Lucjan Stapp · Adam Roman
Michaël Pilaeten

ISTQB® Certified Tester Foundation Level

A Self-Study Guide Syllabus v4.0

Springer

ISTQB® Certified Tester Foundation Level

Lucjan Stapp • Adam Roman • Michaël Pilaeten

ISTQB® Certified Tester Foundation Level

A Self-Study Guide Syllabus v4.0

Springer

Lucjan Stapp
Warszawa, Poland

Adam Roman
Jagiellonian University
Kraków, Poland

Michaël Pilaeten
Londerzeel, Belgium

ISBN 978-3-031-42766-4 ISBN 978-3-031-42767-1 (eBook)
https://doi.org/10.1007/978-3-031-42767-1

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

Preface

Purpose of This Book

This book is aimed at those preparing for the ISTQB® Certified Tester—Foundation Level exam based on the Foundation Level syllabus (version 4.0) published in 2023. Our goal was to provide candidates with reliable knowledge based on this document. We know from experience that one can find a lot of information about ISTQB® syllabi and exams on the Internet, but much of it is unfortunately of poor quality. It even happens that materials found on the Web contain serious errors. In addition, due to the significant changes that have taken place in the syllabus compared to the previous version (3.1.1) published in 2018, the amount of material available to candidates based on the new syllabus is still small.

This book expands and details many issues that are described in the syllabus itself in a perfunctory or general way. According to the ISTQB® guidelines for syllabus-based training, an exercise must be provided for each learning objective at the K3 level and a practical example must be provided for each objective at the K2 or K3 level.[1] In order to satisfy these requirements, we have prepared exercises and examples for all learning objectives at these levels. In addition, for each learning objective, we present one or more sample exam questions similar to those that the candidate will see on the exam. This makes the book an excellent aid for studying, preparing for the exam, and verifying acquired knowledge.

Book Structure

The book consists of four main parts.

[1] More about the learning objectives and K levels is given below.

Part I: Certificate, Syllabus, and Foundation Level Exam

Part I provides **official information on the content and structure of the syllabus and the** ISTQB® Certified Tester—Foundation Level exam. It also discusses the ISTQB® **certification structure**. This section also explains the basic technical concepts on which the syllabus and exam structure are based. We explain what the learning objectives and K levels are and what are the **rules for building and administering the actual exam**. It is worth familiarizing yourself with these issues, as understanding them will help you prepare much better for the exam.

Part II: Discussion of the Content of the Syllabus

Part II is the main part of the textbook. Here, we discuss in detail all the content and learning objectives of the Foundation Level syllabus. This part consists of six chapters, corresponding to the six chapters of the syllabus. Each learning objective at K2 or K3 level is illustrated with a practical example, and each learning objective at K3 level is illustrated with a practical exercise.

At the beginning of each chapter, definitions of **keywords** applicable to the chapter are given. Each keyword, at the place of its first relevant use in the text, is marked in bold and with a book icon. 🕮

At the end of each chapter, the reader will find sample exam questions covering all the learning objectives included in this chapter. The book contains **70 original sample exam questions** covering all the learning objectives, as well as **14 practical exercises** corresponding to the K3 level learning objectives. These questions and exercises are not part of the official ISTQB® materials but are constructed by the authors using the principles and rules that apply to their creation for the actual exams. Thus, they are additional material for the readers, allowing them to verify their knowledge after reading each chapter and better understand the material presented.

Optional Material
The text in the box denotes optional material. It relates to the content of the syllabus but goes beyond it and is not subject to examination. It is "for those, who are curious."

Sections with titles marked with an asterisk (*) are optional. They cover the material that was mandatory for the exam according to the old version of the syllabus. We decided to leave these chapters in the book because of their importance and practical application. The reader who uses the textbook only to study for the exam can skip these chapters while reading. These optional sections are:

Section 3.2.6—Review Techniques
Section 4.2.5—Use Case Testing

Part III: Answers to Questions and Exercises

In Part III, we provide solutions to all sample exam questions and exercises appearing in Part II of the book. The solutions are not limited to just giving the correct answers but also include their justifications. They will help the reader to better understand how the real exam questions are created and to better prepare for solving them during the real exam.

Part IV: Official Sample Exam and Additional Questions

The last part, Part IV of the textbook, contains the official sample ISTQB® exam for the Foundation Level certification, additional questions covering learning objectives not covered in the exam, and information about the correct answers and justifications for those answers.

The book is therefore structured in such a way that all important and useful information is in one place:

- Exam structure and rules
- The content discussed in the syllabus with its comprehensive discussion and examples
- Definitions of terms, the knowledge of which is mandatory for the exam
- Original sample test questions and exercises, with correct answers and their justification
- Sample ISTQB® exam with correct answers and their justification

We hope that the material presented in this publication will help all those interested in obtaining the ISTQB® Certified Tester—Foundation Level certification.

Warszawa, Poland — Lucjan Stapp
Kraków, Poland — Adam Roman
Londerzeel, Belgium — Michaël Pilaeten

Contents

Part II The Syllabus Content

About the Authors[2]

Lucjan Stapp, PhD, is a retired researcher and teacher of the Warsaw University of Technology, where for many years he gave lectures and seminars on software testing and quality assurance. He is the author of more than 40 publications, including 12 on various problems related to testing and quality assurance. As a tester, on his career path, he moved from junior tester to test team leader in more than a dozen projects. He has played the role of co-organizer and speaker at many testing conferences (including TestWarez—the biggest testing conference in Poland). Stapp is the founding member of the Information Systems Quality Association (www.sjsi.org), currently its vice president. He is also a certified tester (including ISTQB® CTAL-TM, CTAL-TA, Agile Tester, Acceptance Tester).

Adam Roman, PhD, DSc, is a professor of computer science and research and teaching fellow at the Institute of Computer Science and Computer Mathematics at Jagiellonian University, where he has been giving lectures and seminars on software testing and quality assurance for many years. He heads the Software Engineering Department and is the co-founder of the "Software Testing" postgraduate program at Jagiellonian University. His research interests include research on software measurement, defect prediction models, and effective test design techniques. As part of the Polish Committee for Standardization, he collaborated on the international ISO/IEEE 29119 Software Testing Standard. Roman is the author of monographs *Testing and Software Quality: Models, Techniques, Tools, Thinking-Driven Testing*, and *A Study Guide to the ISTQB® Foundation Level 2018 Syllabus: Test Techniques and Sample Mock Exams* as well as many scientific and popular publications in the field of software testing. He has played the role of speaker at many Polish and

[2] All three authors are experts in software testing. They are co-authors of the Foundation Level syllabus version 4.0, as well as other ISTQB® syllabi. They also have practical experience in writing exam questions.

international testing conferences (including EuroSTAR, TestWell, TestingCup, and TestWarez). He holds several certifications, including ASQ Certified Software Quality Engineer, ISTQB® Full Advanced Level, and ISTQB® Expert Level—Improving the Test Process. Roman is a member of the Information Systems Quality Association (www.sjsi.org).

Michaël Pilaeten. Breaking the system, helping to rebuild it, and providing advice and guidance on how to avoid problems. That's Michaël in a nutshell. With almost 20 years of experience in test consultancy in a variety of environments, he has seen the best (and worst) in software development. In his current role as Learning and Development Manager, he is responsible for guiding his consultants, partners, and customers on their personal and professional path toward quality and excellence. He is the chair of the ISTQB Agile workgroup and Product Owner of the ISTQB® CTFL 4.0 syllabus. Furthermore, he is a member of the BNTQB (Belgium and Netherlands Testing Qualifications Board), an accredited training for most ISTQB® and IREB trainings, and an international keynote speaker and workshop facilitator.

List of Abbreviations

AC	Acceptance criteria
API	Application Programming Interface
ASQF	Der Arbeitskreis für Software-Qualität und Fortbildung
ATDD	Acceptance test-driven development
BDD	Behavior-driven development
BPMN	Business Process Model and Notation
BVA	Boundary value analysis
CC	Cyclomatic complexity
CD	Continuous delivery
CFG	Control flow graph
CI	Continuous integration
CMMI	Capability Maturity Model Integration
COTS	Commercial off-the-shelf
CPU	Central processing unit
CRM	Customer relationship management
DDD	Domain-driven design
DDoS	Distributed Denial-of-Service
DevOps	Development and operations
DoD	Definition of Done
DoR	Definition of Ready
DTAP	Development, testing, acceptance, and production
EP	Equivalence partitioning
FDD	Feature-driven development
FMEA	Failure mode and effect analysis
GUI	Graphical user interface
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IaC	Infrastructure as Code
INVEST	Independent, Negotiable, Valuable, Estimable, Small, and Testable
IoT	Internet of Things

IREB	International Requirements Engineering Board
ISEB	Information Systems Examination Board
ISO	International Organization for Standardization
ISTQB	International Software Testing Qualifications Board
KPI	Key performance indicator
LO	Learning objective
LOC	Lines of code
MBT	Model-based testing
MC/DC	Modified condition/decision coverage
MCR	Modern Code Review
MTTF	Mean time to failure
MTTR	Mean time to repair
N/A	Not applicable
PERT	Program Evaluation and Review Technique
OAT	Operational acceptance testing
QA	Quality assurance
QC	Quality control
QM	Quality management
Req	Requirement
SDLC	Software development lifecycle
SMART	Specific, Measurable, Attainable, Realistic, and Time-Bound
SQL	Structured query language
TC	Test case
TDD	Test-driven development
TMap	Test Management Approach
UAT	User acceptance testing
UI	User interface
UML	Unified Modeling Language
UP	Unified Process
US	User story
WBS	Work breakdown structure
WIP	Work in process (or work in progress)
XP	eXtreme Programming
XSS	Cross-site scripting

Part I
Certification, Syllabus, and Foundation Level Exam

Foundation Level Certificate

History of the Foundation Level Certificate

Independent certification of software testers began in 1998 in the United Kingdom. At that time, under the auspices of the British Computer Society's Information Systems Examination Board (ISEB), a special unit for software testing was established—the Software Testing Board (*www.bcs.org.uk/iseb*). In 2002, the German ASQF (*www.asqf.de*) also created its own qualification system for testers. The Foundation Level syllabus was created on the basis of the ISEB and ASQF syllabi, with the information contained in it reorganized, updated, and supplemented. The main focus has been on topics that provide the greatest practical support for testers. Foundation Level syllabus was created in order to:

- Emphasize testing as one of the core professional specialties within software engineering
- Create a standard framework for professional development of testers
- Establish a system to enable recognition of testers' professional qualifications by employers, customers, and other testers and raise the status of testers
- Promote consistent good testing practices across all software engineering disciplines
- Identify testing issues that are relevant and valuable to the IT industry as a whole
- Create opportunities for software vendors to hire certified testers and gain a commercial advantage over their competitors by advertising their adopted tester recruitment policy
- Provide an opportunity for testers and those interested in testing to gain an internationally recognized qualification in the field

Existing basic certifications in the field of software testing (e.g., certifications issued by ISEB, ASQF, or ISTQB® national councils) that were awarded prior to the inception of the international certificate are considered equivalent to this certificate.

L. Stapp et al., *ISTQB® Certified Tester Foundation Level*,
https://doi.org/10.1007/978-3-031-42767-1_1

The basic certificate does not expire and does not need to be renewed. The date of certification is placed on the certificate.

Local conditions in each participating country are the responsibility of ISTQB®'s national councils (or "Member Boards"). The responsibilities of national councils are defined by ISTQB®, while the implementation of these responsibilities is left to individual member organizations. The responsibilities of national councils typically include accreditation of training providers and scheduling of exams.

Career Paths for Testers

The system created by ISTQB® allows defining career paths for testing professionals based on a three-level certification program that includes Foundation Level, Advanced Level, and Expert Level. The entry point is the Foundation Level described in this book. Holding the Foundation Level certification is a prerequisite for earning subsequent certifications. The holders of a Foundation Level certificate can expand their knowledge of testing by obtaining an Advanced Level qualification. At this level, ISTQB® offers a number of educational programs. In terms of the basic path, there are three possible programs:

- **Technical Test Analyst** (technology oriented, non-functional testing, static analysis, white-box test techniques, working with source code)
- **Test Analyst** (customer oriented, business understanding, functional testing, black-box test techniques, and experience-based testing)
- **Test Manager** (test process and test team management oriented)

The advanced level is the starting point for acquiring further knowledge and skills at the expert level. A person who has already gained experience as a test manager, for example, can choose to further develop their career as a tester by obtaining expert-level certifications in the areas of **test management** and **test process improvement**.

