Three-point estimation
In this expert-based technique, experts perform three types of estimation: most optimistic (a), most likely (m), and most pessimistic (b). The final estimate (E) is their weighted arithmetic mean calculated as

$$E = (a + 4m + b)/6.$$

The advantage of this technique is that it allows experts to directly calculate also the measurement error:

$$SD = (b - a)/6.$$

For example, if the estimates (in person-hours) are $a = 6$, $m = 9$, and $b = 18$, the final estimate is 10 ± 2 person-hours (i.e., between 8 and 12 person-hours), because

$$E = (6 + 4 * 9 + 18)/6 = 10$$

and

$$SD = (18 - 6)/6 = 2.$$

Note that the estimation result will not always be equal to the most probable value *m*, because it will depend on how far away this value is from the optimistic and pessimistic values. The closer the value of *m* is to the extreme estimation value (a or b), the greater the distance between the value of m and the estimation result E.

The formula shown above is the most commonly used in practice. It is derived from the program evaluation and review technique (PERT) method. However, teams sometimes use other variants, weighting the *m* factor differently, for example, with a weight of 3 or 5. Then the formula takes the form $E = (a + 3m + b) / 5$ or $E = (a + 5m + b) / 7$.

For more information on these and other estimation techniques, see [45, 65, 66].

Example A team uses a planning poker to estimate the implementation and test effort of a certain user story. The planning poker is played in sessions according to the following procedure.

1. Each player has a full set of cards.
2. The moderator (generally the product owner) presents the user story to be estimated.
3. A brief discussion of the scope of work follows, clarifying ambiguities.
4. Each expert chooses one card.
5. At the sign of the facilitator, everyone simultaneously throws their card in the center of the table.
6. If everyone has chosen the same card, a consensus is reached, and the meeting ends—the result is the value chosen by all team members. If, after a fixed number

of iterations, the team has not reached a consensus, the final result is determined according to the established, accepted variant of the procedure (see below).
7. If the estimates differ, the person who chose the lowest estimate and the person who chose the highest present their point of view.
8. If necessary, a brief discussion follows.
9. Return to point 4.

Planning poker, like, indeed, any estimation technique based on expert judgment, will only be as effective as the good experts who participate in the estimation session are (see Fig. 5.5).

Therefore, people who estimate should be experts, preferably with years of experience.

The team has the following possible options for determining the final estimate when consensus has not been reached after a certain number of poker iterations. Assume that the team performing the effort estimation for a certain user story consists of five people and each participant has a full deck of cards with values 0, 1, 2, 3, 5, 8, 13, 20, 40, and 100. Assume that in the last round, five people simultaneously present a card of their choice. The results are:

3, 8, 13, 13, 20.

Variant 1 The result is averaged, and the final estimate is the average (11.4) rounded to the nearest value from the deck, which is 13.

Variant 2 The most common value is chosen, which is 13.

Variant 3 The result is the median (middle value) of the values 3, 8, 13, 13, and 20, that is, 13.

Variant 4 In addition to providing an estimate, each person also rates the degree of their confidence in that estimate, on a scale of 1 (high uncertainty) to 5 (high confidence). The degree of certainty can reflect, for example, the level of experience and expertise in the area in which the estimate is made. Let's assume that estimators 3, 8, 13, 13, and 20 rated the certainty of their choice at 4, 5, 2, 3, and 2, respectively. The score is then calculated as a weighted average:

$$\begin{aligned}&(3^*4 + 8^*5 + 13^*2 + 13^*3 + 20^*2)/(4 + 5 + 2 + 3 + 2)\\ &= (12 + 40 + 26 + 39 + 40)/16 = 157/16 = 9.8.\end{aligned}$$

The result can then be rounded to the nearest value on the scale, which is 8. The advantage of this approach is that the results of those more convinced of the correctness of their estimation weigh more in the averaged assessment. Of course, the accuracy of the method will depend in particular on the accuracy of the individual evaluators' confidence scores.

The above options do not exhaust all possible ways of estimation. Each team can adopt its own individual rules for determining the final value of the parameter that is subjected to the estimation procedure. It should also be borne in mind that all such

Fig. 5.5 Humorously about planning poker (www.commitstrip.com/en/2015/01/22/poker-planning)

variants for selecting the final answer in the absence of consensus are always subject to some error. For example, taking the median (13) from the values 8, 13, 13, 13, 13, and 13 will be subject to less error than, for example, from the values 1, 8, 13, 40, and 100, due to the much greater variance of the results in the latter case.

Factors affecting the test effort

Factors affecting the testing effort include:

- Product characteristics (e.g., product risks, quality of specifications (i.e., test basis), product size, complexity of product domain, requirements for quality characteristics (e.g., safety and reliability), required level of detail in test documentation, regulatory compliance requirements)
- Characteristics of the software development process (e.g., stability and maturity of the organization, SDLC model used, test approach, tools used, test process, time pressure)
- Human factors (e.g., skills and experience of testers, team cohesion, manager's skills)
- Test results (e.g., number, type, and significance of defects detected, number of corrections required, number of tests failed)

Information on the first three groups of factors is generally known in advance, often before we start the project. Test results, on the other hand, are a factor that works "from the inside": we cannot factor test results into the estimation before test execution. Their results can therefore significantly affect the estimation later in the project.

5.1.5 Test Case Prioritization

Once test cases and test procedures are created and organized into test suites, these suites can be arranged into a test execution schedule, which defines the order in which they are to be executed. Various factors should be taken into account when prioritizing test cases (and therefore the order in which they are executed).

A schedule is a way of planning tasks over time. The schedule should take into account:

- Priorities
- Dependencies
- The need for confirmation and regression tests
- The most effective order in which to perform the tests

When creating a test execution schedule, it is important to consider:

- The dates of the core activities within the project timeframe
- That the schedule is within the timeframe of the project schedule
- The milestones: start and end of each stage

One of the most typical graphical ways of presenting the schedule is the so-called Gantt chart (see Fig. 5.6). In this chart, tasks are represented as rectangles expressing their duration, and the arrows define different types of relationships between tasks. For example, if an arrow goes from the end of rectangle X to the beginning of rectangle Y, it means that task Y can only start when task X has finished.

Well-planned tests should have what is known as a linear trend—if you connect the "start of test project" and "end of tests" milestones from Fig. 5.6 with a line, the tasks performed should line up along this line. This means that the test-related tasks

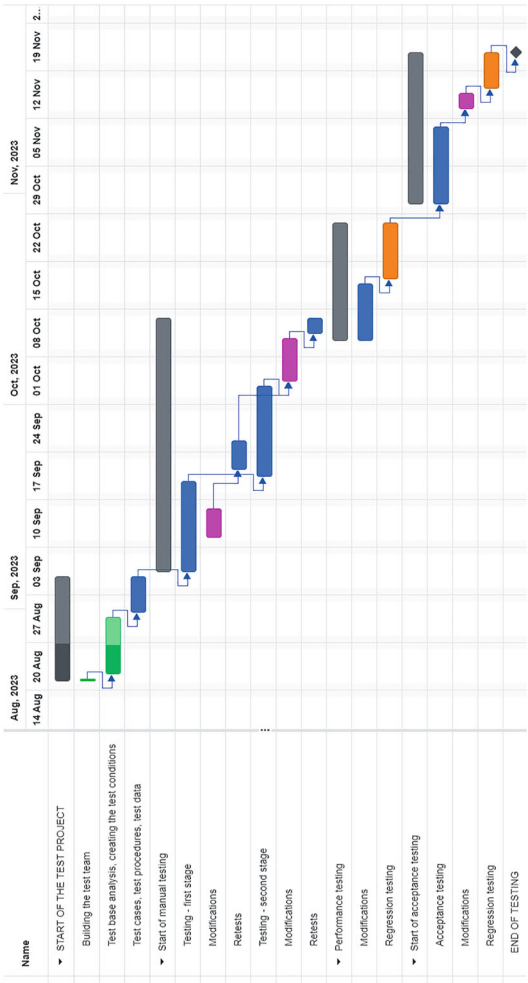

Fig. 5.6 Schedule in the form of a Gantt chart

are performed in such a way that the testing team works in a constant pace, without unnecessary time chasing.

