maintenance testing is both to verify that the change has been implemented successfully and to detect possible side effects (e.g., regressions) in the unchanged parts of the system (i.e., usually in most areas of the system). Maintenance testing therefore includes both the parts of the system that have been changed and the unchanged parts that may have been affected by the changes. Maintenance can be performed both on a scheduled basis (in connection with new versions) and on an unscheduled basis (in connection with *hot fixes*).

In light of the above considerations, it is clear that the most common type of maintenance testing will be regression testing. However, this is not the only type of maintenance testing. For example, if the software is to be replaced with another, either archiving or data migration tests should be performed. These may be entirely new tests that we will be performing only for the first time in the entire, perhaps years-long, history of using the application.

A maintenance release may require the execution of maintenance tests at multiple test levels and using different test types, depending on the extent of the changes made.

The scope of maintenance testing includes:

- The risk level associated with the change (e.g., the extent to which the changed software area communicates with other components or systems)
- The size of the existing system
- The size of the change made

There are several reasons for performing software maintenance and thus maintenance testing. This includes both planned and unplanned changes. The events that trigger maintenance can be divided into the following categories:

- **Modification**. This category includes, but is not limited to, planned enhancements (e.g., in the form of new software versions), corrective and emergency changes, changes to the operating environment (e.g., planned operating system or database upgrades), software updates for COTS software, and patches that fix security defects and vulnerabilities.
- **Upgrade or migration**. This category includes, but is not limited to, the transition from one platform to another, which may involve testing of the new environment and the changed software or data conversion testing (when migrating data from another application to a maintained system).
- **Retirement**. This category refers to the situation in which an application is withdrawn from use. This may require testing migration or archiving of data if there is a need to store it for an extended period of time. Testing of recovery procedures after archiving for an extended period of time may also be necessary. Additionally, regression testing may need to be included to ensure that functionality remaining in use continues to work properly.

In the case of systems related to the IoT, maintenance testing may be necessary after the introduction of completely new or modified elements in the system, such as hardware devices or software services. During maintenance testing of such systems, special emphasis is placed on integration testing at various levels (e.g., at the network and application levels) and on security aspects, especially with regard to personal data.

Example A company produces a CRM (*Customer Relationship Management*) system. This system is based on a relational database, which was not very well designed—it contains a lot of redundant data. The application refers to the database through a set of well-defined functions, rather than directly by constructing queries. The architect decided that the structure of the database should be improved—the database schema should be transformed to the so-called third normal form. The change will require installation on each user terminal.

This is a maintenance triggering event, related to modification (corrective change) and, in a sense, migration (since the database structure will be modified). Therefore, you should perform the following maintenance tests related to the installation of the patch (prior to its implementation):

- Regression tests directly related to the use of functions that communicate with the database
- Tests of data migration and recovery and return to the old database structure in case it turns out that for some reason, the new database structure on the customer's computer does not work as it should

Impact Analysis

Impact analysis allows us to evaluate changes made to the maintained version in terms of both intended effects and expected or potential side effects and allows us to identify areas of the system that will be affected by the changes. In addition, it can help identify the impact of the change on existing tests. The side effects of the change and the areas of the system that may be affected by it should be tested for regression, an activity that may be preceded by an update of existing tests affected by the change.

An impact analysis can be conducted before a change is made to determine whether the change should actually be made (given the potential consequences for other areas of the system).

Conducting an impact analysis can be difficult if:

- Specifications (e.g., business requirements, user stories, architecture) are outdated or unavailable
- Test cases have not been documented or are outdated
- The bidirectional traceability between tests and the test basis has not been established
- Tool support is nonexistent or inadequate
- The people involved do not have knowledge of the field and/or the system in question
- Too little attention has been paid to the system's quality characteristics in terms of maintainability during software development

Difficulties in performing impact analysis are quite common. This is because maintenance is often done months or years after the software is released and often the people who created the original version of the system are no longer in the team or organization and the documentation is outdated and obsolete.

Example. It is necessary to implement a software patch, related to a change in the law on tax calculation. The change affects component X. Using bidirectional traceability, it turns out that module X is associated with risks R1, R3, R6, and R8, all of which have a very high risk level (i.e., they are critical).