In addition to the core track, ISTQB® also offers specialized education programs on topics such as acceptance testing, artificial intelligence testing, automotive testing, gaming machine testing, game testing, mobile application testing, model-based testing, performance testing, security testing, test automation, or usability testing. In terms of agile methodologies, it is Technical Agile Tester or Agile Test Leadership at Scale.

Figure 1 shows the certification scheme offered by ISTQB® as of June 25, 2023. The latest version of the ISTQB® career path review is available at *www.istqb.org*.

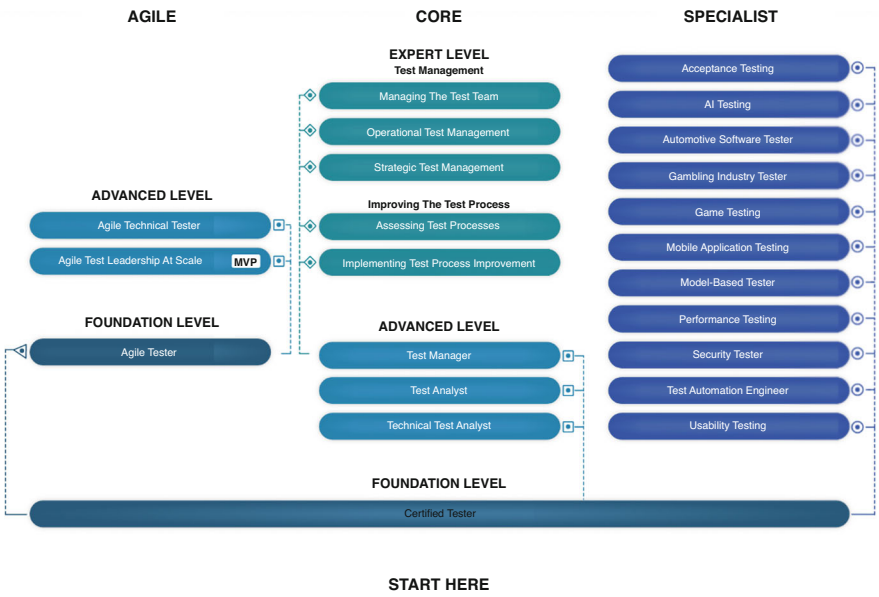

Fig. 1 Official ISTQB® certification scheme (source: www.istqb.org)

Target Audience

The Foundation Level qualification is designed for anyone interested in software testing. This may include testers, test analysts, test engineers, test consultants, test managers, users performing acceptance testing, agile team members, and developers. In addition, the Foundation Level qualification is suitable for those looking to gain basic knowledge in software testing, such as project managers, quality managers, software development managers, business analysts, CIOs, and management consultants.

Objectives of the International Qualification System

- Laying the groundwork for comparing testing knowledge in different countries
- Making it easier for testers to find work in other countries
- Ensuring a common understanding of testing issues in international projects
- Increasing the number of qualified testers worldwide
- Creating an international initiative that will provide greater benefits and a stronger impact than initiatives implemented at the national level
- Developing a common international body of information and knowledge about testing on the basis of syllabuses and a glossary of testing terms and raising the level of knowledge about testing among IT workers

- Promoting the testing profession in more countries
- Enabling testers to obtain widely recognized qualifications in their native language
- Creating conditions for testers from different countries to share knowledge and resources
- Ensuring international recognition of the status of testers and this qualification

Foundation Level Syllabus

Business Outcomes

Associated with each ISTQB® syllabus is a set of so-called business outcomes (business goals). A business outcome is a concise, defined, and observable result or change in business performance, supported by a specific measure. Table 1 lists 14 business outcomes to which a candidate receiving Foundation Level certification should contribute.

Learning Objectives and K-Levels

The content of each syllabus is created to cover the set of *learning objectives* established for that syllabus. The learning objectives support the achievement of business goals and are used to create exams for Foundation Level certificate. Understanding what the learning objectives are and knowing the relation between learning objectives and exam questions are key to effective preparation for the certification exam.

All learning objectives are defined in the syllabus in such a way that each of them constitutes an indivisible whole. Learning objectives are defined at the beginning of each section of the syllabus. Each section of the syllabus deals with exactly one learning objective. This makes it possible to unambiguously link each learning objective (and exam questions) to well-defined portions of the material.

The section below outlines the learning objectives applicable to the Foundation Level syllabus. Knowledge of each topic covered in the syllabus will be tested on the exam according to the assigned learning objective. Each learning objective is assigned a so-called knowledge level, also known as the cognitive level (or K level), K1, K2, or K3, which determines the degree to which a particular piece of material should be assimilated. The knowledge levels for each learning objective are

L. Stapp et al., *ISTQB® Certified Tester Foundation Level*,
https://doi.org/10.1007/978-3-031-42767-1_2

Table 1 Business outcomes pursued by a certified Foundation Level tester

FL-BO1	Understand what testing is and why it is beneficial
FL-BO2	Understand fundamental concepts of software testing
FL-BO3	Identify the test approach and activities to be implemented depending on the context of testing
FL-BO4	Assess and improve the quality of documentation
FL-BO5	Increase the effectiveness and efficiency of testing
FL-BO6	Align the test process with the software development life cycle
FL-BO7	Understand test management principles
FL-BO8	Write and communicate clear and understandable defect reports
FL-BO9	Understand the factors that influence the priorities and efforts related to testing
FL-BO10	Work as part of a cross-functional team
FL-BO11	Know risks and benefits related to test automation
FL-BO12	Identify essential skills required for testing
FL-BO13	Understand the impact of risk on testing
FL-BO14	Effectively report on test progress and quality

presented next to each learning objective listed at the beginning of each chapter of the syllabus.

Level 1: Remember (K1)—the candidate remembers, recognizes, or recalls a term or concept.
Action verbs: Identify, recall, remember, recognize
Examples:

- "Identify typical test objectives."
- "Recall the concepts of the test pyramid."
- "Recognize how a tester adds value to iteration and release planning."

Level 2: Understand (K2)—the candidate can select the reasons or explanations for statements related to the topic and can summarize, compare, classify, and give examples for the testing concept.
Action verbs: Classify, compare, contrast, differentiate, distinguish, exemplify, explain, give examples, interpret, summarize
Examples:

- "Classify the different options for writing acceptance criteria."
- "Compare the different roles in testing" (look for similarities, differences, or both).
- "Distinguish between project risks and product risks" (allows concepts to be differentiated).
- "Exemplify the purpose and content of a test plan."
- "Explain the impact of context on the test process."
- "Summarize the activities of the review process."

Level 3: Apply (K3)—the candidate can carry out a procedure when confronted with a familiar task or select the correct procedure and apply it to a given context.
Action verbs: Apply, implement, prepare, use
Examples:

- "Apply test case prioritization."
- "Prepare a defect report."
- "Use boundary value analysis to derive test cases."

References for the cognitive levels of learning objectives:

Anderson, L. W. and Krathwohl, D. R. (eds) (2001) A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives, Allyn & Bacon

Requirements for Candidates

Candidates taking the ISTQB® Certified Tester Foundation Level exam are only required to have an interest in software testing. However, it is strongly recommended that candidates:

- Had at least basic experience in software development or testing, such as 6 months' experience as a tester performing system testing or acceptance testing or as a developer
- Have completed an ISTQB® training course (accredited by one of the ISTQB® national councils in accordance with the ISTQB® standards)

These are not formal requirements, although fulfilling them makes it significantly easier to prepare for and pass the certification exam. Anyone can take the exam, regardless of interest or experience as a tester.

References to Norms and Standards

The syllabus contains references to IEEE, ISO, etc. standards and norms. The purpose of these references is to provide a conceptual framework or to refer the reader to a source they can use for additional information. It should be noted, however, that only those provisions of the referenced norms or standards to which the specifically discussed sections of the syllabus refer may be the subject of the exam. The content of norms and standards is *not* the subject of the exam, and references to these documents are for informational purposes only.

The current version of the syllabus (v4.0) refers to the following standards:

- **ISO/IEC/IEEE 29119**—*Software Testing Standard*. This standard consists of several parts, the most important of which from the point of view of the syllabus are Part 1 (general concepts) [1], Part 2 (testing processes) [2], Part 3 (test documentation) [3], and Part 4 (test techniques) [4]. Part 3 of this standard replaces the withdrawn IEEE 829 standard.
- **ISO/IEC 25010**—*System and Software Quality Requirements and Evaluation* (aka SQuaRE), *System and software quality models* [5]. This standard describes the software quality model and replaces the withdrawn ISO 9126 standard.
- **ISO/IEC 20246**—*Work Product Reviews* [6]. This standard describes issues related to work product reviews. It replaces the withdrawn IEEE 1028 standard.
- **ISO 31000**—*Risk Management*, *Principles and Guidelines* [7]. This standard describes the risk management process.

Continuous Update

The IT industry is undergoing dynamic changes. In order to take into account the changing situation and ensure that stakeholders have access to useful, up-to-date information, ISTQB®'s working groups have created a list of links to supporting documents and changes in standards, which is available at *www.istqb.org*. The above information is not subject to the Foundation Level syllabus exam.

Release Notes for Foundation Level Syllabus v4.0

The Foundation Level v4.0 syllabus includes best practices and techniques that have stood the test of time, but a number of significant changes have been made from the previous version (v3.1) in 2018 to present the material in a more modern way, make it more relevant to the Foundation Level, and take into account changes that have occurred in software engineering in recent years. In particular:

- Greater emphasis on methods and practices used in agile software development models (whole-team approach, *shift-left* approach, iteration planning and release planning, test pyramid, testing quadrants, "test first" practices like TDD, BDD, or ATDD).
- The section on testing skills, particularly soft skills, has been expanded and deepened.
- The risk management section has been reorganized and better structured.
- A section discussing the DevOps approach has been added.
- A section discussing in detail some test estimation techniques has been added.
- The decision testing technique was replaced by branch testing.
- A section describing detailed review techniques has been removed.

- The use case-based testing technique has been removed (it is discussed in the Advanced Level syllabus "Test Analyst").
- A section discussing sample test strategies has been removed.
- Some tools-related content has been removed, in particular the issue of introducing a tool to an organization or conducting a pilot project.

Content of the Syllabus

Chapter 1. Fundamentals of Testing

- The reader learns the basic principles related to testing, the reasons why testing is required, and what the test objectives are.
- The reader understands the test process, the major test activities, and testware.
- The reader understands the essential skills for testing.

Learning Objectives

1.1. What is testing?

FL-1.1.1 (K1) Identify typical test objectives.
FL-1.1.2 (K2) Differentiate testing from debugging.

1.2. Why is testing necessary?

FL-1.2.1 (K2) Exemplify why testing is necessary.
FL-1.2.2 (K1) Recall the relation between testing and quality assurance.
FL-1.2.3 (K2) Distinguish between root cause, error, defect, and failure.

1.3. Testing principles

FL-1.3.1 (K2) Explain the seven testing principles.

1.4. Test activities, testware, and test roles

FL-1.4.1 (K2) Summarize the different test activities and tasks.
FL-1.4.2 (K2) Explain the impact of context on the test process.
FL-1.4.3 (K2) Differentiate the testware that supports the test activities.
FL-1.4.4 (K2) Explain the value of maintaining traceability.
FL-1.4.5 (K2) Compare the different roles in testing.

1.5. Essential skills and good practices in testing

FL-1.5.1 (K2) Give examples of the generic skills required for testing.
FL-1.5.2 (K1) Recall the advantages of the whole team approach.
FL-1.5.3 (K2) Distinguish the benefits and drawbacks of independence of testing.

Chapter 2. Testing Throughout the Software Development Life Cycle

- The reader learns how testing is incorporated into different development approaches.
- The reader learns the concepts of test-first approaches, as well as DevOps.
- The reader learns about the different test levels, test types, and maintenance testing.