Practice shows that the final testing period (>95% of cases) requires the maximum number of resources to be committed in the final testing phase (the final phase of the project)—see Fig. 5.7.

The most common test case prioritization strategies are as follows.

- **Risk-based prioritization**, where the order of test execution is based on the risk analysis results (see Sect. 5.2.3). Test cases covering the most important risks are executed first.
- **Coverage-based prioritization**, where the order of test execution is based on coverage (e.g., code coverage, requirements coverage). Test cases achieving the highest coverage are executed first. In another approach, called additional coverage-based prioritization, the test case achieving the highest coverage is executed first. Each subsequent test case is the one that achieves the highest additional coverage.
- **Requirements-based prioritization**, where the order in which tests are executed is based on requirements priorities that are linked to corresponding test cases. Requirements priorities are determined by stakeholders. Test cases related to the most important requirements are executed first.

Ideally, test cases should be ordered to execute based on their priority levels, using, for example, one of the prioritization strategies mentioned above. However, this practice may not work if the test cases or functions being tested are dependent. The following rules apply here:

- If a higher-priority test case depends on a lower-priority test case, the lower-priority test case should be executed first.
- If there are interdependencies between several test cases, the order of their execution should be determined regardless of relative priorities.
- Confirmation and regression test executions may need to be prioritized.
- Test execution efficiency versus adherence to established priorities should be properly balanced.

The order in which tests are executed must also take into account the availability of resources. For example, the required tools, environments, or people may only be available within a certain time window.

Example We are testing a system whose CFG is shown in Fig. 5.8. We are adopting a coverage-based test prioritization method. The criterion we use will be statement coverage. Let us assume that we have four test cases TC1–TC4 defined in Table 5.1.

This table also gives the coverage that each test case achieves. The highest coverage (70%) is achieved by TC4, followed by TC1 and TC2 (60% each), and the lowest by TC3 (40%). Accordingly, the test execution priority will be as follows: TC4, TC2, TC1, and TC3, where TC1 and TC2 can be executed in any order as they achieve the same coverage.

Fig. 5.7 Final test period

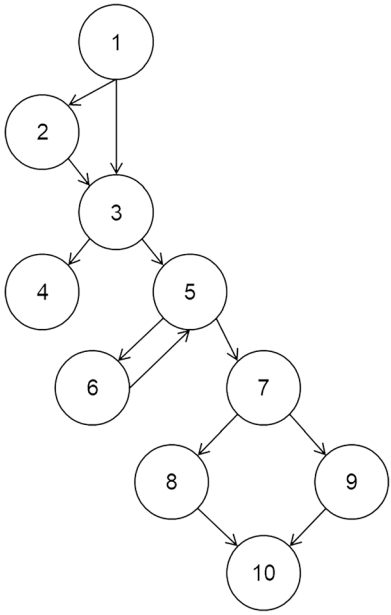

Fig. 5.8 CFG of the system under test

Table 5.1 Existing test cases and the coverage they achieve

Test case	Exercised path	Coverage	Priority
TC1	1→3→5→7→8→10	60%	2
TC2	1→3→5→7→9→10	60%	2
TC3	1→2→3→4	40%	3
TC4	1→3→5→6→5→7→8→10	70%	1

However, note that the execution of TC1, after the execution of TC4 and TC2, will not cover any new statements. So let us apply a variant of prioritization based on additional coverage. The highest priority will be given, as in the previous variant, to the test case achieving the highest coverage, namely, TC4. Now let's see how many *additional statements* not covered by TC4 are covered by the other test cases.

The calculation is shown in Table 5.2 in the penultimate column. TC1 does not cover any additional statements, because the statements it covers (i.e., 1, 3, 5, 7, 8, 10) are already covered by TC4. TC2, relative to TC4, covers additionally statement 9, so it achieves 10% additional coverage (covered one new statement out of ten total). TC3, relative to TC4, additionally covers two statements, 2 and 4, resulting in an additional 20% coverage (covered two new statements out of ten total). Thus, the next test case after TC4 covering the most new statements not yet covered is TC3. We now repeat the above analysis counting the additional coverage relative to the coverage achieved by TC4 and TC3. The calculation is shown in the last column of Table 5.2. TC1 covers nothing new. TC2, on the other hand, covers one new statement that TC4 and TC3 did not cover, namely, 9. So, it achieves an

Table 5.2 Calculation of additional coverage achieved by testing

Test case	Exercised path	Coverage	Additional coverage after TC4	Additional coverage after TC4 and TC3
TC1	1→3→5→7→8→10	60%	0%	0%
TC2	1→3→5→7→9→10	60%	10% (9)	10% (9)
TC3	1→2→3→4	40%	20% (2, 4)	–
TC4	1→3→5→6→5→7→8→10	70%	–	–

additional coverage of 10%. Thus, we obtained the following prioritization according to additional coverage:

TC4 → TC3 → TC2 → TC1.

Note that this order differs from the order resulting from the previous variant of the method. Its advantage is that we get coverage of all statements very quickly (after just the first three tests). In the earlier variant (with the order TC4→TC2→TC1→TC3), statements 2 and 4 are covered only in the last, fourth test.

Another example shows that sometimes technical or logical dependencies between tests can disrupt the desired order of test execution and force us, for example, to execute a lower-priority test first, in order to "unlock" the possibility of executing a higher-priority test.

Example Assume that we are testing the functionality of a system for conducting driving tests electronically. Assume also that at the beginning of each test run, the system database is empty. The following test cases have been identified, along with their priorities:

TC1: Adding the examinee to the database. Priority: low
TC2: Conducting the exam for the examinee. Priority: high
TC3: Conducting a revision exam. Priority: medium
TC4: Making a decision and sending the exam results to the examinee. Priority: medium

The test case with the highest priority here is TC2 (conducting the exam), but we cannot run it before we add the examinee to the database. So even though TC2 has a higher priority than TC1, the logical dependency that occurs requires us to execute TC1 first, which will "unlock" test execution for the other test cases. The next test case to run would be TC2, as the highest-priority task. At the very end, we can execute TC3 and TC4. Note that in order to pass a revision exam, the examinee must know that they failed the original exam. Therefore, from the perspective of a particular user, it makes sense to execute TC3 only after completing TC4. Thus, the final test execution is as follows:

TC1 → TC2 → TC4 → TC3.

Table 5.3 Tasks with their priorities and dependencies

Task	Priority (1 = high, 5 = low)	Dependent on
1	2	5
2	4	
3	1	2, 4
4	1	2, 1
5	3	6
6	3	7
7	5	

In many situations, it may happen that the order in which to perform actions depends on several factors, such as priority and logical and technical dependencies. There is a simple method for dealing with such tasks, even if the dependencies seem complicated. It consists of aiming for the earliest possible execution of higher-priority tasks, but if they are blocked by other tasks, we must execute those tasks first. Within these tasks, we also set their order according to priorities and dependencies. Let us consider this method with a concrete example.

Example A list of tasks with their priorities and dependencies is given in Table 5.3.

We start by establishing an order that depends solely on priorities, without looking at dependencies, with tasks of equal priority grouped together:

$3, 4 \rightarrow 1 \rightarrow 5, 6 \rightarrow 2 \rightarrow 7.$

Within the first group, task 3 is dependent on task 4. We therefore clarify the order of tasks 3 and 4:

$4 \rightarrow 3 \rightarrow 1 \rightarrow 5, 6 \rightarrow 2 \rightarrow 7.$

We check now if we can perform the tasks in this order. It turns out that we can't: task 4 depends on tasks 2 and 1, with task 1 having a higher priority than task 2. So we move both these tasks before task 4 with their priorities preserved, and after task 4, we keep the original order of the other tasks:

$1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5, 6 \rightarrow 7.$

Can we perform the tasks in this order? Still no: task 1 is dependent on task 5, this on task 6, and this in turn on task 7. So we need to move tasks 5, 6, and 7 in front of task 1 with their dependencies (note that here the priorities of tasks 5, 6, and 7 do not matter—the order is forced by the dependency):

$7 \rightarrow 6 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 3.$

Note that at this point, we can already perform the full sequence of tasks, 7, 6, 5, 1, 2, 4, and 3, since each task in the above sequence does not depend on any other task or depends only on the tasks preceding it.