(continued)

The remaining risks—R2, R4, R5, and R7 have a low level (i.e., they are negligible). In addition, component X communicates with four other components among all eight modules in the system. This simple risk analysis shows that the change has a high risk, and its impact on the system is significant.

Sample Questions

Question 2.1
(FL-2.1.1, K2)

You are a tester in a project to develop aircraft autopilot software. The project is being conducted according to the V-model.

Which of the following sentences BEST describes the consequences associated with choosing this model?

A. The emphasis in testing will be on experience-based test techniques.
B. The test team will not conduct static testing.
C. Upon completion of the first iteration, a working prototype of the system will be available.
D. In the initial phases, testers participate in requirements reviews, test analysis, and test design.

Choose one answer.

Question 2.2
(FL-2.1.2, K1)

One good testing practice says that for every development activity, there should be an associated test activity. Which SDLC model shows this principle *explicitly*?

A. Iterative model.
B. Boehm's spiral model.
C. Scrum.
D. V-model.

Choose one answer.

Question 2.3
(FL-2.1.3, K1)

Which test-first approach uses acceptance criteria for user stories as the basis for deriving test cases?

A. TDD.
B. ATDD.
C. FDD.
D. BDD.

Choose one answer.

Question 2.4
(FL-2.1.4, K2)
Which of the following statements describes how the DevOps approach supports TESTING?

A. DevOps makes it possible to shorten the software release cycle by using an automated test data generation for component testing.
B. DevOps enables quick feedback to the developer on the quality of the code they commit to the repository.
C. DevOps enables the automatic generation of test cases for new code committed by a developer to the repository.
D. DevOps makes it possible to reduce the time required for release planning and iteration planning.

Choose one answer.

Question 2.5
(FL-2.1.5, K2)
Which of the following is an example of using the shift-left approach?

A. Application of acceptance test-driven development (ATDD).
B. Basing testing on exploratory testing.
C. Creating GUI prototypes during the requirements elicitation phase.
D. Continuous monitoring of product quality after release to the customer.

Choose one answer.

Question 2.6
(FL-2.1.6, K2)
Which of the following sentences best describes the activities of a tester attending a retrospective meeting?

A. The tester raises only test-related issues. All other topics will be raised by any of the other participants.
B. The tester is an observer, making sure that the meeting follows the principles of retrospectives and the values of agile approach.
C. The tester provides feedback and other information on all activities performed by the team during the completed iteration.
D. The tester gathers the information provided by other meeting participants to design tests for the next iteration based on this information.

Choose one answer.

Question 2.7
(FL-2.2.1, K2)
You are testing the aircraft autopilot system. You want to test the correctness of communication between the geolocation component and the engine controller component.

Which of the following is BEST example of a test basis for designing these tests?

A. Detailed design of the geolocation component.
B. Architecture design.
C. Risk analysis report.
D. Regulations from the avionics/aircraft law area.

Choose one answer.

Question 2.8
(FL-2.2.2, K2)
Which of the following is an example of a nonfunctional test?

A. Covering a certain combination of conditions and observing the executed actions to check the correct implementation of a certain business rule.
B. Covering the result “TRUE” in the decision “IF (x > 5) THEN . . .” in the code.
C. Verifying that the system correctly validates the syntax of the email address entered by the user in the user registration form.
D. Verifying that the system login time is less than 5 ms when 1000 users are already logged in at the same time.

Choose one answer.

Question 2.9
(FL-2.2.3, K2)
Which of the following tests usually cannot be scheduled in advance?

A. Regression tests.
B. Operational acceptance tests (OAT).
C. User acceptance tests (UAT).
D. Confirmation tests.

Choose one answer.

Question 2.10
(FL-2.3.1, K2)
During the testing of IoT (Internet of Things) software, a defect was discovered, but it was not fixed due to the close timing of the software’s release. After the release, it has not caused a failure at the customer until now. A month after the release, the team decided to fix the defect.

Which of the maintainability triggering events are we dealing with in this situation?

A. Software upgrade.
B. Software migration.
C. Software modification.
D. Adding new functionality to the system.

Choose one answer.