Learning Objectives

2.1. Testing in the context of a software development life cycle

FL-2.1.1 (K2) Explain the impact of the chosen software development life cycle on testing.
FL-2.1.2 (K1) Recall good testing practices that apply to all software development life cycles.
FL-2.1.3 (K1) Recall the examples of test-first approaches to development.
FL-2.1.4 (K2) Summarize how DevOps might have an impact on testing.
FL-2.1.5 (K2) Explain the shift-left approach.
FL-2.1.6 (K2) Explain how retrospectives can be used as a mechanism for process improvement.

2.2 Test levels and test types

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

2.3 Maintenance testing

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

Chapter 3. Static Testing

- The reader learns about the static testing basics, the feedback, and review process.

Learning Objectives

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.

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.

Chapter 4. Test Analysis and Design

- The reader learns how to apply black-box, white-box, and experience-based test techniques to derive test cases from various software work products.
- The reader learns about the collaboration-based test approach.

Learning Objectives

4.1. Test techniques overview

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

4.2. Black-box test techniques

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

4.3. White-box test techniques

FL-4.3.1 (K2) Explain statement testing.
FL-4.3.2 (K2) Explain branch testing.
FL-4.3.3 (K2) Explain the value of white-box testing.

4.4. Experience-based test techniques

FL-4.4.1 (K2) Explain error guessing.
FL-4.4.2 (K2) Explain exploratory testing.
FL-4.4.3 (K2) Explain checklist-based testing.

4.5. Collaboration-based test approaches

FL-4.5.1 (K2) Explain how to write user stories in collaboration with developers and business representatives.
FL-4.5.2 (K2) Classify the different options for writing acceptance criteria.
FL-4.5.3 (K3) Use acceptance test-driven development (ATDD) to derive test cases.

Chapter 5. Managing the Test Activities

- The reader learns how to plan tests in general and how to estimate test effort.
- The reader learns how risks can influence the scope of testing.
- The reader learns how to monitor and control test activities.
- The reader learns how configuration management supports testing.
- The reader learns how to report defects in a clear and understandable way.

Learning Objectives

5.1 Test planning

FL-5.1.1 (K2) Exemplify the purpose and content of a test plan.
FL-5.1.2 (K1) Recognize how a tester adds value to iteration and release planning.
FL-5.1.3 (K2) Compare and contrast entry criteria and exit criteria.
FL-5.1.4 (K3) Use estimation techniques to calculate the required test effort.
FL-5.1.5 (K3) Apply test case prioritization.
FL-5.1.6 (K1) Recall the concepts of the test pyramid.
FL-5.1.7 (K2) Summarize the testing quadrants and their relationships with test levels and test types.

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.

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.

5.4 Configuration management

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

5.5 Defect management

FL-5.5.1 (K3) Prepare a defect report.

Chapter 6. Test Tools

- The reader learns to classify tools and to understand the risks and benefits of test automation.

Learning Objectives

6.1 Tool support for testing

FL-6.1.1 (K2) Explain how different types of test tools support testing.

6.2. Benefits and risks of test automation

FL-6.2.1 (K1) Recall the benefits and risks of test automation.

Foundation Level Exam

Structure of the Exam

The description of the Foundation Level certification exam is defined in a document entitled "Exam Structure Rules," which is available at *www.istqb.org.*

The exam is in the form of a multiple-choice test and consists of **40 questions**. To pass the exam, it is necessary to answer at least **65% of the questions** correctly (i.e., **26 questions**).

Examinations can be taken as part of an accredited training course or independently (e.g., at an examination center or in a public examination). Completion of an accredited course is not a prerequisite for taking the exam, but attending such a course is recommended, as it allows you to better understand the material and significantly increases your chances of passing the exam. If you fail the exam, you can retake it as many times as you like.

Exam Rules

- The Foundation Level exams are based on the Foundation Level syllabus [8].
- Answering an exam question may require using material from more than one section of the syllabus.
- All learning objectives included in the syllabus (with cognitive levels from K1 to K3) are subject to examination.
- All definitions of keywords in the syllabus are subject to examination (at the K1 level). An online dictionary is available at *www.glossary.istqb.org*.
- Each Foundation Level exam consists of a set of multiple-choice questions based on the learning objectives for Foundation Level syllabus. The level of coverage and distribution of questions are based on the learning objectives, their K levels, and their level of importance according to the ISTQB® assessment. For details on

L. Stapp et al., *ISTQB® Certified Tester Foundation Level*,
https://doi.org/10.1007/978-3-031-42767-1_3

Table 1 Distribution of exam questions by K level

K-level	Number of questions	Time per question [in min]	Average time for level K [in min]
K1	8	1	8
K2	24	1	24
K3	8	3	24
Total	40		56

the structure of each exam component, see the "Distribution of Questions" subsection below.

- In general, it is expected that the time to read, analyze, and answer questions at K1 and K2 levels should not exceed 1 min, and at K3 level, it may take 3 min. However, this is only a guideline for average time, and it is likely that some questions will require more and others less time to answer.
- The exam consists of 40 multiple-choice questions. Each correct answer is worth one point (regardless of the K level of the learning objective to which the question applies). The maximum possible score for the exam is 40 points.
- The time allotted for the exam is exactly 60 min. If the candidate's native language is not the language of the exam, the candidate is allowed an additional 25% of the time (in the case of the Foundation Level exam, this means that the exam will last 75 min).
- A minimum of 65% (26 points) is required to pass.
- A general breakdown of questions by K level is shown in Table 1.

Rules and recommendations for writing multiple-choice questions can be found in ISTQB®—Exam Information document [9].

If you add up the expected time to answer the questions according to the rules given above, then—taking into account the distribution of questions by K levels—you will find that it should take about 56 min to answer all the questions. This leaves a 4-min reserve.

Each exam question should test at least one learning objective (LO) from the Foundation Level syllabus. Questions may include terms and concepts that exist in the K1 level sections, as candidates are expected to be familiar with them. In case the questions are related to more than one LO, they should refer (and be assigned) to the learning objective with the highest K level value.

Distribution of Questions

The structure of the Foundation Level exam is shown in Table 2. Each exam requires mandatory questions covering specific learning objectives, as well as a certain number of questions based on the selected learning objectives. If the number of learning objectives is greater than the number of questions for a specific group of

Table 2 Detailed distribution of exam questions by K levels and chapters

The distribution of questions	K-level	Number of questions from the selected group of learning objectives	Scoring of a single question	
Chapter 1				
FL-1.1.1, FL-1.2.2	K1	1	1	A total of 8 questions required for Chap. 1 K1 = 2 K2 = 6 K3 = 0 Number of points for this chapter = 8
FL-1.5.2	K1	1	1	
FL-1.1.2, FL-1.2.1, FL-1.2.3	K2	1	1	
FL-1.3.1	K2	1	1	
FL-1.4.1, FL-1.4.2, FL-1.4.3, FL-1.4.4, FL-1.4.5	K2	3	1	
FL-1.5.1, FL-1.5.3	K2	1	1	
Chapter 2				
FL-2.1.2	K1	1	1	A total of 6 questions required for Chap. 2 K1 = 2 K2 = 4 K3 = 0 Number of points for this chapter = 6
FL-2.1.3	K1	1	1	
FL-2.2.1, FL-2.2.2	K2	1	1	
FL-2.2.3, FL-2.3.1	K2	1	1	
FL-2.1.1, FL-2.1.6	K2	1	1	
FL-2.1.4, FL-2.1.5	K2	1	1	
Chapter 3				
FL-3.1.1, FL-3.2.1, FL-3.2.3, FL-3.2.5	K1	2	1	A total of 4 questions required for Chap. 3 K1 = 2 K2 = 2 K3 = 0 Number of points for this chapter = 4
FL-3.1.2, FL-3.1.3	K2	1	1	
FL-3.2.2, FL-3.2.4	K2	1	1	
Chapter 4				
FL-4.1.1	K1	1	1	A total of 11 questions required for Chap. 4 K1 = 0 K2 = 6 K3 = 5 Number of points for this chapter = 11
FL-4.3.1, FL-4.3.2, FL-4.3.3	K2	2	1	
FL-4.4.1, FL-4.4.2, FL-4.4.3	K2	2	1	
FL-4.5.1, FL-4.5.2	K2	1	1	
FL-4.2.1, FL-4.2.2, FL-4.2.3, FL-4.2.4, FL-4.5.3	K3	5	1	
Chapter 5				
FL-5.1.2, FL-5.1.6, FL-5.2.1, FL-5.3.1	K1	1	1	A total of 9 questions required for Chap. 5 K1 = 1 K2 = 5 K3 = 3 Number of points for this chapter = 9
FL-5.1.1, FL-5.1.3	K2	1	1	
FL-5.1.7	K2	1	1	
FL-5.2.2, FL-5.2.3, FL-5.2.4	K2	1	1	
FL-5.3.2, FL-5.3.3	K2	1	1	

(continued)

Table 2 (continued)

The distribution of questions	K-level	Number of questions from the selected group of learning objectives	Scoring of a single question	
FL-5.4.1	K2	1	1	
FL-5.1.4, FL-5.1.5, FL-5.5.1	K3	3	1	
Chapter 6				
FL-6.1.1	K2	1	1	A total of 2 questions required for Chap. 6 K1 = 1 K2 = 1 K3 = 0 Number of points for this chapter = 2
FL-6.2.1	K1	1	1	
Certified Tester—Foundation Level SUMMARY			40 points, 60 min	A total of 40 questions

learning objectives described in the table, each question must cover a different learning objective.

The analysis of Table 2 shows that the following 17 learning objectives are *certain* to be covered and examined:

- Five K1-level questions (FL-1.5.2, FL-2.1.2, FL-2.1.3, FL-4.1.1, FL-6.2.1)
- Four K2-level questions (FL-1.3.1, FL-5.1.7, FL-5.4.1, FL-6.1.1)
- Eight questions covering all eight learning objectives at the K3 level

Each of the remaining 23 questions will be selected from a group of two or more learning objectives. Since it is not known which of these learning objectives will be covered, the candidate must master all the material on the syllabus (all learning objectives) anyway.

Tips: Before and During the Exam

In order to successfully pass the exam, the first thing you need to do is carefully read the syllabus and glossary of terms, knowledge of which is required at Foundation Level. This is because the exam is based on these two documents. It is advisable to also solve sample test questions and sample exams. On the ISTQB® website (*www.istqb.org*), you can find official sample exam sets in English and on the Member Boards websites in other languages. The list of all ISTQB® member boards and their websites is published on *www.istqb.org/certifications/member-board-list*.

This publication, in addition to a number of sample questions for each chapter of the syllabus, also includes the official ISTQB® sample exam.

During the exam itself, you should:

- Read the questions carefully—sometimes one word changes the whole meaning of the question or is a clue to give the correct answer!
- Pay attention to keywords (e.g., in what software development life cycle model the project is run).
- Try to match the question with the learning objective—then it will be easier to understand the idea of the question and justify the correctness and incorrectness of individual answers.
- Be careful with questions containing negation (e.g., "which of the following is NOT...")—in such questions, three answers will be true statements, and one will be false statement. You need to indicate the answer containing *the false statement*.
- Choose the option that directly answers the question. Some answers may be completely correct sentences, but do not answer the question asked—for example, the question is about the risks of automation, and one of the answers mentions some benefit of automation.
- Guess if you don't know which answer to choose—there are no negative points, so it doesn't pay to leave questions unanswered.
- Remember that answers with strong, categorical phrases are usually incorrect (e.g., "always," "must be," "never," "in any case")—although this rule may not apply in all cases.

Part II
The Syllabus Content

Chapter 1 Fundamentals of Testing

Keywords:

Coverage	the degree to which specified coverage items are exercised by a test suite expressed as a percentage. *Synonyms:* test coverage.
Debugging	the process of finding, analyzing, and removing the causes of failures in a component or system.
Defect	an imperfection or deficiency in a work product where it does not meet its requirements or specifications. *After* ISO 24765. *Synonyms:* bug, fault.
Error	a human action that produces an incorrect result. *After* ISO 24765. *Synonyms:* mistake.
Failure	an event in which a component or system does not perform a required function within specified limits. *After* ISO 24765.
Quality	The degree to which a work product satisfies stated and implied needs of its stakeholders. *After* IREB.
Quality assurance	activities focused on providing confidence that quality requirements will be fulfilled. *Abbreviation:* QA. *After* ISO 24765.
Root cause	a source of a defect such that if it is removed, the occurrence of the defect type is decreased or removed. *References:* CMMI
Test analysis	the activity that identifies test conditions by analyzing the test basis.
Test basis	The body of knowledge used as the basis for test analysis and design. *After* TMap.