5.1.6 Test Pyramid

Test pyramid 🕮 is a model showing that different tests can have different "granularity." The test pyramid model supports the team in test automation and test effort allocation. The layers of the pyramid represent groups of tests. The higher the layer, the lower the test granularity, test isolation, test execution speed, and cost of quality.

The tests in the bottom layer of the test pyramid are small, isolated, and fast and check a small piece of functionality, so you usually need a lot of them to achieve reasonable coverage. The top layer represents large, high-level end-to-end tests. These high-level tests are slower than the tests in the lower layers and usually check a large chunk of functionality, so you usually only need a few of them to achieve reasonable coverage.

The number and naming of layers in a test pyramid can vary. For example, the original test pyramid model [67] defines three layers: "unit tests" (called component tests in the ISTQB® syllabus), "service tests," and "UI tests" (see Fig. 5.9). Another popular model defines unit (component) tests, integration tests, and end-to-end tests.

The cost of test execution from a lower layer is usually much less relative to the cost of test execution from a higher layer. However, as mentioned earlier, a single test from a lower layer usually achieves much less coverage than a single test from a higher layer. Therefore, in practice, in typical development projects, the test effort is well reflected by the test pyramid model. The team typically writes a lot of low-level tests and few high-level tests. However, keep in mind that the number of tests from different levels will always be driven by the broad context of the project and product. It may happen, for example, that most of the tests the team will perform are integration and acceptance tests (e.g., in the situation of integrating two products purchased from a software house; in this case, there is no point in performing component testing at all).

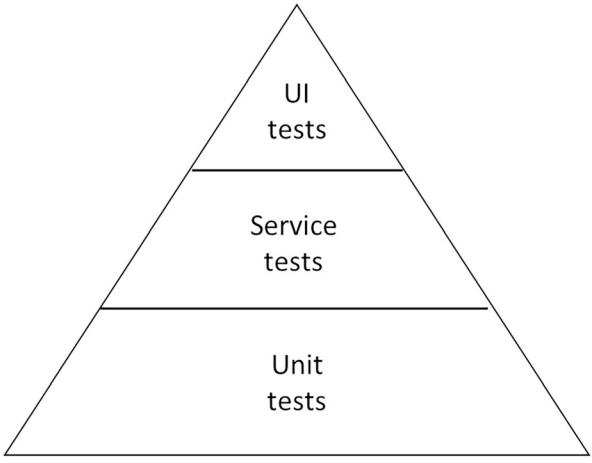

Fig. 5.9 Test pyramid

Even though the test pyramid is widely known, there are still quite a few organizations that flip this pyramid upside down, by focusing on the more complex (UI) tests. A variant is the tilted pyramid, where specific tests and series of tests slice through the pyramid, following a functional approach.

5.1.7 Testing Quadrants

The **testing quadrants** 🕮 model, defined by Brian Marick [68], [21], aligns test levels with relevant test types, activities, techniques, and work products in agile development approaches. The model supports test management in ensuring that all important test types and test levels are included in the software development process and in understanding that some test types are more related to certain test levels than others. The model also provides a way to distinguish and describe test types to all stakeholders, including developers, testers, and business representatives. The testing quadrant model is shown in Fig. 5.10.

In the testing quadrants, tests can be directed at the needs of the business (user, customer) or the technology (developer, manufacturing team). Some tests validate software behavior and hence support the work done by the agile team; others verify (critique) the product. Tests can be fully manual, fully automated, a combination of

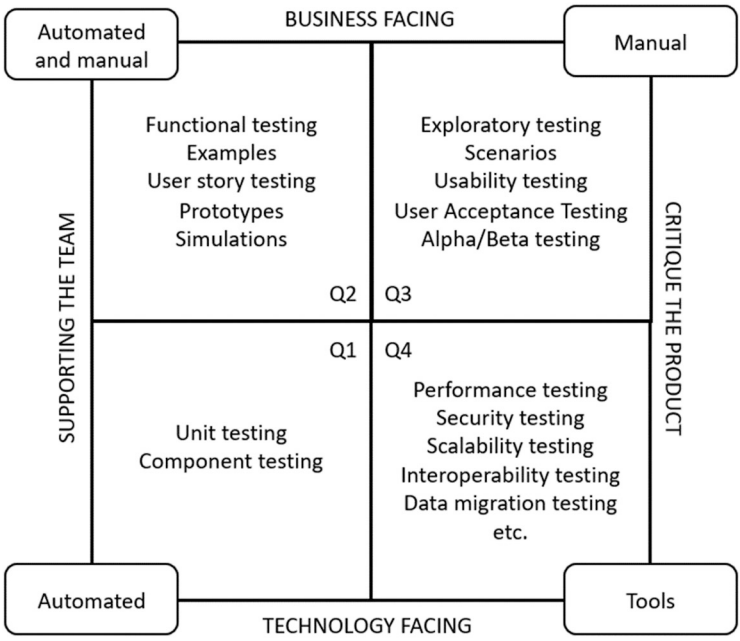

Fig. 5.10 Testing quadrants

manual and automated, or manual but supported by tools. The four quadrants are as follows:

Quadrant Q1
Quadrant Q1 describes the component test level, oriented toward technology and supporting the team. This quadrant includes component testing. These tests should be automated as much as possible and integrated into the continuous integration process.

Quadrant Q2
Quadrant Q2 describes the system testing level, which is considered business-oriented and team supporting. This quadrant includes functional tests, examples (see Sect. 4.5), user story testing, prototypes, and simulations. These tests check acceptance criteria and can be manual or automated. They are often created during the development of user stories, thus improving their quality. They are useful when creating automated regression test suites.

Quadrant Q3
Quadrant Q3 describes a business-oriented acceptance testing level that includes product critique tests using realistic scenarios and data. This quadrant includes exploratory testing, scenario-based testing (e.g., use case-based testing) or process flow testing, usability testing, and user acceptance testing, including alpha and beta testing. These tests are often manual and are user-oriented.

Quadrant Q4
Quadrant Q4 describes a technology-oriented acceptance testing level that includes "product critique" tests. This quadrant contains most non-functional tests (except usability tests). These tests are often automated.

During each iteration, tests from any or all quadrants may be required. Testing quadrants refer to dynamic testing rather than static testing.

5.2 Risk Management

FL-5.2.1 (K1)	Identify risk level by using risk likelihood and risk impact
FL-5.2.2 (K2)	Distinguish between project risks and product risks
FL-5.2.3 (K2)	Explain how product risk analysis may influence thoroughness and scope of testing
FL-5.2.4 (K2)	Explain what measures can be taken in response to analyzed product risks

Organizations face many internal and external factors that make it uncertain if and when they will achieve their goals [7]. **Risk management** 🕮 allows organizations to increase the likelihood of achieving their goals, improve product quality, and increase stakeholder confidence and trust.

The risk level is usually defined by risk probability and risk impact (see Sect. 5.2.1). Risks, from the tester's point of view, can be divided into two main groups: project risks and product risks (see Sect. 5.2.2). The main risk management activities include:

- Risk analysis (consisting of risk identification and risk assessment; see Sect. 5.2.3)
- Risk control (consisting of risk mitigation and risk monitoring; see Sect. 5.2.4)

An approach to testing in which test activities are managed, selected, and prioritized based on risk analysis and risk control is called **risk-based testing** 🕮.

One of the many challenges in testing is the proper selection and prioritization of test conditions. Risk is used to properly focus and allocate the effort required during testing these test conditions. It is used to decide where and when to start testing and to identify areas that need more attention. Testing is used to reduce risk, that is, to reduce the likelihood of an adverse event or to reduce the impact of an adverse event. Various forms of testing are one of the typical risk mitigation activities in software development projects (see Sect. 5.2.4). They provide feedback on identified and remaining (unresolved) risks. Early product risk analysis (see Sect. 5.2.3) contributes to project success.