Chapter 3 Static Testing

Keywords

Anomaly	A condition that deviates from expectation. *After* ISO 24765
Dynamic testing	Testing that involves the execution of the test item. *After* ISO 29119-1
Formal review	A type of review that follows a defined process with a formally documented output. *After* ISO 20246
Informal review	A type of review that does not follow a defined process and has no formally documented output
Inspection	A type of formal review that uses defined team roles and measurement to identify defects in a work product and improve the review process and the software development process. *After* ISO 20246
Review	A type of static testing in which a work product or process is evaluated by one or more individuals to detect defects or to provide improvements
Static analysis	The process of evaluating a component or system without executing it, based on its form, structure, content, or documentation. *After* ISO 24765
Static testing	Testing that does not involve the execution of a test item
Technical review	A formal review by technical experts that examine the quality of a work product and identify discrepancies from specifications and standards
Walkthrough	A type of review in which an author leads members of the review through a work product and the members ask questions and make comments about possible issues. *After* ISO 20246. *Synonyms:* structured walkthrough

L. Stapp et al., *ISTQB® Certified Tester Foundation Level*,
https://doi.org/10.1007/978-3-031-42767-1_6

3.1 Static Testing Basics

FL-3.1.1 (K1)	Recognize types of products that can be examined by the different static test techniques
FL-3.1.2 (K2)	Explain the value of static testing
FL-3.1.3 (K2)	Compare and contrast static and dynamic testing

Static testing is a set of testing methods and techniques in which the component or system under test is not run (not executed). Static testing can also apply to non-executable work products other than software, such as design, documentation, specifications, etc.

The goals of static testing include, in particular, quality improvement, defect detection, and evaluation of such characteristics as readability, completeness, correctness, testability, or consistency of the work product under review. Thus, static testing can be used for both verification and validation of a work product.

In agile software development, testers, business representatives (customers, users), and developers work together during requirements development (e.g., writing user stories, creating examples, or participating in backlog refinement sessions). The goal of this collaborative work is to make sure that user requirements and related work products meet certain criteria, such as the definition of ready (see Sect. 5.1.3). Review techniques can be used to ensure that requirements are complete, understandable, and testable and, in the case of user stories, contain testable acceptance criteria. By asking the right questions, testers examine, challenge, and help improve proposed requirements.

Static testing can be divided into two major groups of techniques:

- Static analysis
- Reviews

Static analysis involves evaluating the work product under test (usually code, requirements, or design documents) using tools. The Foundation Level syllabus does not describe static analysis methods in detail. You will find more details in the Advanced Level syllabus "Technical Test Analyst" [29]. Examples of static analysis techniques include:

- Code measurement (e.g., measuring its size or cyclomatic complexity)
- Control flow analysis
- Data flow analysis
- Checking the compatibility of variable types, verification of the correct application of code writing standards (e.g., variable naming), etc.

Static analysis can identify problems before **dynamic testing**, requiring less effort because test cases are not required and the analysis is usually performed with tools. Static analysis is often included in continuous integration frameworks as one step of the automated deployment pipeline (see Sect. 2.1.4). Although largely used to detect specific code defects, static analysis is also widely used to assess the maintainability, performance, and vulnerability of code to security attacks.

Reviews are much more commonly used static testing techniques. Therefore, we will discuss them in more detail in Sect. 3.2.

3.1.1 Work Products Examinable by Static Testing

Static testing techniques can be applied to virtually any work product. The syllabus lists many such examples of work products subjected to static testing. A somewhat expanded list is given below:

- All kinds of specifications (business, functional requirements, non-functional requirements, etc.)
- Epics, user stories, acceptance criteria, and other types of documentation used in agile projects
- Architecture design
- Source code—perhaps excluding source code written by third parties to which we do not have access (e.g., in the case of commercial projects)
- All kinds of testware, including test plans, test procedures, test cases, automated test scripts, test data, risk analysis documents, and test charters
- User manuals, including built-in online help, system operator manuals, installation instructions, and release notes
- Websites (in terms of their content, structure, usability, etc.)
- Project documents (e.g., contracts, project plans, schedules, budgets)

While the review can apply to basically **any** work product, for it to make sense, the participants in the review must be able to read and analyze the product with understanding. Additionally, in the case of static analysis, the work products reviewed by this technique must have a formal structure against which the testing is done. Examples of such formal structures are:

- Source code (since every program is written in a programming language defined based on a formal grammar)
- Models (e.g., UML diagrams that have a specific syntax and formal rules for creating them)
- Text documents (because the text can be checked, e.g., against the grammar rules of the language)

Static analysis is most often applied to formalized work products, such as formal system architecture models or requirements (e.g., written in specification languages such as Z or UML). For documents written in natural language, static analysis can involve, for example, checking for readability, syntax, grammar, punctuation, or spelling.