L. Stapp et al., *ISTQB® Certified Tester Foundation Level*,
https://doi.org/10.1007/978-3-031-42767-1_4

Test case	a set of preconditions, inputs, actions (where applicable), expected results, and postconditions, developed based on test conditions.
Test completion	the activity that makes testware available for later use, leaves test environments in a satisfactory condition, and communicates the results of testing to relevant stakeholders.
Test condition	a testable aspect of a component or system identified as a basis for testing. *References:* ISO 29119-1. *Synonyms:* test situation, test requirement.
Test control	the activity that develops and applies corrective actions to get a test project on track when it deviates from what was planned.
Test data	data needed for test execution. *Synonyms:* test dataset.
Test design	the activity that derives and specifies test cases from test conditions.
Test execution	the activity that runs a test on a component or system producing actual results.
Test implementation	the activity that prepares the testware needed for test execution based on test analysis and design.
Test monitoring	the activity that checks the status of testing activities, identifies any variances from planned or expected, and reports status to stakeholders.
Test object	the work product to be tested.
Test objective	the purpose for testing. *Synonyms:* test goal.
Test planning	the activity of establishing or updating a test plan.
Test procedure	a sequence of test cases in execution order and any associated actions that may be required to set up the initial preconditions and any wrap-up activities post execution. *References:* ISO 29119-1.
Test result	the consequence/outcome of the execution of a test. *Synonyms:* outcome, test outcome, result.
Testing	the process within the software development life cycle that evaluates the quality of a component or system and related work products.
Testware	work products produced during the test process for use in planning, designing, executing, evaluating, and reporting on testing. *After* ISO 29119-1.
Validation	Confirmation by examination that a work product matches a stakeholder's needs. *After* IREB.
Verification	Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled. *References:* ISO 9000.

1.1 What Is Testing?

FL-1.1.1 (K1) Identify typical test objectives.
FL-1.1.2 (K2) Differentiate testing from debugging.

Nowadays, there is probably no area of life where software is not used to a greater or lesser extent. Information systems are playing an increasingly important role in our lives, from business solutions (banking sector, insurance) to consumer devices (cars), entertainment (computer games), or communications. Using software that contains defects can:

- Cause a loss of money or time
- Cause a loss of customer confidence
- Make it difficult to gain new customers
- Eliminate from the market
- In extreme situations—cause a threat to health or life

Testing 🕮 of software enables the assessment of software quality and contributes to reducing the risk of software failure in action. Therefore, good testing is essential for project success. Software testing is a set of activities carried out to facilitate the detection of defects and evaluate the properties of software artifacts. Each of these testing artifacts is known as the **test object**. 🕮

Many people, including those working in the IT industry, mistakenly think of testing as just executing tests, that is, running software to find defects. However, executing tests is only part of testing. There are other activities involved in testing. These activities occur both before (items 1–5 below) and after (item 7 below) test execution. These are:

1. Test planning
2. Test monitoring and test control
3. Test analysis
4. Test design
5. Test implementation
6. Test execution
7. Test completion

Testing activities are organized and carried out differently in different software development life cycle (*SDLC*) models (see Chap. 2). Moreover, testing is often seen as an activity focused solely on **verification** 🕮 of requirements, user stories, or other forms of specification (i.e., checking that the system meets the specified requirements). But testing also includes **validation** 🕮—that is, verifying that the system meets user requirements and other stakeholder needs in its operational environment.

Testing may require running the component or system under test—then we have what is called dynamic testing. You can also perform testing without running the

object under test—such testing is called static testing. Testing thus also includes reviews of work products such as:

- Requirements
- User stories
- Source code

Static testing is described in more detail in Chap. 3. Dynamic testing uses different types of test techniques (e.g., black-box, white-box, and experience-based) to derive test cases and is described in detail in Chap. 4.

Testing is not just a technical activity. The testing process must also be properly planned, managed, estimated, monitored, and controlled (see Chap. 5). Testers extensively use various types of tools in their daily work (see Chap. 6), but it is important to remember that testing is largely an *intellectual*, *sapient* activity, requiring testers to have specialized knowledge, analytical skills, critical thinking, and systems thinking [10, 11].

ISO/IEC/IEEE 29119-1 [1] provides additional information on the concept of software testing.

It is worth remembering that testing is a technical study to obtain information about the quality of the test object:

- *Technical*—because we use an engineering approach, using experiments, experience, formal techniques, mathematics, logic, tools (supporting programs), measurements, etc.
- *Study*—because it is a continuous organized search for information

1.1.1 Test Objectives

Testing enables the detection of failures or defects in the work product under test. This fundamental property of testing enables to achieve a number of objectives. The primary **test objectives** 🕮 are:

- Evaluating work products such as requirements, user stories, designs, and code
- Triggering failures and finding defects
- Ensuring required coverage of a test object
- Reducing the level of risk of inadequate software quality
- Verifying whether specified requirements have been fulfilled
- Verifying that a test object complies with contractual, legal, and regulatory requirements
- Providing information to stakeholders to allow them to make informed decisions
- Building confidence in the quality of the test object
- Validating whether the test object is complete and works as expected by the stakeholders

Different goals require different testing strategies. For example, in the case of component testing—i.e., testing individual pieces of an application/system (see Sect. 2.2)—the goal may be to trigger as many failures as possible so that the defects causing them can be identified and fixed early. One may also aim to increase code coverage through component testing. In acceptance testing (see Sect. 2.2), on the other hand, the goals may be:

- To confirm that the system works as expected and meets its (user) requirements
- To provide stakeholders with information on the risks involved in releasing the system at a given time

In acceptance testing [especially user acceptance testing (UAT)], we do not expect to detect a large number of failures or defects, as this may lead to a loss of confidence by future users (see Sect. 2.2.1.4). These failures or defects should be detected at earlier stages of testing.

1.1.2 Testing and Debugging

Some people think that testing is about debugging. However, it is important to remember that testing and debugging are two different activities. Testing (especially dynamic testing) is supposed to reveal **failures** 🕮 caused by defects. **Debugging** 🕮, on the other hand, is a programming activity performed to identify the cause of a **defect** 🕮 (fault), correcting the code and verifying that the defect has been correctly fixed.

When dynamic tests detect a failure, a typical debugging process will consist of the following steps:

- Failure reproduction (in order to make sure that the failure actually occurs and so that it can be triggered in a controlled manner in the subsequent debugging process)
- Diagnosis (finding the cause of the failure, such as locating the defect responsible for the occurrence of that failure)
- Fixing the cause (eliminating the cause of failure, such as fixing a defect in the code)

The subsequent confirmation testing (re-testing) performed by the tester is to ensure that the fix actually fixed the failure. Most often, confirmation testing is performed by the same person who performed the original test that revealed the problem. Regression testing can also be performed after the fix to verify that the fix in the code did not cause the software to malfunction elsewhere. Confirmation testing and regression testing are discussed in detail in Sect. 2.2.3.

When static testing discovers a defect, the debugging process is simply to eliminate the defect. It is not necessary, as in the case of failure discovery in dynamic testing, to perform failure reproduction and diagnosis, because in static testing, the work product under test is not run. The related source code might not even have been

Fig. 1.1 Software failure

created. This is because static testing does not find failures but directly identifies defects. Static testing is discussed in detail in Chap. 3.

Example Consider a simplified version of the problem described by Myers [10]. We are testing a program that receives three natural numbers, *a*, *b*, *c*, as the input. The program answers "yes" or "no," depending on whether a triangle can be constructed from segments with sides of lengths *a*, *b*, *c*. The program is in the form of an executable file `triangle.exe` and takes input values from the keyboard.

The tester prepared several test cases. In particular, the tester ran the program for the input data a = b = c = 16,500 (a test case involving the entry of very large, but valid input values) and received the result presented in Fig. 1.1.

The program answered "no," i.e., it stated that it is impossible to build a triangle from three segments of length 16,500 each. This is a failure, because the expected result is "yes"—it is possible to build an equilateral triangle from such segments. The tester reported this defect to the developer. The developer repeated the test case and got the same result. The developer began to analyze the code, which looks as follows:

```
int main(int argc, _TCHAR* argv[])
{
    short a,b,c,d;
    scanf("%d",&a);
    scanf("%d",&b);
    scanf("%d",&c);
    d=a+b;
    if (abs(a-b)<c && c<d)
        printf ("yes");
    else
        printf("no");
    return 0;
}
```

The developer stated that the condition in the *if* statement is defined correctly, so the defect must be elsewhere. The developer noted that where the sum of variables a and b is assigned to variable d, there may be a problem with register overflow. This is because the variables a, b, d are of the same type (short), so they range from −32,768 to 32,767. However, the sum of 16,500+16,500 is 33,000, more than the maximum

value of the short variable. The operation d = a+b "rolls over the counter" of the variable d, as a result of which, it takes a negative value. At this point, the condition in the *if* statement is false, because it is not true that c < d.

The developer finds that the easiest way to fix the code is to eliminate the variable d, so the counting of the sum of a+b will be done "on the fly." The corrected code looks as follows:

```
int main(int argc, _TCHAR* argv[])
{
    short a,b,c;
    scanf("%d",&a);
    scanf("%d",&b);
    scanf("%d",&c);
    if (abs(a-b)<c && c<a+b)
        printf ("yes");
    else
        printf("no");
    return 0;
}
```

The developer checks the test again for a = b = c = 16,500, and now, the program works correctly. The solution works, because when the compiler has to calculate "on the fly" the sum of a+b, it automatically allocates as much memory as needed for this operation. The defect in the previous version of the code was that the sum was written into the variable d, which had a predefined type and therefore had a limit on the upper value. The developer informs the tester that the defect has been fixed, and the tester re-runs the test and finds that this time, the program returns the correct answer. The tester closes the defect report, recognizing that the defect has been fixed correctly.

In the above example, all of the tester's activities fell under testing, while the developer's activities (other than test execution) fell under debugging.

1.2 Why Is Testing Necessary?

FL-1.2.1 (K2) Exemplify why testing is necessary.
FL-1.2.2 (K1) Recall the relation between testing and quality assurance.
FL-1.2.3 (K2) Distinguish between root cause, error, defect, and failure.

Testing of components, systems, and related documentation supports the identification of software defects. Testing also detects gaps and other deficiencies in software specifications. Hence, testing can help reduce the risk of failure during operation.

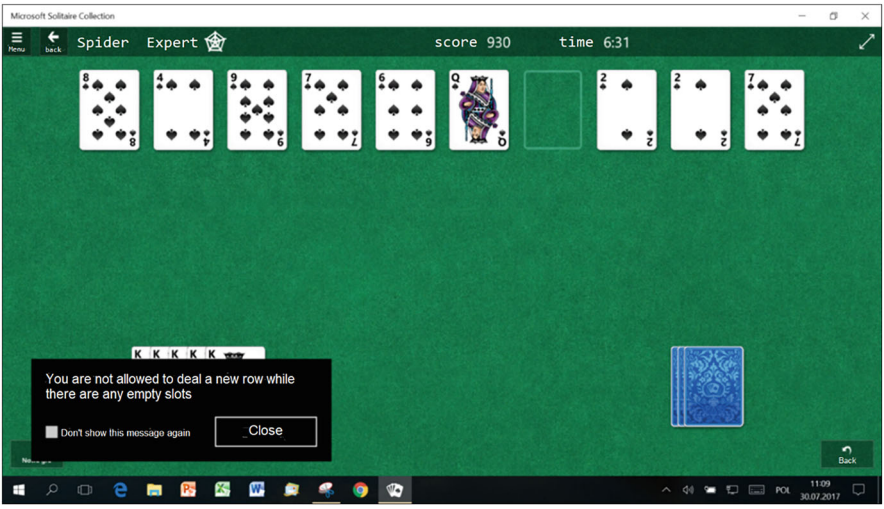

Fig. 1.2 Solitaire game "Spider"

When detected defects are fixed, this contributes to improving the quality of the test object. In addition, software testing may also be required to meet contractual or legal requirements or to meet regulatory standards.