Risk-based testing contributes to reducing product risk levels and provides comprehensive information to help decide whether a product is ready for release (this is one of the main goals of testing; see Sect. 1.1.1). Typically, the relationship between product risk and testing is controlled through a bi-directional traceability mechanism.

The main benefits of risk-based testing are:

- Increasing the probability of discovering defects in order of their priority by performing tests in order related to risk prioritization
- Minimizing residual product risk after release by allocating test effort according to risk
- Reporting residual risk by measuring test results in terms of levels of related risks
- Counteracting the effects of time pressure on testing and allowing the testing period to be shortened with the least possible increase in product risk associated with a reduction in the time allotted for testing

To ensure that the likelihood of product failure is minimized, risk-based testing activities provide a disciplined approach to:

- Analyze (and regularly reassess) what can go wrong (risk)
- Determine which risks are important enough to address
- Implement measures to mitigate these risks
- Design contingency plans to deal with risks when they occur

In addition, testing can identify new risks, help determine which risks should be mitigated, and reduce risk-related uncertainty.

5.2.1 Risk Definition and Risk Attributes

According to the ISTQB® Glossary of Testing Terms [42], **risk** 🕮 is a factor that may result in negative consequences in the future; it is usually described by *impact* and *likelihood* (probability). Therefore, **risk level** 🕮 is determined by the likelihood of an adverse event and the impact—or consequences (the harm resulting from the event). This is often defined by the equation:

$$\text{Risk level} = \text{Risk likelihood} * \text{Risk impact}.$$

This equation can be understood symbolically (as described above) or quite literally. We deal with the latter case in the so-called quantitative approach to risk, where probability and impact are expressed numerically—probability as a number in the interval (0, 1), while impact, in money. The impact represents the amount of loss we will suffer when the risk materializes.

Example An organization defines a risk level as the product of its probability and impact. A risk has been identified that a key report is generated too slowly when there are a large number of simultaneously logged-in users.

Since the requirement for high performance is critical in this case, the impact of this risk was estimated to be very high. Taking into account the number of users who could experience delays in report generation and the consequences of these delays, the risk impact was estimated at $800,000.

The likelihood of poor program performance, on the other hand, was estimated to be very low, due to a fairly well-defined testing process, scheduled architecture reviews, and the team's extensive experience with performance testing. The risk likelihood was estimated at 5%.

Finally, the risk level according to the above equation was set at:

$$5\% \times \$800{,}000 = (5/100) \times \$800{,}000 = \$40{,}000.$$

5.2.2 Project Risks and Product Risks

There are two types of risks in software testing: project risks and product risks.

Project risks

Project risks 🕮 are related to project management and control. Project risks are risks that affect the success of the project (the ability of the project to achieve its objectives). Project risks include:

- Organizational issues (e.g., delays in delivering work products, inaccurate estimates, cost cutting, poorly implemented, managed, and maintained processes, including the requirements engineering or quality assurance process)
- Human issues (e.g., insufficient skills, conflicts, communication problems, staff shortages, lack of available subject matter experts)
- Technical issues (e.g., scope creep; poor tool support; inaccurate estimation; last-minute changes; insufficient requirements clarification; inability to meet requirements due to time constraints; failure to make the test environment available in time; late scheduling of data conversion; migration or provision of tools needed for this; defects in the manufacturing process; poor quality of project work products, including requirements specifications or test cases; accumulation of defects or other technical debt)
- Supplier-related issues (e.g., failure of third-party delivery, bankruptcy of the supporting company, delay in delivery, contractual problems)

Project risks, when they occur, can affect a project's schedule, budget, or scope, which in turn affects the project's ability to achieve its goals. The most common direct consequence of project risks is that the project is delayed, which brings further problems, such as increased costs due to the need to allocate more time to certain activities or the need to pay contractual penalties for late delivery of a product.

Product risks

Product risks 🕮 are related to the quality characteristics of a product, such as functionality, reliability, performance, usability, and other characteristics described, for example, by the ISO/IEC 25010 quality model [5]. Product risks occur wherever a work product (e.g., a specification, component, system, or test) may fail to meet the needs of users and/or stakeholders. Product risks are defined by areas of possible failure in the product under test, as they threaten the quality of the product in various ways. Examples of product risks include:

- Missing or inadequate functionality
- Incorrect calculations
- Failures during the operation of the software (e.g., the application hangs or shuts down)
- Poor architecture
- Inefficient algorithms
- Inadequate response time
- Bad user experience
- Security vulnerabilities

Product risk, when it occurs, can result in various negative consequences, including:

- End user dissatisfaction
- Loss of revenue
- Damage caused to third parties
- High maintenance costs

- Overloading the help desk
- Loss of image
- Loss of confidence in product
- Criminal penalties

In extreme cases, the occurrence of product risks can cause physical harm, injury, and even death.

5.2.3 Product Risk Analysis

The purpose of **risk analysis** 🕮 is to provide risk awareness to focus the testing effort in such a way as to minimize the product's residual risk level. Ideally, product risk analysis begins early in the development cycle. Risk analysis consists of two major phases: risk identification and risk assessment. The product risk information obtained in the risk analysis phase can be used to:

- Test planning
- Specification, preparation, and execution of test cases
- Test monitoring and control

Early **product risk analysis** contributes to the success of the entire project, because product risk analysis makes it possible to:

- Identify specific test levels and test types to be performed
- Define the scope of the tests to be performed
- Prioritize testing (to detect critical defects as early as possible)
- Select appropriate test techniques that will most effectively achieve the set coverage and detect defects associated with identified risks
- Estimate for each task the amount of effort put into testing
- Determine whether other measures in addition to testing can be used to reduce (mitigate) risks
- Establish other activities unrelated to testing

Risk identification

Risk identification 🕮 involves generating a comprehensive list of risks. Stakeholders can identify risks using various techniques and tools, such as:

- Brainstorming (to increase creativity in finding risks)
- Risk workshops (to jointly conduct risk identification by various stakeholders)
- Delphi method or expert assessment (to obtain expert agreement or disagreement on risk)
- Interviews (to ask stakeholders about risks and gather their opinions)
- Checklists (to address typical, known, commonly occurring risks)
- Databases of past projects, retrospectives, and lessons learned (to address past experience)

- Cause-and-effect diagrams (to discover risks by performing root cause analysis)
- Risk templates (to determine who can be affected by risks and how)

Risk assessment

Risk assessment 🕮 involves categorizing identified risks; determining their likelihood, impact, and risk level; prioritizing them; and proposing ways to deal with them. Categorization helps assign mitigation actions, as typically risks in the same category can be mitigated using a similar approach.

Risk assessment can use a quantitative or qualitative approach or a mix of the two. In a quantitative approach, the risk level is calculated as the product of likelihood and impact. In a qualitative approach, the risk level can be calculated using a risk matrix.

An example of a risk matrix is shown in Fig. 5.11. The risk matrix is a kind of "multiplication table," in which the risk level is defined as the product of certain categories of likelihood and impact.

In the risk matrix in Fig. 5.11, we have defined four categories of likelihood: low, medium, high, and very high, and three categories of impact: low, medium, and high. At the intersection of the defined categories of likelihood and impact, a risk level is defined. In this risk matrix, six risk level categories are defined: very low, low, medium, high, very high, and extreme. For example, a risk with medium likelihood and high impact has a high risk level, because

$$\text{medium likelihood} * \text{high impact} = \text{high risk level}.$$

There are other risk methods in which—as in the *failure mode and effect analysis* (FMEA) method, for example—likelihood and impact are defined on a numerical scale (e.g., from 1 to 5, where 1 is the lowest and 5 is the highest category) and the risk level is calculated by multiplying the two numbers. For example, for a likelihood of 2 and an impact of 4, the risk level is defined as 2 * 4 = 8.