3.1.2 Value of Static Testing

Static testing is usually not cheap (it requires manual checking of the product), but—if done correctly—it is effective and efficient. This is because it allows finding defects and problems very early in the SDLC that would be difficult to detect in later phases. This usually refers to design defects. This type of defect is usually very insidious, because a design defect undetected in the initial phases propagates very easily to subsequent phases and a faulty implementation is created based on a faulty design. The problem usually becomes apparent at the system or acceptance testing stage, where it can be very expensive to fix such a design defect.

Well-conducted static testing before dynamic test execution results in quick (and reasonably inexpensive) removal of problems that could be the source of other defects. As a result, fewer problems are detected in dynamic testing, so the total cost of dynamic testing is lower than if static testing was not performed. This use of static testing is in line with the principle of early testing and shift-left (see Sect. 1.3).

Static testing provides an opportunity to assess quality and build confidence in the reviewed work product. Stakeholders can assess whether the documented requirements describe their actual needs. Because static testing can be performed early in the SDLC, a common understanding of the product and its requirements is created among the stakeholders involved in static testing. This shared understanding also improves communication. For this reason, it is recommended for reviews to involve various stakeholders representing the broadest, most diverse perspectives on the work product under review.

Defects in code can be detected and corrected using static testing more efficiently than when dynamic testing is used, because in dynamic testing, the occurrence of a failure usually requires tedious analysis of the cause of the failure and spending a great deal of time identifying the defect causing it. This efficiency of static testing usually results in both fewer remaining defects in the code and a lower overall workload.

The effectiveness of static techniques is empirically proven. Jones and Bonsignour [31] provide a wealth of data showing that static techniques contribute significantly to overall product quality and to lower testing and maintenance costs. In the case of inspection (one type of review described in the next section), defect removal efficiency increases by an average of 85%!

The following is a list of benefits of static testing:

- Early, effective defect detection (and potential removal), even before dynamic testing begins—the ability to test even before a working software prototype is created.

- Identifying defects that are difficult to detect in subsequent dynamic testing. This is especially true for design and architectural defects.
- Identifying defects that are impossible to be detected in dynamic testing, such as locating infeasible (unreachable) code,[1] unused code, incorrect use or lack of use of design patterns in code, or all kinds of defects in non-executable work products, such as documentation.
- Preventing the occurrence of defects in design and code by detecting ambiguities, contradictions, omissions, oversights, redundant elements, or inconsistencies in documents such as the requirements specification or architecture design.
- Increasing the efficiency of programming work. Improving the design and code maintainability can be enforced, for example, by imposing a uniform standard for writing them. This makes the code easier to modify not only by the author but also by another developers, and the chance of introducing a defect when modifying the code decreases.
- Reducing the cost and time of software development, including testing—especially dynamic testing.
- Reducing the cost of quality throughout the software development cycle by reducing costs in the maintenance phase, as well as reducing failures in the software exploitation phase.
- Improving communication among team members through participation in reviews, including review meetings.

When evaluating the costs incurred for testing, the concept of "cost of quality" is introduced. This is the total cost incurred for quality activities, which consists of the costs of:

- Preventive activities (e.g., training costs)
- Detection (e.g., cost of testing)
- Internal failures (e.g., cost of fixing defects found in production)
- External failures (e.g., cost of fixing field defects found by users)

Although reviews can be costly to implement, the overall quality costs are usually much lower than if reviews are not performed, because less time and effort must be spent on defect removal later in the project. Participants in the review process also benefit from a better shared understanding of the product being reviewed.

[1] Identification of infeasible code is possible only in some cases, because in general the problem of reachability of a certain place in the code is a so-called undecidable problem. This means that there is no algorithm that, for *any* program and *any* place in its code, would answer the question of whether there are inputs to the program for which the control reaches that place in the code.