Most of us have encountered software that didn't work as expected—even outside of work. Below we cite three examples—although some occurred quite a long time ago, they are still relevant, as the types of failures we describe also occur in software being produced today.

Case study 1: "Spider" solitaire game
The player plays with 104 cards, 54 of which are spread out in 10 stacks, and 50 lie on the right side of the table in rows of 10 cards. The goal of the game is to stack the cards from king to ace in descending order. When you get to the situation as in Fig. 1.2—there are 9 cards on the table and 30 in groups (making a total of 3*13 = 39 cards—3 full colors); a message appears: *You are not allowed to deal a new row while there are any empty slots*.

When there is an anomaly in an application, two questions must always be answered:

- What is the probability of its occurrence?
- What is the impact of this problem?

In this case, the impact is practically zero: the player can either give up further play or make a work-around—the *Undo* button puts the last stack back on the table and can be spread over the free columns. The probability of such a situation—when all four colors are played, not just spades, as in Fig. 1.2—is on the order of 10^{-6}. This means that a failure will occur about once in a million opportunities for it to

Fig. 1.3 Patriot missile battery

occur. When the cost of failure is close to zero and the probability of occurrence is minimal, defects—especially in non-critical systems—are often not corrected.

Case study 2: The Ariane 5 rocket
On June 4, 1996, the Ariane 5 rocket, prepared by the European Space Agency, exploded 40 seconds after takeoff in Kourou, French Guiana. It was the rocket's first flight, after a decade of preparation that cost $7 billion. The rocket and its payload were worth $500 million.

After a 2-week investigation, it turned out that the problem was in the software, and it was very similar to the *Triangle* problem discussed above. As part of the rocket's upgrade, the 16-bit processor in Ariane 4 was replaced with a 64-bit processor. When the rocket's vertical velocity exceeded 32,767 ($2^{15} - 1$) units, the value of the corresponding variable was read as a negative number, due to register overflow. The control system determined that the rocket had stopped ascending and falls down, so the emergency procedure was activated, which led to the rocket explosion as a result of the self-destruction mechanism.

Conclusion: as part of the system upgrade, appropriate regression tests were not conducted.

Case study 3: The Patriot missile system
Each Patriot system missile battery consists of a radar, a command post (computer), and mobile launchers (see Fig. 1.3). Each battery is assigned an area to protect; in other words, if a targeted missile aims at the area, the battery fires; if the missile's target is outside the area, the battery does not fire.

On February 25, 1991, in Dhahran, Saudi Arabia, during the first Gulf War, a US Patriot missile failed to intercept an Iraqi Scud that hit a US Army barracks, killing 28 and injuring more than 100 soldiers. A General Accounting Office report,

GAO/IMTEC-92-26, titled *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*, states the cause of the incident.

This was an arithmetic error. The system measured time in tenths of a second, using a 24-bit fixed-point register. The fraction 1/10 in the decimal system has a finite representation (0.1), but in the binary system, it is a fraction with infinite expansion (0.000110011001100...). Since the computer represents numbers in finite registers, some part of the fraction must be truncated, which will cause a minimal rounding error. After about 100 h of operation, this rounding error accumulated to 0.34 s, which, at Scud's speed of more than 1676 m/s, gave a distance of more than 0.5 km. The Patriot tracking system therefore considered the Iraqi missile out of its range and did not fire the missile to destroy the enemy Scud.

The report also stated that the defect was known—its elimination required rebooting the system approximately every 4 h with a reboot time of about 5 min; this was stated in the system manual. This is still a typical situation today—users do not read the manuals (if they exist at all). What is more, the defect was known and was fixed in most, but not all, places in the system. This means that developers copied the same code in many places. This still happens today—a defect that has already been fixed must then be reanalyzed and removed.

Today, the likelihood of such a catastrophic situation is less likely, but we can still encounter software malfunctioning.

1.2.1 Testing's Contribution to Success

Testing helps achieve the agreed objectives within the agreed scope, time, quality, and budget. The contribution of testing to project success can be considered in terms of:

- Product quality
- Process quality
- Project goals
- People skills

Product Quality

Testing provides a cost-effective means of detecting defects. Rigorous testing of systems and documentation can reduce the risk of problems (failure) in the production environment and contribute to achieving high product **quality** 🕮. The detection and subsequent removal of defects contribute to the quality of components or systems. An appropriate level of testing may also be required by contractual provisions, legal requirements, or industry standards.

Through testing, users are given an indirect voice in the development project, allowing their needs to be taken into account throughout the entire development process. There are many well-known examples of software and systems (see above) that have been released but due to defects have failed or otherwise failed to meet stakeholder needs. However, with the right test approaches—expertly applied at the

right test levels and in the right phases of the software development cycle—the incidence of such problems can be reduced.

Verification and validation of software by testers before the release enables the detection of failures and facilitates the removal of related defects (debugging). This facilitates testing, reduces risk, and increases the likelihood that the software will meet the needs of stakeholders and fulfill the requirements. Thus, the probability of project success increases.

Process Quality
Testing can indirectly contribute to the quality of the manufacturing process. This is very important because it is known that the final quality of a product is very much influenced by the quality of the process by which the product is developed. The quality of the process can be increased in various ways. For example, introducing test automation improves the efficiency of the system release process. The use of risk-based testing optimizes testing expenditures. The data collected through test monitoring (see Sect. 5.3) helps identify places in the development process that need improvement (e.g., too long time to repair defects, too many defects introduced in a certain phase of the development cycle, etc.).

Project Goals
Testing can help increase the likelihood of achieving project goals. For example, using static testing early in the project reduces software maintenance costs and improves developer efficiency by reducing the time spent on defect fixes. As a result, there is a greater chance that the product will be completed on time and within budget.

Testing provides a means of directly evaluating the quality of a test object at various stages in the SDLC. These measures are used as part of a larger project management activity, contributing to decisions to move to the next stage of the SDLC, such as the release decision.

People Skills
A "side effect" of testing practices is to increase the skills of team members and other stakeholders. For example, performing code reviews (e.g., in the form of walk-throughs; see Sect. 3.2.4) increases understanding of the code and allows less experienced developers to improve their programming and design skills). Close cooperation between testers and system architects during the design phase of the work enables both parties to better understand the project.

1.2.2 Testing and Quality Assurance (QA)

Testing is often mistakenly equated with *quality assurance* (QA). These are two separate (though related) processes that are included in the broader term "Quality *Management"* (QM). QM encompasses all activities aimed at guiding and overseeing an organization's quality activities.

The two basic elements of quality management are:

- Quality assurance
- Quality control

Quality Assurance
Quality assurance 🕮 focuses on establishing, implementing, monitoring, improving, and adhering to quality-related processes. It works on the basis that if a good process is followed correctly, then it will generate a good product. When the relevant processes are implemented correctly, it contributes to defect prevention and increases confidence that appropriate levels of work product quality will be achieved. QA, when applied to software development and maintenance, should also be applied to software testing, which is part of each of these activities. In addition, the use of root cause analysis to detect causes of defects and the application of lessons learned from retrospective meetings to improve processes are also important for effective quality assurance.

Quality Control
Quality control (QC) encompasses a range of activities, including testing activities, that support the achievement of appropriate quality levels. Testing activities are an important part of the overall software development or maintenance process. The proper conduct of QC, especially testing activities, is important for quality assurance, and quality assurance supports proper testing. This is because the test results are used by both quality assurance and quality control. In quality control, they are used to fix defects, while in quality assurance, they provide feedback on how well the development and test processes are performing.

Table 1.1 summarizes the key differences between quality assurance and quality control processes.

1.2.3 Errors, Defects, Failures, and Root Causes

The ISTQB® approach distinguishes between three stages leading to an abnormal result, related to three very important concepts, which are:

- **Error** (also known as an *mistake*)—a human action that causes an incorrect result
- **Defect** (*bug*, *fault*)—an imperfection or defect in a work product that involves failure to meet requirements
- **Failure**—an event in which a component or system fails to perform a required function within a specified context

As a result of human **error** 🕮 in software code or other related work product, a defect can be introduced to the work product. Note that this terminology is important to pay attention to during the exam, as it is counter-intuitive and falls under the vocabulary that the exam is required to know. Usually, in common speech, a defect is called an error—we often say, “There is an error at this place in the code.” From

Table 1.1 Comparison of quality assurance and quality control processes

Category	Quality assurance	Quality control
General description	Implementing processes, methodologies, and standards to ensure that the developed product meets the required quality standards	Performing activities to verify that the developed product meets the required quality standards
Target	Improving the manufacturing process	Product improvement through failure and defect detection
Type of process	Preventive (defect prevention), proactive	Control (defect detection), reactive
Examples of activities	Implementation of processes, e.g., defect management, change management, software release; quality audits; process and product measurements; verification of correct implementation and execution of processes; training of team members; selection of tools	Static analysis of project documentation; code reviews; analysis, design, implementation of test cases; dynamic testing; writing and executing test scripts; defect reporting; using tools to support testing

the point of view of ISTQB® terminology, this is incorrect. There may be *a defect* in the program. An *error* always refers to *human mistake*. Running a piece of code where there is a defect may or may not cause a failure.

Example Let's consider a simple example showing why executing a program line containing a defect does not necessarily lead to a failure. Suppose a piece of code calculates the average value of measurements by dividing their sum by the number of measurements taken. In the code, a following statement is used to do this:

```
AverageValue := SumOfValues / NumberOfMeasurements
```

In this line, there is a defect: the program does not control whether the denominator of the fraction being counted is zero. This is because dividing by zero is not allowed and performing such an operation may result in termination of the program. In a situation where the number of measurements is positive, the execution of the above statement will not cause any problems—the program will work perfectly and will return the correct result. The test that forces the execution of this statement with a positive number of measurements will pass, and we will not notice any sign of failure. However, if this line is executed when there are zero measurements (the value of the `NumberOfMeasurements` variable is 0), failure will occur, and we will notice it.

Example Now let's consider a slightly more sophisticated, though still relatively simple, situation. The program *Count* shown below counts how many entries of an array named T are zeros. We refer to the elements of the array T by its indices—for example, T[3] denotes the element that occurs in the array at position number 3. Let us further assume that the elements of the array (as in many real programming languages) are indexed from zero, so the first element of the array is T[0], the next is

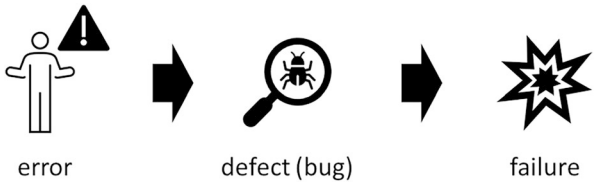

Fig. 1.4 Error, defect, failure

T[1], and so on. The following is the pseudo-code of our program, which contains a defect—the loop passing over the next elements of the array starts passing from element T[1] instead of T[0]:

```
program Count
input: an array T (with elements T[0], T[1], ..., T[n])
output: number of cells in T containing zero
number := 0
for each i = 1, 2, ..., n do
    if T[i] == 0 then number := number + 1
return number
```

Note that the line with the defect (the "for each" loop) will execute for each run of the code. However, the occurrence of a failure (bad result) will depend on whether the cell T[0] contains zero or another number. In the first case (T[0]=0), the result will be incorrect—the program will count one cell too few. For example, if T = (0, 3, 2, 0, 1), the program will return 1 instead of 2. However, in the second case (T[0]≠0), the result will be correct. For example, if T = (5, 2, 0, 1, 0, 0), the program will return 3, which is the correct result despite running a line with a defect. Among other things, test design is about taking these types of situations into account and making sure that the test suite is able to detect similar defects in the code.

An error resulting in the introduction of a defect in one work product can cause an error resulting in the introduction of a defect in another, related work product. The execution of code containing a defect can cause failure, but, as we saw in the example above, this does not necessarily happen with every defect. Some defects cause failure, for example, only after strict input or due to the occurrence of certain preconditions, which may occur very rarely or never. The consecutive occurrence of three factors (error, defect, failure) causes the observed malfunction of the product under testing (Fig. 1.4).