In such cases, however, care must be taken. Multiplication is commutative, so the risk level of a likelihood of 1 and impact of 5 is 5 and is the same as for a risk with a likelihood of 5 and impact of 1, because 1 * 5 = 5 * 1. In practice, however, impact is

RISK LEVEL		RISK LIKELIHOOD			
		low	medium	high	very high
RISK IMPACT	low	very low	low	medium	medium
	medium	low	medium	high	high
	high	medium	high	very high	extreme

Fig. 5.11 Risk matrix

a much more important factor for us. A risk with a likelihood of 5 and an impact of 1 will perhaps occur very often, but will cause practically no trouble, due to its negligible impact. In contrast, a risk with a likelihood of 1 and an impact of 5 admittedly has a very small chance of occurring, but when it occurs, its consequences can be catastrophic.

In addition to the quantitative and qualitative approaches to risk assessment, there is also a mixed approach. In this approach, probability and impact are not defined by specific numerical values, but are represented by ranges of values that express some uncertainty about the estimation of these parameters. For example, likelihood can be defined as the interval [0.2, 0.4], and impact—as the interval [$1000, $3000]. This means that we expect the likelihood of the risk to be somewhere between 20% and 40%, while its impact is expected to be somewhere between $1000 and $3000. The risk level in this approach can also be specified as a range, according to the following multiplication formula:

$$[a, b] * [c, d] = [a * c, b * d],$$

i.e., in our case, the risk level will be

$$[0.2, 0.4] * [\$1000, \$3000] = [\$200, \$1200].$$

The risk level is therefore between $200 and $1200.

The mixed approach is a good solution if we find it difficult to estimate the exact values of the probabilities and impacts of individual risks (quantitative approach), but at the same time, we want more accurate risk assessment results than those expressed on an ordinal scale (qualitative approach).

Example An organization defines a risk level as the product of its likelihood and impact. The following risks are defined, along with estimates of the likelihood and impact of each risk:

Risk 1: likelihood of 20%, impact of $40,000
Risk 2: likelihood of 10%, impact of $100,000
Risk 3: likelihood of 5%, impact of $20,000

To calculate the *total* risk level, we add up the respective products:

Total risk level = 0.2 * $40,000 + 0.1 * $100,000 + 0.05 * $20,000 = $8000 + $10,000 + $1000 = $19,000. This means that if we do not take any measures to minimize (mitigate) these risks, the average (expected) loss we will incur as a result of their potential occurrence will be about $19,000.

Note that quantitative risk analysis for a single risk doesn't make much sense, because risks—from the point of view of the tester or an outside observer—are *random* phenomena. We don't know when or if they will occur at all and, if so, what kind of damage they will really cause. An analysis of a single random phenomenon with likelihood P and impact X makes no sense, because it will occur or not, so the loss will be either 0 or X (assuming our estimation of impact was correct) and will

never be P*X. However, an analysis performed for *multiple* risks, as in the example above, makes much more sense. This is because the sum of the risk levels for all the identified risks can be treated as the expected value of the loss we will incur due to the occurrence of *some* of these risks.

5.2.4 Product Risk Control

Product risk control 🕮 plays a key role in risk management, as it includes all measures that are taken in response to identified and assessed product risks. Risk control is defined as measures taken to mitigate and monitor risks throughout the development cycle. Risk control is a process that includes two main tasks: risk mitigation and risk monitoring.

Risk mitigation
Once the risks have been analyzed, there are several options for **risk mitigation** 🕮 or possible responses to risk [71, 72]:

- Risk acceptance
- Risk transfer
- Contingency plan
- Risk mitigation by testing

Accepting (ignoring) risks can be a good idea when dealing with low-level risks. Such risks are usually prioritized lowest and dealt with at the very end. Often, we may simply run out of time to deal with them, but this does not particularly bother us. This is because even when such risks occur, the damage they bring is minor or even negligible. So there is no point in wasting time dealing with them when we have other, much more important, and serious issues to consider (e.g., risks of a much higher level).

An example of risk transfer could be, for example, taking out an insurance policy. In exchange for paying the insurance premium, one transfers the risk of incurring the cost of the risk to a third party, in this case—to the insurer.

Contingency plans are developed for certain types of risks. These plans are developed in order to avoid chaos and a delay in responding to risks when an undesirable situation occurs. Thanks to the existence of such plans, when a risk occurs, everyone knows exactly what to do. The response to risk is therefore quick and well thought out in advance and—therefore—effective.

In the Foundation Level syllabus, of the above risk response methods, only risk mitigation by testing is presented in detail, as the syllabus is about testing. Tests can be designed to check if a risk is actually present. If the test fails (triggers a failure, so shows a defect), the team is aware of the problem. This accordingly mitigates the risk, which is already identified and not unknown. If the test passes, the team gains more confidence in the quality of the system. Risk mitigation can reduce the likelihood or impact of a risk. In general, testing contributes to lowering the overall

risk level in the system. Each passed test can be interpreted as a situation in which we have shown that a specific risk (with which the test is associated) does not exist because the test is passed. This reduces the total residual risk by the value of the risk level of that particular risk.

Actions that can be taken by testers to mitigate product risk are as follows:

- Selecting testers with the right level of experience, appropriate for the type of risk
- Applying an appropriate level of independence of testing
- Conducting reviews
- Performing static analysis
- Selecting appropriate test design techniques and coverage levels
- Prioritizing tests based on risk level
- Determining the appropriate range of regression testing

Some of the mitigation activities address early preventive testing by applying them before dynamic testing begins (e.g., reviews). In risk-based testing, risk mitigation activities should occur throughout the SDLC.

The agile software development paradigm requires team self-organization (see Sect. 1.4.2 on testing roles and Sect. 1.5.2 on the "whole team" approach) while providing risk mitigation practices that can be viewed as a holistic risk mitigation system. One of its strengths is its ability to identify risks and provide good mitigation practices. Examples of risks reduced by these practices are [69]:

- The risk of customer dissatisfaction. In an agile approach, the customer or customer's representative sees the product on an ongoing basis, which, with good project execution, frequent feedback, and intensive communication, mitigates this risk.
- The risk of not completing all functionality. Frequent release and product planning and prioritization reduce this risk by ensuring that increments related to high business priorities are delivered first.
- The risk of inadequate estimation and planning. Work product updates are tracked daily to ensure management control and frequent opportunities for correction.
- The risk of not resolving problems quickly. A self-organized team, proper management, and daily work reports provide the opportunity to report and resolve problems on a daily basis.
- The risk of not completing the development cycle. In a given cycle (the shorter, the better), the agile approach delivers working software (or, more precisely, a specific piece of it) so that there are no major development problems.
- The risk of taking on too much work and changing expectations. Managing the product backlog, iteration planning, and working releases prevent the risk of unnoticed changes that negatively impact product quality by forcing teams to confront and resolve issues early.

Risk monitoring

Risk monitoring 🕮 is a risk management task that deals with activities related to checking the status of product risks. Risk monitoring allows testers to measure implemented risk mitigation activities (mitigation actions) to ensure they are achieving their intended effects and to identify events or circumstances that create new or increased risks. Risk monitoring is usually done through reports that compare the actual state with what was expected. Ideally, risk monitoring takes place throughout the SDLC.

Typical risk monitoring activities involve reviewing and reporting on product risks. There are several benefits of risk monitoring, for example:

- Knowing the exact and current status of each product risk
- The ability to report progress in reducing residual product risk
- Focusing on the positive results of risk mitigation
- Discovering risks that have not been previously identified
- Capturing new risks as they emerge
- Tracking factors that affect risk management costs

Some risk management approaches have built-in risk monitoring methods. For example, in the aforementioned FMEA method, estimating the likelihood and impact of a risk is done twice—first in the risk assessment phase and then, again, after implementing actions to mitigate that risk.

5.3 Test Monitoring, Test Control, and Test Completion

FL-5.3.1 (K1) Recall metrics used for testing
FL-5.3.2 (K2) Summarize the purposes, content, and audiences for test reports
FL-5.3.3 (K2) Exemplify how to communicate the status of testing

The primary objective of **test monitoring** 🕮 is collecting and sharing of information to gain insight into test activities and visualize the testing process. The monitored information can be collected manually or automatically, using test management tools. We rather suggest an automatic approach, using the test log and information from the defect reporting tool, supported by a manual approach (direct control of the tester's work; this allows a more objective assessment of the current status of testing). However, do not forget to collect information manually, e.g., during daily Scrum meetings (see Sect. 2.1.1).