Example economic analysis of inspections

Consider a simplified economic simulation of whether implementing static testing in an organization pays off. In Scenario A, the team will perform only dynamic testing. In Scenario B, dynamic testing will be preceded by static testing. The following model is very simple and is only meant to illustrate how static testing can reduce the total cost of software development and maintenance.

Model assumptions:

- The team of testers consists of six people.
- The monthly cost of one tester salary is $7000.
- Static tests and dynamic tests last 4 months each, and the entire testing team participates. Static tests (if any) are performed before dynamic tests.
- The total number of software defects is 130.
- Static testing finds 50% of all existing defects (see, e.g., [31] for industry data).
- Dynamic testing finds 80% of all existing defects. The remaining 20% are field defects, discovered by the customer after the software is released (these values are derived from historical data).
- Fixing one defect found in static tests costs $500 on average (industry data).
- Fixing one defect found in dynamic testing costs $1800 on average (historical data).
- Fixing one field defect (found by the user after the software is released) costs $12,600 (historical data).

Scenario A: Without static testing

Number of defects detected in dynamic testing: 80% * 130 = 104
Number of field defects: 20% * 130 = 26

Cost of dynamic tests: 6 * $7000 * 4 = $168,000
Cost of defect removal before release: 104 * $1800 = $187,200
Cost of removing defects after release: 26 * $12,600 = $327,600

Total quality cost: $168,000 + $187,200 + $327,600 = $682,800

Scenario B: With static testing

Number of defects detected in static testing: 50% * 130 = 65
Number of defects detected in dynamic testing: 80% * (130 − 65) = 52
Number of field defects: 130 − (65 + 52) = 13

Cost of static testing: 6 * $7000 * 4 = $168,000
Cost of dynamic testing: 6 * $7000 * 4 = $168,000

(continued)

Cost of removing defects found in static testing: 65 * \$500 = \$32,500
Cost of defect removal before release: 52 * \$1800 = \$93,600
Cost of removing defects after release: 13 * \$12,600 = \$163,800

Total quality cost: \$168,000 + \$168,000 + \$32,500 + \$93,600 + \$163,800 = \$625,900

Schematically, the comparison of these two scenarios is shown in Fig. 3.1. Pre-release quality costs are higher in Scenario B (\$462,100 vs. \$355,200), due to high inspection costs. However, the gain comes after release, due to half the number of field defects in Scenario B, thanks to the use of reviews. The total cost thus turns out to be lower in Scenario B. The exact cost difference between Scenarios A and B is \$682,800 − \$625,900 = \$56,900.

Of course, in the analysis, we must take into account estimation errors, as well as the fact that the testing team needed four additional months to conduct static testing. Nevertheless, the analysis shows that the use of static testing can not only significantly improve the final quality of the product (26 field defects vs. 13) but also significantly reduce development costs. In the above simulation, we saved about \$57,000, so the costs in Scenario B are about 8.3% less than those in Scenario A.

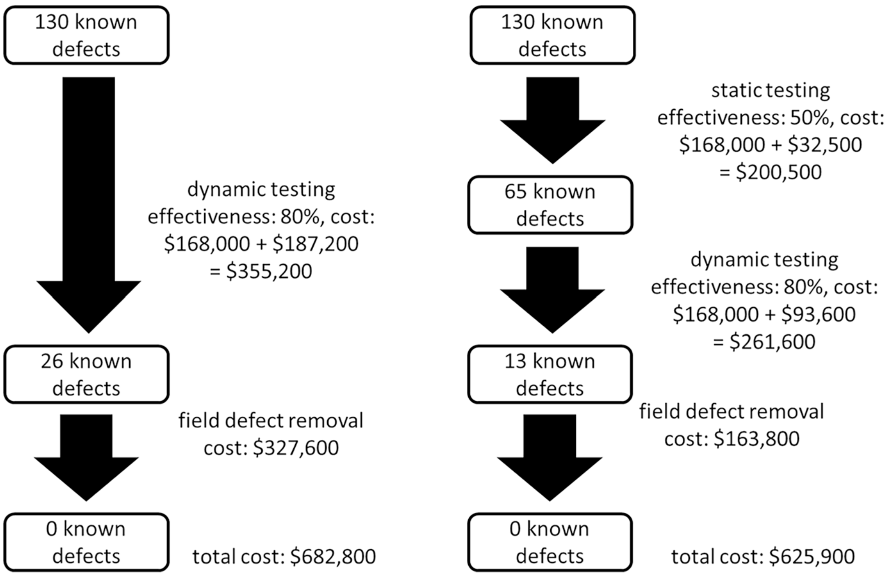
130 known defects
dynamic testing effectiveness: 80%, cost: \$168,000 + \$187,200 = \$355,200
26 known defects
field defect removal cost: \$327,600
0 known defects
total cost: \$682,800
130 known defects
static testing effectiveness: 50%, cost: \$168,000 + \$32,500 = \$200,500
65 known defects
dynamic testing effectiveness: 80%, cost: \$168,000 + \$93,600 = \$261,600
13 known defects
field defect removal cost: \$163,800
0 known defects
total cost: \$625,900