Errors can occur for many reasons:

- Time pressure
- Human fallibility
- Lack of experience or insufficient skills of project team members
- Problems with information exchange among stakeholders
- Ambiguities regarding understanding of requirements and project documentation
- Complexity of the code, design, architecture, problem being solved, and/or technology being used

- Misunderstandings about interfaces within and between systems, especially when there are a large number of them
- Using new, unfamiliar technologies

Failures in turn can be caused *not necessarily* by human error but also by environmental factors, such as:

- Radiation
- Electromagnetic field
- Contamination

These factors can cause failures in the embedded software or affect the software's performance by changing hardware operating conditions.

First "bug" ever
The first bug in history was found in 1947. At Harvard University in Cambridge, Massachusetts, researchers discovered that their computer, Mark II, was consistently experiencing failures. Upon investigating the computer's hardware, they made an unexpected discovery—a moth had become trapped inside. The insect had caused disruptions to the computer's electronics. Figure 1.5 shows an excerpt from the original system log, including the actual "bug" (moth)—the cause of the failures.

Not all unexpected **test results** 🕮 mean failures. A *false-positive result* can be the result of errors related to test execution, defects in test data, test environment, other testware, etc. False-positive results are reported as defects that are not actually there. Similar problems can cause the opposite situation—a *false-negative result*, that is, a situation in which tests fail to detect a defect that they should detect (see Table 1.2).

Formally, a false-positive result[1] is a positive test result (a failed test), when in fact the test should have been passed. An example of such a situation is when the tester misunderstands the specification and incorrectly defines the expected result. A false-negative result is a negative test result (a passed test), when in fact the test should have been failed. An example of such a situation is changing requirements between test cycles. In the second cycle, the changed requirement should cause the test to fail, but in the test, the expected result remained unchanged, and the test is still passed.

Test cases should be designed to avoid defect masking, that is, situations in which the occurrence of one defect prevents the detection of another defect or the occurrence of two defects cancels their mutual effect. There are several best practices used

[1] The terminology "false positive" and "false negative" comes from the field of medical analytics. A negative result means the absence of a virus in the body, for example, and a positive result means its presence. So the analogy with testing is as follows: a test is positive when it detects a failure (the presence of a defect) and negative when it does not detect it.

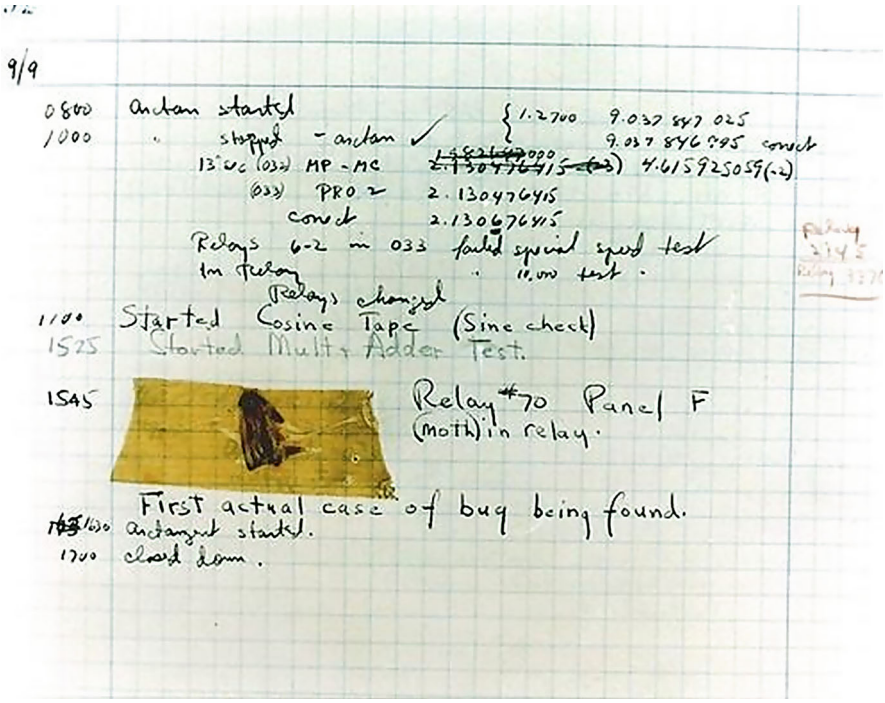

Fig. 1.5 First "bug" ever (www.atlasobscura.com/places/grace-hoppers-bug)

Table 1.2 Possible test results in the context of their correctness

Possible test results		INTERPRETED RESULT	
		TEST PASSED	TEST FAILED
REAL TEST RESULT	TEST PASSED	Correct (negative) result	False positive result
	TEST FAILED	False negative result	Correct (positive) result

in test design that help the tester avoid such situations (more on this in Chap. 4), but in general, masked defects are difficult to detect.

In connection with the above considerations, an important factor is the analysis of the **root cause** 🕮 of the defect: the primary reason that resulted in the defect. This reason can be the occurrence of a specific situation or the occurrence of human error. Analyzing a defect to identify the root cause helps reduce the occurrence of similar defects in the future. Root cause analysis, focusing on the most important root causes of defects, can lead to process improvements, which can in turn translate into further defect reductions in the future.

Example A defect in the code causes incorrect calculation of the discount on bulk purchases in the e-store, resulting in customer complaints. The defective code was written based on a user story, but the product owner misunderstood the discount calculation rules and wrote the story wrong.

In this example:

- Customer complaints are the *consequences*.
- Incorrect calculation of discounts is a *failure*.
- Incorrect discount calculation formula implemented in the code is the *defect*.
- Product owner's lack of knowledge is the *root cause*.
- The root cause is a result of the product owner's *error*.

1.3 Testing Principles

FL-1.3.1 (K2) Explain the seven testing principles.

There are many different "laws" or "principles" related to testing that hold true regardless of the project context or the type of product being developed, and those have emerged over the past 60 years. The Foundation Level syllabus describes seven such principles. These basic principles of testing are:

1. Testing shows the presence, not the absence of defects.
2. Exhaustive testing is impossible.
3. Early testing saves time and money.
4. Defects cluster together.
5. Tests wear out.
6. Testing is context dependent.
7. Absence-of-defects fallacy.

1. Testing Shows the Presence, Not the Absence of Defects

This famous principle was formulated by Edsger Dijkstra, a Danish computer scientist, at the "Software Engineering Techniques" conference in Rome in 1969 [12]. While testing can show that defects exist, we cannot prove that there are no defects in the program under testing. Thus, testing only reduces the likelihood that unidentified defects will remain in the software. The fact that defects are not detected is not a proof of the correctness of the system under test. This principle has serious implications: testing is *negative* in nature, i.e., it shows that something does not work, not that everything is fine. This carries with it some important psychological implications, which will be discussed later.

It is interesting to note that Dijkstra's statement can be formulated as a strict and precise mathematical theorem—this is related to the fact that certain computer problems, such as the so-called halting problem (which can be treated as an

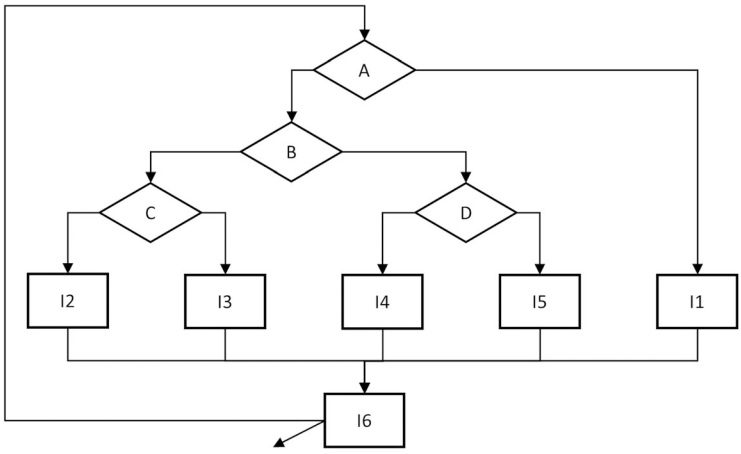

Fig. 1.6 Control flow graph of the code fragment to be tested

algorithm defect), are unsolvable. It is impossible, in general, to answer the question whether a given program will eventually stop for every possible input. So we are not able to detect this type of defect (the possibility of looping of the program) in every case.

2. **Exhaustive Testing Is Impossible**

Let's consider the following example [10]. Consider a piece of code to test with four decisions (A, B, C, D) and six statements (I1, I2, I3, I4, I5, I6), shown in Fig. 1.6. The code is executed in a loop, with at most 20 loop iterations. In the first iteration, we have five possible paths P1–P5 (the "!" symbol indicates the falsity of the decision, for example, "!A" means that the decision A is false; the truth of the condition results with a control flow indicated by the right arrow and the falsity by the left arrow):

P1: A I1 I6
P2: !A B D I5 I6
P3: !A B !D I4 I6
P4: !A !B C I3 I6
P5: !A !B !C I2 I6

Two loop iterations can realize 25 possible paths:

P1 P1; P1 P2; P1 P3; P1 P4; P1 P5;
P2 P1; P2 P2; P2 P3; P2 P4; P2 P5;
P3 P1; P3 P2; P3 P3; P3 P4; P3 P5;
P4 P1; P4 P2; P4 P3; P4 P4; P4 P5;
P5 P1; P5 P2; P5 P3; P5 P4; P5 P5.

Three loop iterations give $5^3 = 125$ possible paths. In general, if the loop is executed n times, we have 5^n possible path realizations. Since we assumed that the loop will execute at most 20 times, the number of different possible realizations of the control flow paths will be:

$$5 + 5^2 + 5^3 + \ldots + 5^{20} = 119{,}209{,}289{,}550{,}780 > 10^{14}.$$

If we assume that we need only 0.001 s to test one path, we would need about 3860 years to test all paths. Unfortunately, we don't have that much time!

Notice that in the above example, we analyzed a very simple (and restricted in terms of the number of loop iterations) problem. In real life, to fully (thoroughly) test a given application, one would need to check:

- Every possible input value for each variable (including result and working variables)
- Every possible sequence of program execution
- Every possible hardware/software configuration
- All possible—but generally *unimaginable*—ways of use of the tested product by the end user

Thorough testing must mean that after testing is completed, everyone will be sure that no failures will occur, but this is impossible (except in trivial cases) [13]. Instead of thorough testing, risk analysis and prioritization should be used to guide testing. This means that software tester role is essential in the software development cycle. In addition, testers must have mastered certain test techniques.

3. **Early Testing Saves Time and Money**

Early testing is sometimes called *shift-left*. Test activities should start as early as possible for the software under test. This saves time and money, because defects that are fixed early in the process will not cause subsequent defects in derived work products such as design or code. This, in turn, increases programming productivity, reduces development costs and time, and can have a positive impact on the required testing effort [14]. The overall cost of quality will be reduced because there will be fewer failures later in the development cycle. Testing should always be directed toward meeting well-defined objectives. Even if we are not ready to perform dynamic testing (because, e.g., the software has not yet been implemented), we can perform static testing, documentation reviews, design reviews, etc.

Figure 1.7 shows the famous so-called Boehm's curve, which illustrates the relation between the cost of fixing a defect and the time elapsed to its finding (we assume that the defect was introduced in the *Analysis* phase). This curve is exponential, meaning that the later a defect is found, the greater the increase in the cost of repairing it. Modern research suggests that this curve does not quite rise as fast, but nevertheless its increase in each case is significant, which clearly suggests that finding defects early is very cost-effective.

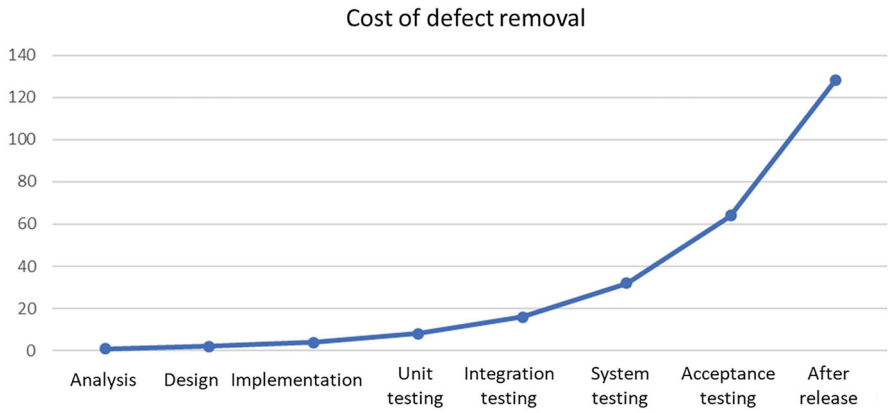

Fig. 1.7 Cost of defect removal as a function of time

4. **Defects Cluster Together**

Defects are not evenly distributed either in the software or over time. Most of the defects found in prerelease testing of software or that cause production failures are found in a small number of components [15]. As a result, the predicted defect clusters and the defect clusters actually observed during the testing or operational phase are an important part of the risk analysis that is done to guide testing efforts accordingly. This does not mean that there are fewer defects in the other components—it's just that within testing (or in production), we focus on the most user-relevant paths, and that's where we find most of the defects.