The results obtained are applied to:

- Measurement of the fulfillment of exit criteria, e.g., achievement of assumed product risk coverage, requirements, and acceptance criteria
- Assessing the progress of the work against the schedule and budget

Activities performed as part of **test control** 🕮 are mainly as follows:

- Making decisions based on information obtained from test monitoring
- Re-prioritizing tests when identified risks materialize (e.g., failure to deliver software on time)
- Making changes to the test execution schedule
- Assessing the availability or unavailability of the test environment or other resources

5.3.1 Metrics Used in Testing

During and after a given test level, metrics can (and should) be collected allowing to estimate:

- The progress of schedule and budget implementation
- The current quality level of the test object
- The appropriateness of the chosen test approach
- The effectiveness of the test activities from the point of view of achieving the objectives

Test monitoring collects various metrics to support test control activities. Typical test metrics include:

- **Project metrics** (e.g., task completion, resource utilization, testing effort, percentage of completion of planned test environment preparation work, milestone dates)
- **Test case metrics** (e.g., test case implementation progress, test environment preparation progress, number of test cases run/not run, passed/failed, test execution time)
- **Product quality metrics** (e.g., availability, response time, mean time to failure)
- **Defect metrics** (e.g., number of defects found/repaired, defect priorities, defect density (number of defects per unit volume, e.g., 1000 lines of code), defect frequency (number of defects per unit time), percentage of defects detected, percentage of successful confirmation tests)
- **Risk metrics** (e.g., residual risk level, risk priority)
- **Coverage metrics** (e.g., requirements coverage, user stories coverage, acceptance criteria coverage, test conditions coverage, code coverage, risks coverage)
- **Cost metrics** (e.g., cost of testing, organizational cost of quality, average cost of test execution, average cost of defect repair)

Note that different metrics serve different purposes. For example, from a purely management point of view, a manager will be interested, for example, in how many planned test executions were made or whether the planned budget was not exceeded. From a software quality point of view, however, more important will be such issues as, for example, the number of defects found by their degree of criticality, mean time between failures, etc. Being able to choose the right set of metrics to measure is an art, as it is usually impossible to measure everything. In addition, the more data in the

reports, the less readable they become. You should choose a small set of metrics to measure, which at the same time will allow you to answer all the questions about the aspects of the project you are interested in. The *Goal-Question-Metric* (GQM) technique, for example, can help develop such a measurement plan, but we do not present it here, as it is beyond the scope of the syllabus.

5.3.2 Purpose, Content, and Audience for Test Reports

Test reports are used to summarize and provide information on test activities (e.g., tests at a given level) both during and after their execution. A test report produced during the execution of a test activity is called a **test progress report** 🕮, and the report produced at the completion of such an activity—**a test completion report** 🕮. Examples of these reports are shown in Figs. 5.12, 5.13, and 5.14.

According to the ISO/IEC/IEEE 29119-3 standard [3], a typical test report should include:

- Summary of the tests performed
- Description of what happened during the testing period
- Information on deviations from the plan
- Information on the status of testing and product quality, including information on meeting the exit criteria (or Definition of Done)
- Information on factors blocking the tests

Summary Status Test Report for: New subscription system (NSS) **Vers.:** Iteration 3

Covers: Complete NSS iteration 3 results.

Progress against Test Plan: Test has been done in the iteration on the 5 user stories for this iteration.

For the one high risk story 92 % statement coverage was achieved, and for the others 68 % statement coverage was achieved on average.

There are no outstanding defects of severity 1 and 2, but the showcase showed that the product has 16 defects of severity 3.

Factors blocking progress: None

Test measures: 6 new test procedures have been developed, and 2 of the other test procedures have been changed.

The testing in the iteration has taken up approx. 30 % of the time. The test took about 2½ hours.

New and changed risks: The risks for the stories have been mitigated satisfactorily. New risks are not identified yet.

Planned testing: As per test plan.

Backlog added: 16 defects (severity 3)

Fig. 5.12 Test progress report in an agile organization (after ISO/IEC/IEEE 29119-3)

- Measures related to defects, test cases, test coverage, work progress, and resource utilization
- Residual risk information
- Information on work products for reuse

A typical test progress report further includes information on:

- Status of test activities and the progress of the test plan
- Tests scheduled for the next reporting period

Test reports must be tailored to both the project context and the needs of the target audience. They must include:

- Detailed information on defect types and related trends—for technical audience
- Summary of defect status by priority, budget, and schedule, as well as passed, failed, and blocking test cases—for business stakeholders

The main recipients of the test progress report are those who are able to make changes to the way tests are conducted. These changes may include adding more

Test report for: New subscription system (NSS) **Vers.:** Iteration 3361

Covers: NSS final iteration result, including result of previous iterations, in preparation for a major customer delivery (for use).

Risks: The live data risk was retired by creation of a simulated database using historic live data "cleaned" by the test team and customer.

Test Results: Customer accepted this release of the product based on:

16 user stories were successful, including one added after the last status report.

100% statement coverage was achieved in technology-facing testing with the one high risk story, and for the others 72% statement coverage was achieved on average.

Team accepted the backlog on 4 defects of severity 3.

Showcase was accepted by the customer with no added findings. Showcase demo iteration features interfaced with "live" data.

Performance of the iteration features was found to be acceptable by team and customer.

New, changed, and residual risks: Security of the system could become an issue in future releases, assuming a follow work activity is received from the customer.

Notes for future work from retrospective:

Iteration team feels a new member could be needed given possible new risk since no one has knowledge in this area.

Severity 3 defects that move on to backlog should be addressed in next release to reduce technical debt.

The modified live data worked well and should be maintained.

Test automation and exploratory testing is working, but additional test design techniques should be considered, e.g. security and combinatorial testing.

Fig. 5.13 Test summary report in an agile organization (after ISO/IEC/IEEE 29119-3)

Project PC-part of the UV/TRT-14 33a product

System Test Completion Report, V 1.0, 29/11/2019
Written by: Carlo Titlefsen (Test Manager)
Approved by: Benedicte Rytter (Project Manager)

Summary of testing performed:
— The test specification was produced; it included 600 test procedures.
— The test environment was established according to the plan.
— Test execution and recording were performed according to the plan.
— Based on the outcomes of testing, it is recommended that UAT commence.

Deviations from planned testing: The Requirement Specification was updated during the test to v5.6. This entailed rewriting of a few test cases, but this did not have an impact on the schedule.
Test completion evaluation: All test procedures were executed and passed testing, except 3. There are no remaining severity 1 or 2 defects, and all open severity 3 and 4 defects have been signed off by the business. The remaining 3 failed test cases will be rerun during UAT, once remaining severity 3 and 4 defects have been fixed. The requirements for these 3 failed tests are low risk.
Factors that blocked progress: Deployment was delayed on 5 occasions, preventing testing from starting on time. This equated to an increase in testing duration by 25 person days.

Test measures:
Three test procedures out of 605 were not passed due to defects. All test procedures that were run had passed at the end of the planned 3 weeks of testing.
During test execution, 83 defects were detected and 80 were closed.
Duration of test design and execution:
— 12,080 working hours were spent on the production of test cases and test procedures
— 10 working hours were spent on the establishment of the test environment
— 15,040 working hours were spent on test execution and defect reporting
— One hour was spent on the production of this report.
Residual risks: All the risks listed in the test plan have been eliminated, except the one with the lowest exposure, Risk # 19. This risk has been deemed acceptable by the Business Representative.
Test deliverables: All deliverables specified in the plan have been delivered to the common CM-system according to the procedure.
Reusable test assets: The test specification and the related test data and test environment requirements can be reused for maintenance testing, if and when this is needed.
Lessons learned: The test executors should have been given a general introduction to the system; it sometimes delayed them finding out how to perform a test case, but fortunately the test analyst was available throughout the test period.