Fig. 3.1 Comparison of defect removal processes without and with static testing

3.1.3 Differences Between Static Testing and Dynamic Testing

Both static testing and dynamic testing have the same goal—to evaluate product quality and identify defects as soon as possible. These activities are complementary, as they enable the detection of different types of defects.

Figure 3.2 symbolically depicts the fundamental difference between static testing (top part of the figure) and dynamic testing (bottom part of the figure). With static testing, we are able to find a defect directly in the work product. We do not find failures, because with static techniques, by definition, we are not executing the software and a failure can only occur as a result of the system that is running.

In the case of dynamic testing, most often the first sign of malfunctioning software is, in turn, failure (the result of failing a test). When a failure is observed, a debugging process is launched, during which the source code is analyzed in order to find the defect responsible for causing this failure.

Sometimes, a defect in a work product can remain hidden for a very long time because it does not cause a failure. In addition, the path it is on may be rarely tested or difficult to access, making it not easy to design and execute dynamic test that would detect it. Static testing can find a defect with much less effort. On the other hand, there are defects that are much easier to detect with dynamic testing than static testing, such as memory leaks.

Another difference is that static testing can be used to increase the consistency and internal quality of work products, while dynamic testing focuses mainly on externally visible behavior.

In addition, only static testing can be applied to non-executable work products, such as source code or documentation. Dynamic testing, on the other hand, can only be performed against a running work product, that is, running software. A

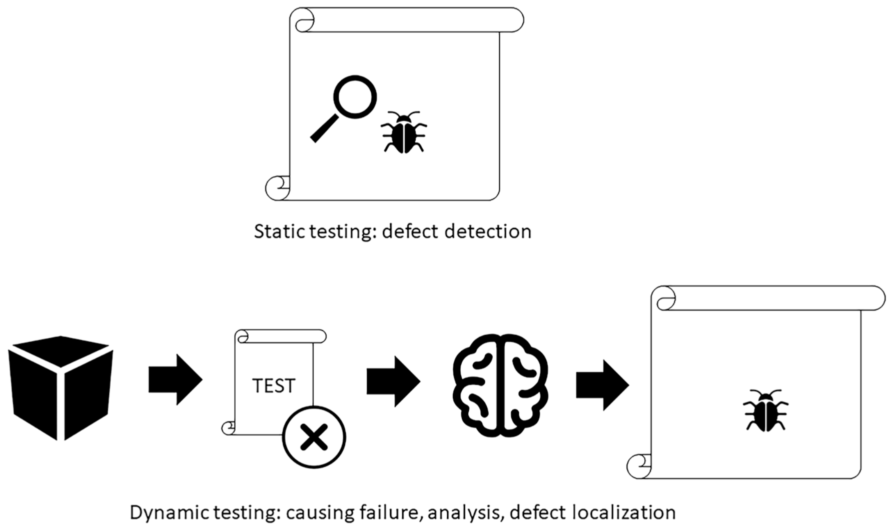
Static testing: defect detection
TEST
Dynamic testing: causing failure, analysis, defect localization

Fig. 3.2 Difference between static testing and dynamic testing

consequence of this fact is that dynamic testing can be used to measure some quality characteristics that are impossible to measure in static testing, because they depend on the running program. An example would be performance tests that measure the system's response time for a specific user request.