When defects accumulate, a well-known principle applies, the so-called Pareto rule, which states that a small number of causes cause a large number of effects. In testing terminology, for example, it could be translated like this: about 20% of components contain about 80% of defects.

Figure 1.8 shows a typical distribution of the number of defects in the components (histogram) and the cumulative number of defects (line). Components are sorted in descending order by the number of defects. Typically, only a few components have a very large number of defects, and the rest have few. The graph shows an example of the application of the Pareto rule to optimize effort. If we know (e.g., by estimation or reference to historical data) the distribution of the expected number of defects, as in the aforementioned chart, we can concern ourselves with testing a small number of the most defective components, so that we can detect most defects in a short time. For example, the analytical and reporting components contain a total of 320 defects, that is, 20% of the components (2 out of 10) contain about 70% of all predicted defects (320 out of 460).

If during testing we see that with comparable testing effort we detect many more defects in component A than in component B, this means that there are probably even more undetected defects in component A. A rational approach based on the

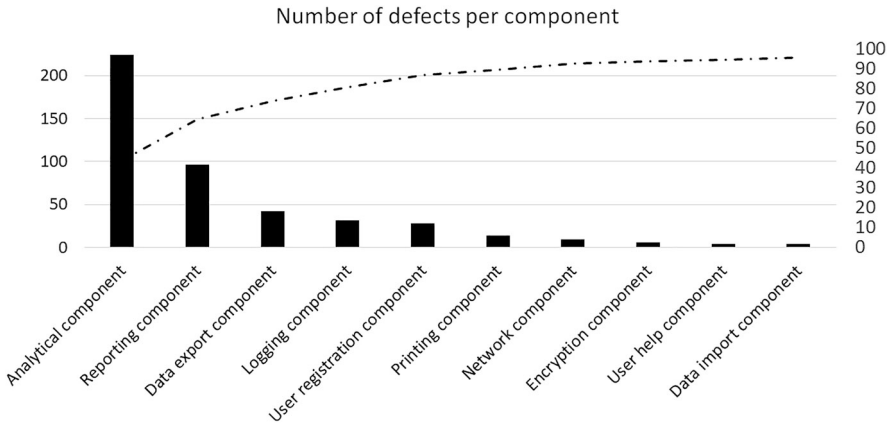

Fig. 1.8 Example of Pareto-like distribution

"Defects cluster together" principle would require focusing even more on component A than on component B in this situation.

5. **Tests Wear Out (or the "Pesticide Paradox")**

If the same tests are repeated again and again, then—after changes that lead to the removal of detected defects—no more new defects are found [16]. To overcome this phenomenon, test cases must be *regularly reviewed and modified*. Moreover, in order to test new or corrected parts of the system under test, new tests must be created or ran with a different set of test data.

Unmodified tests lose their ability to detect defects over time. Sometimes—for example, with automated regression testing—the pesticide paradox can be beneficial, because it allows us to confirm that the number of defects associated with regression testing is small (in the case of regression testing, we rather care that the tests always pass).

6. **Testing Is Context Dependent**

This is a fairly obvious principle: testing should be done differently in different situations. We pay attention to something else when testing *life-critical* systems, something else when testing banking systems—here the most important thing is functional accuracy (e.g., correct calculation of interest on capital)—and something else when testing computer games, here non-functional testing attributes such as performance (smoothness of play) or usability (interesting game interface) will probably be more important. Nevertheless, functional correctness (e.g., the two-headed horse in Fig. 1.9) should also be paid attention to in games.

The "context" mentioned in the principle is very broad in scope. It concerns, among other things, the nature of the software being developed, the business domain within which the software is being developed, project and product risks, functional

Fig. 1.9 Two-headed horse in a computer game

and non-functional requirements, characteristics of the target user group, all kinds of risks and their consequences related to the incorrect operation of the software, legal regulations, practices, existing norms and standards in the field, etc.

A consequence of this principle is that there is no one-size-fits-all approach to testing [17]. Testing by its very nature is an intellectual process, requiring knowledge, skill, and often a good deal of intuition and creativity.

7. **Absence-of-Defects Fallacy**

Some organizations still expect testers to be able to run all possible tests and detect all possible defects, but principles (1) and (2) show that this is impossible. It is also wrong to believe that simply finding and fixing a large number of defects will ensure successful system implementation, because even a defect-free application (correct verification) may not meet user requirements (incorrect validation).

This principle basically says that within the test process, verification alone is not enough—you still need validation, by which we make sure that the program meets the customer's requirements, and not just the technical assumptions that the project

team made based on the requirements [14]. We can create a perfect, defect-free product that is completely useless from the user's point of view.

1.4 Test Activities, Testware, and Test Roles

FL-1.4.2 (K2) Explain the impact of context on the test process.
FL-1.4.3 (K2) Differentiate the testware that supports the test activities.
FL-1.4.4 (K2) Explain the value of maintaining traceability.
FL-1.4.5 (K2) Compare the different roles in testing.

1.4.1 Test Activities and Tasks

There is no one-size-fits-all software testing process. However, there are typical and essential test activities necessary to achieve the established goals. These activities form the test process. Which test activities are included in this test process, how they are implemented, and when they take place are usually defined in the organization's test strategy or test approach as part of test planning for a specific situation (see Chap. 5).

It is a good practice to define measurable **coverage** 🕮 criteria for the test basis (for each test level or test type under consideration). In practice, they can act as so-called key performance indicators *(*KPIs) that favor the performance of specific activities and allow the test team to demonstrate the achievement of test objectives (e.g., coverage criteria may require at least one test for each test basis item).

The test process may or may not be formally defined. In typical situations, it consists of the following groups of activities:

1. Test planning
2. Test monitoring and test control
3. Test analysis
4. Test design
5. Test implementation
6. Test execution
7. Test completion

For example, software development in *agile* methodologies involves small iterations of designing, building, and testing software that takes place continuously, supported by ongoing planning. Therefore, testing also takes place in an iterative, continuous manner within this manufacturing approach. Even in sequential development life cycles, groups of the test process activities, presented above as sequential, in the actual process can occur iteratively, overlap, occur simultaneously (e.g., in exploratory testing), or be skipped. Time relations between these activities always depend on the specific project. Contextual factors that affect the selection of an organization's test process include:

- Software development lifecycle and project methodologies used
- Test levels and test types considered
- Product risks and project risks
- Business domain
- Contractual and regulatory requirements
- Operational limitations
  - Budgets and resources
  - Schedules
- Complexity of the domain
- Test policy and practices of the organization
- Required internal and external norms/standards

The following will discuss the *activities* within each group of the test process shown in Fig. 1.10. Section 1.4.3, in turn, will discuss the *work products* produced within these activities.

Test Planning—Activities

Test planning 🕮 involves defining test objectives and the test approach to achieve them within the constraints imposed by the context. Test planning is further explained in Sect. 5.1.

Typical test planning activities include:

- Defining test objectives
- Identifying the test activities needed to fulfill the project's mission and meet the test objectives
- Defining an approach to achieving test objectives within the limits set by the context
- Determining appropriate test techniques and test tasks
- Formulating a test execution schedule
- Defining metrics

Test plans can be revised based on feedback from test monitoring and test control activities. Test planning should be an ongoing activity.

Test monitoring and test control—activities

Test monitoring 🕮 is the continuous comparison of actual and planned test progress using metrics specifically defined for this purpose in the test plan. **Test control** 🕮 is the proactive taking of actions that are necessary to achieve the objectives set in the test plan (taking into account its possible updates). These actions are taken on the basis of monitoring information. Test monitoring and control are further explained in Sect. 5.3.

Progress against the test plan is communicated to stakeholders in written or verbal test progress reports (see Sect. 5.3.2), which include any noteworthy deviations from the plan as well as testing impediments and related work-arounds.

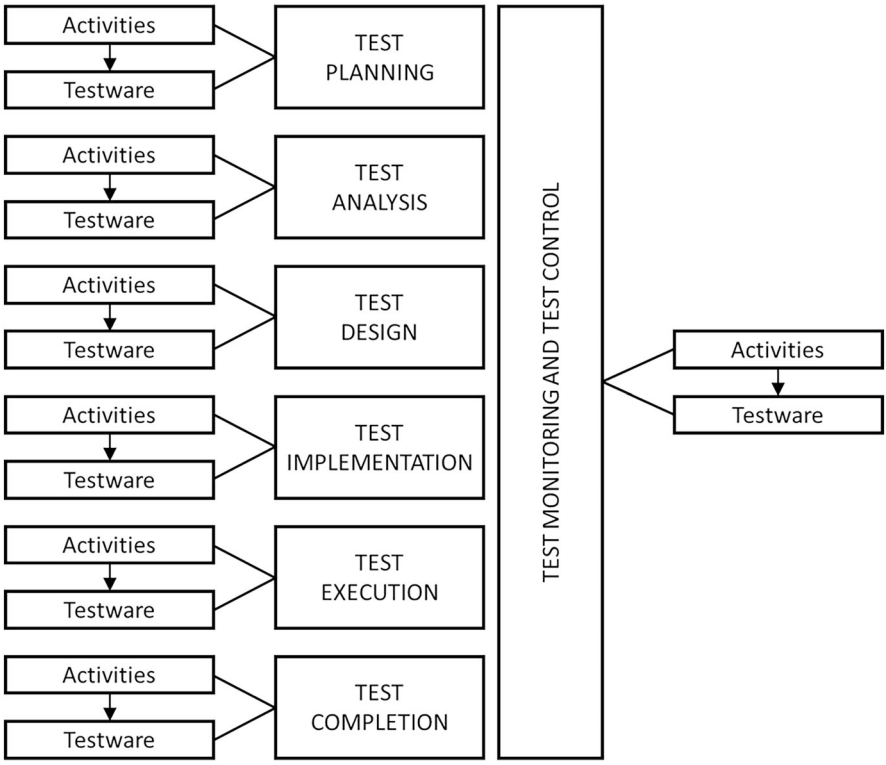

Fig. 1.10 Activities and work products in the test process

An element that supports test monitoring and control is the evaluation of exit criteria (often called the Definition of Done (DoD) in an agile approach) from the test plan, which can include:

- Checking the test results and test logs against the specified coverage criteria
- Estimating the quality level of a component or system, based on the test results and test logs
- Determining whether further tests are necessary (if the tests performed so far do not achieve the original product risk coverage level; this involves writing and executing additional tests)
- Informing stakeholders about the progress of the test plan
- Writing test progress reports

Test Analysis—Activities

The task of **test analysis** 🕮 is to look at the **test basis** 🕮 and analyze it to identify testable features, define the associated test conditions, and determine "what to test" (in terms of measurable coverage criteria). The general test objectives are transformed into specific test conditions.

The test basis refers to any documentation or information that describes how software should work and serves as a foundation or reference for designing and executing test cases. Common examples of test basis include:

- Requirements specification
- Design specification
- Use cases
- User stories
- Source code
- Business rules

A **test condition** 🕮 is any kind of property, feature, or attribute of the software that can be *checked* using tests. Test conditions are initial ideas for testing. They typically do not contain expected results. For example, in the case of testing an ATM, the following test conditions can be defined: "verify that the ATM correctly recognizes payment cards," "verify that the ATM accepts the correct PIN code entered by the user," "verify that the user can print the account balance," etc. Test conditions are the basis for deriving test cases and test data.