Fig. 5.14 Test summary report in a traditional organization (per ISO/IEC/IEEE 29119)

testing resources or even making changes to the test plan. Typical members of this group are therefore project managers and product owners. It may also be useful to include in this group people that are responsible for factors inhibiting test progress, so that it is clear that they are aware of the problem. Finally, the test team should also

be included in this group, as this will allow them to see that their work is appreciated and help them understand how their work contributes to overall progress.

The audience of the test completion report can be divided into two main groups: those responsible for making decisions (what to do next) and those who will conduct tests in the future using the information from the report. The decision-makers will vary depending on the testing that is reported (e.g., test level, test type, or project) and may decide which tests should be included in the next iteration or whether to deploy a test object given the reported residual risk. The second group are those who are responsible for future testing of the test object (e.g., as part of maintenance testing) or those who can otherwise decide on the reuse of products from reported tests (e.g., cleaned customer test data in a separate project). In addition, all individuals appearing on the original distribution list for the test plan should also receive a copy of the test completion report.

5.3.3 Communicating the Status of Testing

The means of communicating the status of testing varies, depending on the concerns, expectations, and vision of the test management, organizational test strategies, regulatory standards, or, in the case of self-organizing teams (see Sect. 1.5.2), the team itself. Options include, in particular:

- Verbal communication with team members and other stakeholders
- Dashboards such as CI/CD dashboards, task boards, and burndown charts
- Electronic communication channels (e.g., emails, chats)
- Online documentation
- Formal test reports (see Sect. 5.3.2)

One or more of these options can be used. More formal communication may be more appropriate for distributed teams, where direct face-to-face communication is not always possible due to geographic or time differences.

Figure 5.15 shows an example of a typical dashboard for communicating information about the continuous integration and continuous delivery process. From this dashboard, you can very quickly get the most important basic information about the state of the process, for example:

- The number of releases by release status (successful, not built, aborted, etc.)
- The dates and status of the last ten releases
- Information about the last ten commits
- Frequency of releases
- Average time to create a release as a function of time

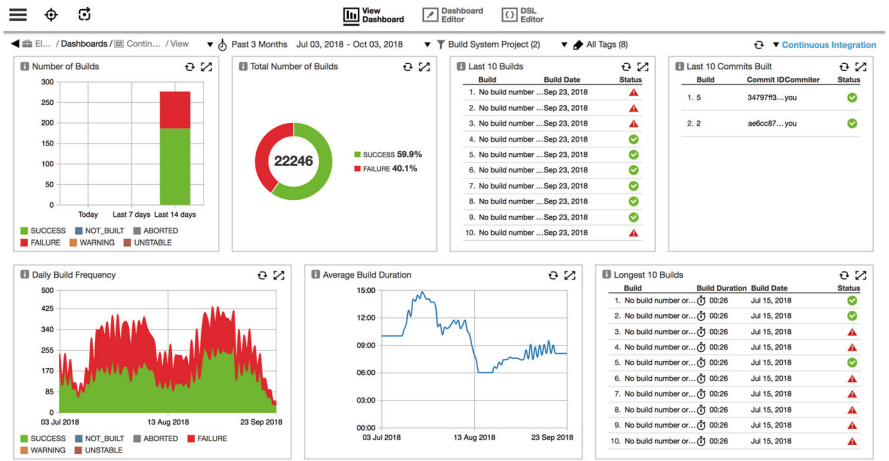

Fig. 5.15 Example of a CI/CD dashboard (source: https://devops.com/electric-cloud-extends-continuous-delivery-platform/)

5.4 Configuration Management

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

The primary goal of configuration management is to ensure and maintain the integrity of the component/system and testware and the interrelationships between them throughout the project and product lifecycle. Configuration management is to ensure that:

- All configuration items, including test objects, test items (individual parts of test objects), and other testware, have been *identified*, version controlled, tracked for changes, and *linked to each other* in a way that maintains traceability at all stages of the test process
- All identified documentation and software items are referenced explicitly in the test documentation

Today, appropriate tools are used for configuration management, but it is important to note that configuration management procedures along with the necessary infrastructure (tools) should be identified and implemented at the planning stage, since the process covers all work products that occur within the development process.

Figure 5.16 shows a graphical representation of a certain repository stored in the git code versioning system. The vertices C0, C1, ..., C6 denote so-called snapshots, i.e., successive versions of the source code. The arrows indicate on the basis of which version the next version was created.

For example, commit C1 was created on the basis of commit C0 (e.g., a developer downloaded the C0 code to their local computer, made some changes to it, and

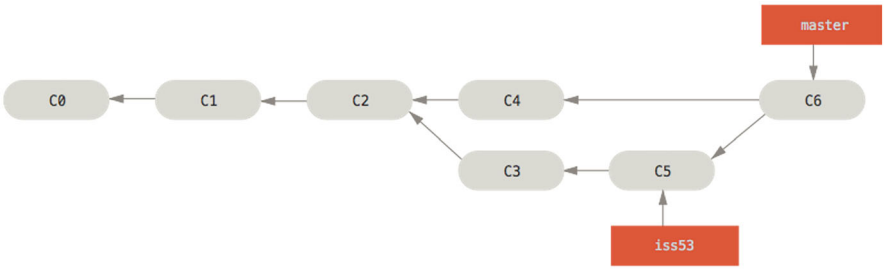

Fig. 5.16 Graphical representation of a code repository in the Git system (source: https://git-scm.com/docs/gittutorial)

committed it to the repository). The repository distinguishes between these two versions of the code. Commits C3 and C4 may have been created independently by two other developers, based on the same C2 commit. The "master" rectangle represents the master branch. Commit C5, on the other hand, represents the "iss53" branch, where the code was created as a result of fixing an error detected in the C3 version of the code. Commit C6 was created as a result of a *merge of* changes made independently in commits C5 (defect repair) and C4 (e.g., adding a new feature based on the C2 version of the code).

By using a version control system like git, developers have full control over their code. They can track, save, and—if necessary—recreate any historical version of the code. If an error is made or a failure occurs in some compilation, it is always possible to undo the changes and return to the previous, working version.

Nowadays, the use of code versioning tools is the de facto standard. Without such tools, the project would be in chaos very quickly.

Example Consider a very simplified example of the application of configuration management in practice. Assume that the system we are producing consists of three components: A, B, and C. The system is used by two of our customers: C1 and C2. The following sequence of events shows the history of changes in the code repository and the history of software releases to customers. We assume that the creation of a given version of the application always takes into account the latest versions of the components included in it.

1. Uploading version 1.01 of component A to the repository
2. Uploading version 1.01 of component B to the repository
3. Uploading version 1.02 of component A (after defect removal) to the repository
4. Uploading version 1.01 of component C to the repository
5. Creating version 1.0 of the application for release and sending it to C1
6. Uploading version 1.02 of component C to the repository (adding new functionality)
7. Creating version 1.01 of the application for release and sending it to C1
8. Uploading version 1.03 of component A to the repository (adding new functionality)
9. Creating version 1.02 of the application for release and sending it to C2

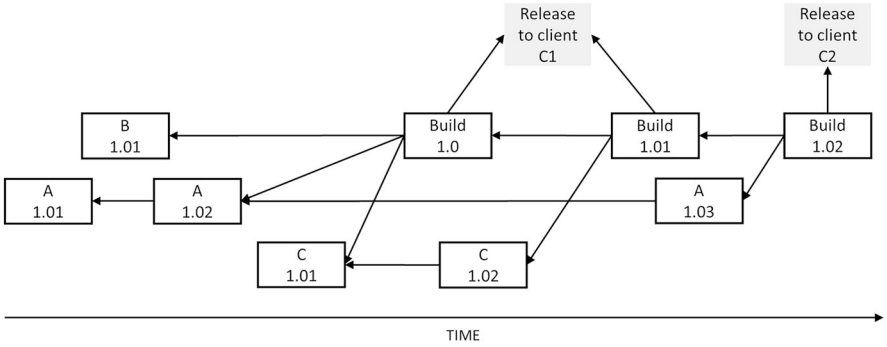

Fig. 5.17 History of the creation of successive versions of components and software releases

This process is graphically depicted in Fig. 5.17.