We mentioned earlier that static testing usually detects different defects than dynamic testing. Examples of typical defects that are easier and cheaper to detect and fix with static testing than with dynamic testing are:

- Defects in requirements (such as inconsistencies, ambiguities, omissions, contradictions, inaccuracies, omissions, repetitions, redundant elements)
- Design defects (e.g., inefficient algorithms or database structures, high *coupling*, [2] low *cohesion*, [3] poor code modularization)
- Specific types of code defects (e.g., variables with undefined values, variables declared but never used, inaccessible code, duplicated code, inefficiently implemented algorithms with too high time or memory complexity)
- Deviations from standards (e.g., lack of compliance with code development standards)
- Incorrect interface specifications (e.g., use of different units of measure in the calling and called systems, incorrect type or incorrect order of parameters passed to an API function call)
- Security vulnerabilities (e.g., susceptibility to buffer overflow attacks, SQL injection, XSS [4] (*cross-site scripting*), DDoS attack [5])
- Gaps or inaccuracies in traceability or coverage (e.g., no tests that match acceptance criteria for a given user story)

[2] Coupling refers to the degree to which one software component depends on another and expresses the level of interdependence between modules. High coupling means that a change in one module may require a change in other modules, which can lead to higher maintenance and development costs and make it more difficult to modify and expand the system. Low coupling means that components are more independent of each other, making them easier to manage.

[3] Cohesion refers to the degree to which components within a module are interrelated and pursue a common goal. High cohesion means that the components within a module are focused on achieving the same, single goal (task or functionality), making them easy to understand, maintain, and develop. Low cohesion means that components within a module pursue different goals, which makes them difficult to understand, introduces unnecessary dependencies between them, and can lead to increased system maintenance and development costs and poor readability and scalability.

[4] XSS is a form of cyber-attack on websites to hack their users. The hacker places a malicious script on a seemingly friendly and secure website, making any person using it vulnerable to attack (according to home.co.uk).

[5] A DDoS attack involves launching an attack simultaneously from multiple locations at the same time (from multiple computers). Such an attack is carried out mainly from computers over which control has been taken using special software (e.g., bots and trojans) (according to home.pl).

Table 3.1 Static analysis results for code maintainability

Component	LOC	COMMENT	CC
`main`	129	0%	7
`split`	206	1%	12
`to_lower`	62	1%	6
`to_upper`	63	3%	6
`merge`	70	0%	15
`config`	42	15%	8
`stdio`	243	2%	21

Example The testing team wants to evaluate the ease and cost of maintaining the software after its release to the customer. To do so, it intends to use static analysis techniques. Three metrics have been identified that will be used to measure the source code:

- LOC—the number of executable lines of code in a component
- COMMENT—the percentage of annotated lines of code in a component
- CC—cyclomatic complexity[6] of a component

The results are shown in Table 3.1.
The static analysis led to the following conclusions:

- The components are rather small. The largest of them, `stdio`, has 243 lines of code. The size of the components should therefore not be a major problem regarding maintainability.
- The code is not commented enough. Except for the `config` module, in which 15% of the code is commented, in all other modules, the percentage of commented lines is between 0 and 3%. The `main` and `merge` modules do not contain comments at all. This fact should definitely be brought to the developers' attention.
- Three components, `split`, `merge`, and `stdio`, have high (above 10) cyclomatic complexity. They should be analyzed. This is especially true for `merge` in which the density of decision statements is very high: one decision for about five lines of code, which may suggest very poor readability of the source code. Failure to refactor these components could make it difficult to maintain them in the future.

[6]The cyclomatic complexity of a component is defined as the number of decision statements in the code of that module plus one. This is a very simple measure of the complexity of the code structure.

3.2 Feedback and Review Process

FL-3.2.1 (K1)	Identify the benefits of early and frequent stakeholder feedback
FL-3.2.2 (K2)	Summarize the activities of the review process
FL-3.2.3 (K1)	Recall which responsibilities are assigned to the principal roles when performing reviews
FL-3.2.4 (K2)	Compare and contrast the different review types
FL-3.2.5 (K1)	Recall the factors that contribute to a successful review

In the following paragraphs, we will discuss in detail:

- Advantages of early and frequent customer feedback
- Work product review process
- Roles and responsibilities in formal reviews
- Types of reviews
- Success factors in reviews

In the last subsection of this chapter (Sect. 3.2.6), we discuss techniques for reviewing work products. This section was present in the Foundation Level syllabus v3.1 but was removed in the new one (v4.0). Nonetheless, we decided to leave it in the book as an extra section, because it discusses the important issue of how to *practically* approach the very activity of reviewing a work product. This stage is a central step in the review process (see Sect. 3.2.2), so it is useful to know the basic techniques of reviewing. The material presented in Sect. 3.2.6 is not examinable.