Now, assume that at this point (after step 9), customer C1 has reported a failure. If we didn't have a configuration management process established, we wouldn't know in which versions of components A, B, and C to look for potential defects. Let us assume that the customer sent us information that the problem was observed in version 1.01 of the software. The configuration management tool, based on this information and the above data, is able to reconstruct the individual software components that went into the software version 1.01 delivered to the customer C1 in step 7. Moreover, the tool is able also to reproduce proper versions of the test cases used for software in version 1.01, proper versions of environment components, databases, etc. that were used to build version 1.01 of the software. In particular, analyzing the above sequence of events, we see that the software version 1.01 consists of:

- Component A version 1.02
- Component B version 1.01
- Component C version 1.02

This means that the potential defect is in one (or more) of these three components. Note that if the defect is found to be in component B, after fixing it and creating version 1.02, a new version of the software (1.03) should be sent not only to customer C1 but also to customer C2, since the current version of the software used by C2 uses the defective component B in version 1.01.

5.5 Defect Management

FL-5.5.1 (K3) Prepare a defect report

One of the goals of testing is to find defects, and therefore, all defects found should be logged. How a defect is reported depends on the testing context of the component or system, the test level, and the chosen SDLC model. Each identified defect should

be investigated and tracked from the moment it is detected and classified until the problem is resolved.

The organization should implement a **defect management** 🕮 process, with its formalization varying from an informal approach to a very formal one. It may be that some reports describe false-positive situations rather than actual failures caused by defects. Testers should try to minimize the number of false-positive results that are reported as defects; nevertheless, in real projects, a certain number of reported defects are false positives.

The primary objectives of the **defect report** 🕮 are:

- Communication:
  - Provide manufacturers with information about the incident to enable them to identify, isolate, and repair the defect if necessary.
  - Provide test management with current information about the quality of the system under test and the progress of testing.
- Enabling the collection of information about the status of the product under test:
  - As a basis for decision-making
  - To identify risks early
  - As a basis for post-project analysis and improvement of development procedures

In the ISO/IEC/IEEE 29119-3 standard [3], defect reports are called *incident reports*. This is a slightly more precise nomenclature, because not every observed anomaly immediately has to mean a problem to be solved: a tester can make a mistake and consider something that is perfectly correct as a defect that needs fixing. Such situations are usually analyzed during the defect lifecycle, and typically such reports are rejected, with a status of "not a defect" or similar.

An example of a defect lifecycle is shown in Fig. 5.18. From the tester's point of view, only the statuses *Rejected, Changed*, *Postponed*, and *Closed* mean that the defect analysis has been completed; the status *Fixed* means that the developer has fixed the defect, but it can be closed only after retesting.

The defect report for dynamic testing should include:

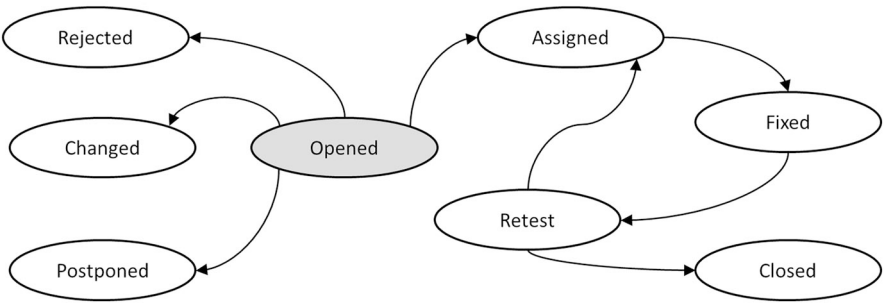

Fig. 5.18 An example of a defect lifecycle and defect statuses

- Unique identifier
- Title and a brief summary of the reported anomaly
- Date of the report (the date the anomaly was discovered)
- Information about the defect report's author
- Identification of the item under test
- Phase of the software development cycle in which an anomaly was observed
- Description of the anomaly to enable its reproduction and removal, including any kind of logs, database dumps, screenshots, or recordings
- Actual and expected results
- Defect removal priority (if known)
- Status (e.g., open, deferred, duplicate, re-opened, etc.)
- Description of the nonconformity to help determine the problem's cause
- Urgency of the solution (if known)
- Identification of a software or system configuration item
- Conclusions and recommendations
- History of changes
- References to other elements, including the test case through which the problem was revealed

An example of a defect report is shown in Fig. 5.19.

Stakeholders interested in defect reports include testers, developers, test managers, and project managers. In many organizations, the customer also gains access to defect management system, especially in the period immediately after the release. However, it should be taken into account that the typical customer is not familiar with the rules of defect reporting. Also, it is typical that from the client point of view, almost every defect is critical, because it impedes the customer's work.

Sample Questions

Question 5.1
(FL-5.1.1, K2)
Which of the following is NOT part of the test plan?

A. Test strategy.
B. Budgetary constraints.
C. Scope of testing.
D. Risk register.

Choose one answer.

Question 5.2
(FL-5.1.2, K2)
Which of the following is done by the tester during release planning?

A. Conducting detailed risk analysis for user stories.
B. Identifying non-functional aspects of the system to be tested.

Incident Registration Form			
Number	278		
Short Title	Information truncated		
Software product	Project PC-part of the UV/TIT-14 33a product		
Version (n.m)	5.2		
Status = Created			
Registration created by	Heather Small	Date & time	14th May
Anomaly observed by	Heather Small	Date & time	14th May
Comprehensive description	*The text in field "Summary" is truncated after 54 characters; it should be able to show 75 characters.*		
Observed during	~~Walk-through / Review / Inspection / Code & Build /~~ Test ~~/ Use~~		
Observed in	~~Requirement / Design / Implementation /~~ Test ~~/ Operation~~		
Symptom	~~Oper. system crash / Program hang-up / Program crash / Input /~~ Output ~~/ Total product failure / System error / Other:~~		
User impact	~~High /~~ Medium ~~/ Low~~		
User urgency	~~Urgent / High / Medium /~~ Low ~~/ None~~		

Fig. 5.19 Example of a defect report according to ISO/IEC/IEEE 29119-3

C. Estimating test effort for new features planned in a given iteration.
D. Defining user stories and their acceptance criteria.

Choose one answer.

Question 5.3
(FL-5.1.3, K2)

Consider the following entry and exit criteria:

i. Availability of testers.
ii. No open critical defects.
iii. 70% statement coverage achieved in component testing.
iv. All smoke tests performed before system testing have passed.

Which of the above are entry criteria and which are exit criteria for testing?

A. (i), (iii), and (iv) are the entry criteria; (ii) is the exit criterion.
B. (ii) and (iii) are the entry criteria; (i) and (iv) are the exit criteria.

C. (i) and (ii) are the entry criteria; (iii) and (iv) are the exit criteria.
D. (i) and (iv) are the entry criteria; (ii) and (iii) are the exit criteria.

Choose one answer.

Question 5.4
(FL-5.1.4, K3)

The team uses the following extrapolation model for the test effort estimation:

$$E(n) = \frac{E(n-1) + E(n-2) + E(n-3)}{3},$$

where $E(n)$ is the effort in the n-th iteration. The effort in the n-th iteration is thus the average of the effort of the last three iterations.

In the first three iterations, the actual effort was 12, 15, and 18 person-days, respectively. The team has just completed the third iteration and wants to use the above model to estimate the test effort needed in the FIFTH iteration. What should be the estimate made by the team?

A. 24 person-days.
B. 16 person-days.
C. 15 person-days.
D. 21 person-days.

Choose one answer.

Question 5.5
(FL-5.1.5, K3)

You want to prioritize test cases for optimal test execution order. You use a prioritization method based on additional coverage. You use feature coverage as a coverage metric. There are seven features in the system, labeled A, B, C, D, E, F, and G. Table 5.4 shows which features are covered by each test case:

Which test case will be executed THIRD in order?

A. TC5.
B. TC2.
C. TC4.
D. TC3.

Choose one answer.

Table 5.4 Feature coverage by test cases

Test case	Features covered
TC1	A, B, C, F
TC2	D
TC3	A, F, G
TC4	E
TC5	D, G