3.2.1 Benefits of Early and Frequent Stakeholder Feedback

Reviews are a form of providing early feedback to the team because they can be done early in the development cycle. However, feedback should not be limited to reviews alone but should be a common practice throughout the project's development cycle. Feedback can come from testers or developers, but equally important to the team will be that from the customer itself.

Early and frequent feedback allows for quick notification of potential quality issues and enables the team to respond quickly to the problem. If there is little stakeholder involvement during software development, the product being developed may not meet their original or current vision. Failure to deliver what stakeholders expect can result in costly rework, missed deadlines, and blame games and even lead to the complete failure of the project.

Frequent feedback from stakeholders during software development can prevent misunderstandings about requirements and ensure that requirement changes are

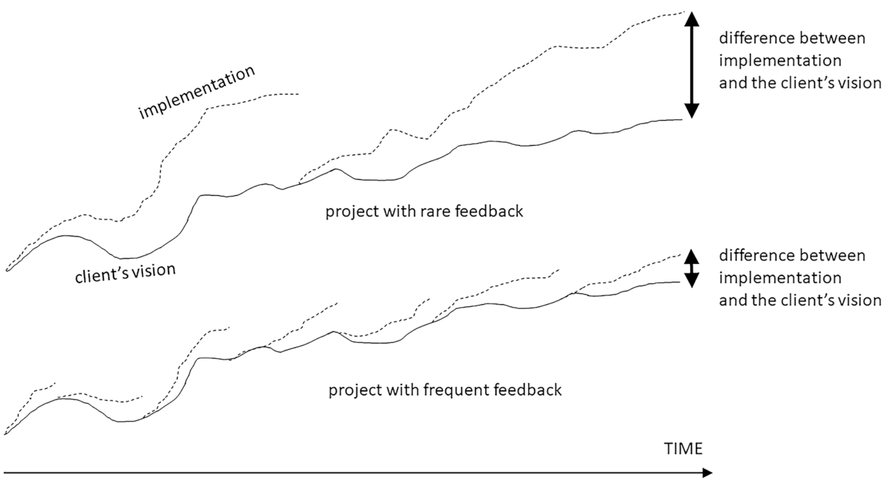
implementation
client's vision
project with rare feedback
difference between implementation and the client's vision
project with frequent feedback
difference between implementation and the client's vision
TIME

Fig. 3.3 Illustration of the benefit of frequent feedback

understood and implemented earlier. This helps the team better understand what they are building. It allows them to focus on those functions that deliver the most value to stakeholders and that have the most positive impact on agreed-upon risks.

Figure 3.3 symbolically shows the benefit of frequent feedback. The solid line represents the customer's vision, and the dashed line represents what the product actually looks like in implementation. These two visions diverge more and more over time. At moments of feedback, the implementation is adjusted to the customer's vision. The top graph shows a situation in which contact with the customer is sporadic and occurs only twice. It can be seen that the result is an implementation that diverges significantly from what the customer's vision was. The bottom graph represents a situation in which the customer frequently provides feedback to the team. This results in fewer differences between the customer's vision and the actual implementation, because much less time elapses between meetings with the customer than in the situation in the top chart. As a result, the final product is much more aligned with the customer's vision. In addition, it should be noted that the cost of "adjusting" the product to the customer's vision will be much higher in the case of infrequent feedback, due to the fact that the product will have to be modified and adjusted to a much greater extent than in the situation of frequent feedback.

3.2.2 Review Process Activities

Reviews vary in type, level of formalization, or objectives, but all types of reviews tend to have similar phases or groups of activities. The ISO 20246 standard *Software and systems engineering—Work product reviews* [6] defines a generic review process that provides a structured but flexible framework within which to tailor a