Test analysis activities verify that the requirements:

- Are consistent
- Are correctly expressed
- Are complete
- Are testable (are suitable for deriving acceptance criteria)
- Are ready for starting developing the software (the Definition of Ready—DoR)
- Do not need further grooming and thus can be used as a source for estimation
- Properly reflect the needs of the customers, users, and other stakeholders

Typical test analysis activities include:

- Familiarizing with the test basis that defines the desired functional and non-functional behavior of a component or system
- Analysis of design and implementation information such as diagrams or documents describing system or software architecture, control flow diagrams, UML diagrams [18], entity relationship diagrams, interface specifications, or similar work products; these documents define the structure of a component or system
- Analysis of the implementation of the component or system itself: code, metadata, database queries, and interfaces
- Analysis of risk analysis reports, which may cover functional, nonfunctional, and structural aspects of a component or system
- Assessing testability of the test basis to identify common types of defects: ambiguities, omissions, inconsistencies, inaccuracies, contradictions, redundant (unnecessary) instructions
- Identifying the features and feature sets to be tested
- Defining test conditions for individual features and prioritizing them based on test basis analysis, taking into account functional, nonfunctional, and structural parameters, other business and technical factors and risk levels

- Creating bidirectional traceability between test basis elements and their associated test conditions

Use of black-box, white-box, and experience-based test techniques can be useful as part of test analysis to reduce the likelihood of missing important test conditions and to define more precise and accurate test conditions (see Chap. 4). More formal test conditions are often represented as so-called test models (e.g., state transition diagrams, decision tables, control flow diagrams).

Identifying defects in the test basis as a result of test analysis is an important benefit, especially when a separate review of the test basis is not conducted. Test analysis activities can not only verify that requirements are consistent, properly expressed, and complete but also verify that requirements properly address the needs of customers, users, and other stakeholders.

Test Design—Activities

During **test design** 🕮, test conditions are transformed into **test cases** 🕮 at a high (logical) level, collections of such test cases, and into other testware. Test design answers the question "how to test."

Transforming test conditions into test cases often involves identifying test coverage items and using test techniques (see Chap. 4). Test coverage items also serve as guidelines for determining test case input data and in some situations may even define this data (e.g., values identified by the boundary value analysis).

Test design precedes test implementation, and sometimes they can be done simultaneously, but it is important to distinguish between the two activities. By designing tests before test implementation, you can identify defects in the test design, thus reducing the risk of wasted implementation time and effort.

Test design activities include:

- Designing (sets of) high-level test cases and prioritizing them
- Identifying the necessary test data
- Identifying the requirements for the test environment(s)
- Identifying any necessary tools and infrastructure elements
- Creating bidirectional traceability between test basis, test conditions, test cases, and test procedures (expanding the traceability matrix)
- Identifying defects in the test basis

Test Implementation—Activities

During **test implementation** 🕮, the tester creates and/or finalizes the testware necessary for test execution, including, but not limited to, transforming high-level (logical) test cases into low-level (concrete) test cases, assembling test cases into **test procedures** 🕮, creating automated test scripts, acquiring **test data** 🕮, and often implementing a test environment. Therefore, while test design answers the question "how to test," test implementation answers the question "do we have everything we need to run the tests?"

Test implementation activities include:

- Where needed: making high-level (logical) test cases more concrete by specifying detailed test data (e.g., high-level test case that requires to "Enter an age between 18 and 65" might result in multiple concrete test cases requiring to "Enter an age of 18," "Enter an age of 42," and "Enter an age of 65")
- Developing test procedures and prioritizing them
- Creating test suites (based on test procedures) and automated test scripts (if test automation is used)
- Organizing test sets into a test execution schedule to ensure that the entire process runs efficiently
- Building a test environment, including—if necessary—mock objects, (drivers, stubs), service virtualization, simulators, and other infrastructure elements, and verifying that it has been configured correctly
- Preparing the test data and verifying that it has been correctly loaded into the test environment
- Verifying and updating traceability between test basis, test conditions, test cases, test procedures, and test sets

Test Execution—Activities

During **test execution**, 🕮 test suites are run according to the test execution schedule. Within this phase, the following activities are performed:

- Registering the identification and version data of test items, test objects, test tools, and other testware
- Performing tests manually or with tools, including smoke[2] or sanity tests
- Comparing actual test results with expected ones
- Analyzing anomalies to determine their likely causes (e.g., defects in the code, false positives)
- Reporting defects, based on observed failures
- Logging the test execution results (passed, failed, blocking test)
- Repeating the necessary testing activities (confirmation testing, execution of a revised test, regression testing)
- Verifying and updating bidirectional traceability between the test basis and all the testware used

Test Completion—Activities

In the **test completion** 🕮 phase, the data from completed test activities is collected to consolidate lessons learned, testware, and other relevant information. This is done when project milestones are reached, such as:

- Handing over the software system for operation
- Completion (or cancellation) of the project

[2] Smoke test is a quick, simple test to check the correct implementation of basic functionality.

- Completion of an iteration of an agile project (e.g., after a demo or in a retrospective meeting)
- Completion of a test level
- Completion of work on the maintenance release

As part of the test completion, the following activities are performed:

- Checking that all defect reports are closed and creating change requests or Product Backlog items for any unresolved defects
- Identifying and archiving any test cases that may be useful in the future
- Handing over the testware to the operations team, other project teams, or other stakeholders who could benefit from its use
- Bringing the test environment to an agreed state
- Analyzing completed test activities to identify lessons learned and identify improvements for future iterations, releases, or projects
- Creating a report on the completion of testing and distributing it to stakeholders

1.4.2 Test Process in Context

Testing is not done in isolation. It is a process that supports the development process established within an organization. Testing is also a process sponsored by stakeholders, each of which has requirements or expectations from the final product. Therefore, the testing process must be aligned with the organization's established software development process, and how it is constructed in detail will depend on a number of contextual factors. These factors include, in particular:

- Stakeholders (needs, expectations, requirements, including business requirements for the product, willingness to cooperate with the test team, etc.)
- Team members (skills, knowledge, level of experience, availability, training needs, relationships with other team members, etc.)
- Business domain (identified product risks, market needs, specific legal conditions, etc.)
- Technical factors (project architecture, technology used, etc.)
- Project constraints (project scope, time, available budget, available resources, project risks, etc.)
- Organizational factors (organizational structure, existing policies, including test policies, practices used, etc.)
- Software development life cycle (engineering practices, development methods, etc.)
- Tools (availability, usability, compliance, etc.)
- Policies (data, privacy, cookies, etc.)

The above factors will significantly affect how the test process is organized and carried out in the project. In particular, they will affect:

- Test strategy
- Test techniques used
- Degree of test automation
- Required test coverage level for the requirements and identified risks
- Level of detail and type of test documentation to be developed
- Level of detail of test progress reporting
- Level of detail of defect reporting

1.4.3 Testware

The test process produces **testware** 🕮, which are the work products associated with testing (see Fig. 1.9). They may have different types and different names in different organizations. A good reference for testing work products is the international standard ISO/IEC/IEEE 29119-3 [3]. It is worth noting that many test work products can be created and managed using test management tools and defect management tools.

Test Planning—Work Products
Typical test planning work products are:

- Test plan
- Risk register
- Entry criteria and exit criteria

A risk register is a list of risks identified by the team, along with information on their probability, impact, and how they will be mitigated (see Sect. 5.2). The risk register and entry and exit criteria are usually part of the test plan. In larger projects, there may be more than one test plan, e.g., several plans relating to specific test levels (system integration test plan, acceptance test plan, etc.).

The test plan contains information on the test basis, to which other test work products will be linked via bidirectional traceability information. It defines the exit criteria (Definition of Done) that will be used for test monitoring and test control. Test plans can be adjusted based on feedback from test monitoring and test control. It is important to remember that planning is an ongoing process; it does not end at the initial phase of the project. Adjustments of plans can occur at any point in the software development life cycle, but any adjustment must be approved by the stakeholders.

Example Acceptance test plan template (based on [19]).

XYZ system acceptance test plan
ID, author, date, revision history
1. Introduction (purpose of the document, basis for its development, description of the system)
2 Test approach
2.1 Preliminary assumptions
2.1.1 Entry criteria for testing (available and required documentation, accepted test execution schedule, logistical and organizational conditions, completed preliminary tests, available documentation from preliminary tests)
2.1.2 Entry criteria for subsequent test iterations (fixed defects from previous iterations, possibly no defects of sufficiently high priority, necessary modifications to the test documentation to the extent resulting from the previous iteration, achieving adequate coverage)
2.1.3 Types of acceptance tests (describe the types of acceptance tests performed; identify where they are performed; identify who is to perform them, e.g., alpha testing, beta testing, user acceptance testing, operational acceptance testing; identify the roles of stakeholders in performing these tests—e.g., the role of the contractor, the role of the key users)
2.2 Test organization
2.2.1 Personnel resources (required roles, qualifications, team compositions, role descriptions)
2.2.2 Test procedure (general and specific procedures describing the course of acceptance testing: prerequisites, how to provide information, such as test data by the contractor)
2.2.3 Defect classification (description of defect classification, e.g., critical/serious/low priority, with description of actions in case of a defect of a given category)
2.2.4 Defect reporting procedure (process definition, tools used)
2.2.5 Defect handling (the process of handling defect reports and resolving them)
2.2.6 Progress reporting (description of work reporting rules, types of reports)
2.2.7 Conditions for acceptance of different types of tests
2.3 Test environments (description of environments, infrastructure and test data, how they are acquired and maintained; division of data into dictionary and operational data, procedures for handling data)
2.4 Test process schedule
2.5 Types and levels of tests performed (e.g., regression tests, scenario-based tests, exploratory tests)
3 Testing
3.1 Test cases (hierarchical list of related test cases with tracking to test scenarios, risks and other relevant elements; may be included in separate documents)
3.2 Test execution schedule (the order of test case execution)
3.3 Test scenarios (specification of scenarios; may be included in a separate document)
4 Attachments (e.g., excerpts from customer contracts, sample reports, and other necessary documentation)

For more information on the purpose and content of the test plan, see Sect. 5.1.1.

Test Monitoring and Test Control—Work Products
The essential work products in this group are:

- Test progress reports (see Sect. 5.3.2)
- Documentation of control directives (see Sect. 5.3)
- Risk information (see Sect. 5.2)

Test progress reports are created on an ongoing basis and/or at regular intervals, as agreed with the stakeholders. All test progress reports should contain audience-relevant details of current test progress, including a summary of test execution results as they become available. The reports should always be tailored to the needs of

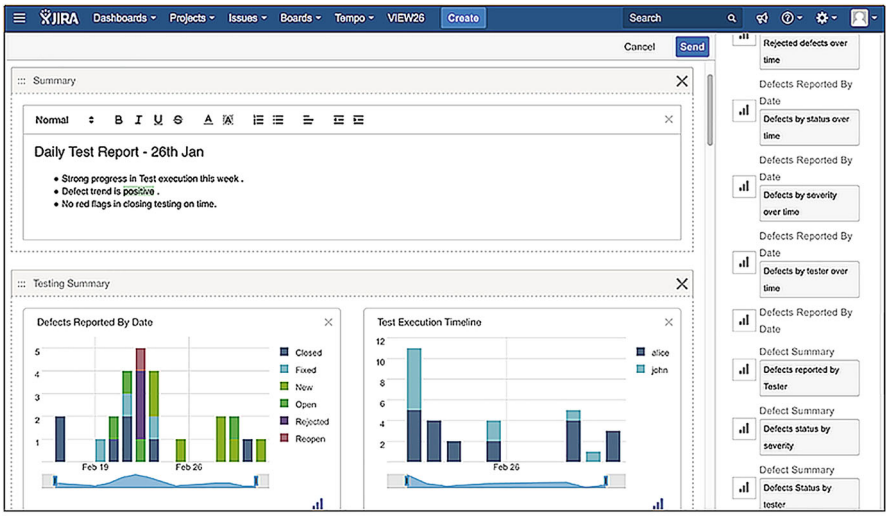

Fig. 1.11 Sample daily test report (source: community.atlassian.com)

specific audiences. In addition, test progress reports should include project management issues, information on completed tasks, and resource allocation and consumption.

Risk information can be stored in the project's risk register or locally in the test plan.

Example An example of a test progress report is shown in Fig. 1.11. It is a daily test report with data compared to the previous days. The left graph describes the distribution of the number of defects in specific time intervals, broken down by category (closed, fixed, new, open, rejected, reopen). The right graph shows the daily test execution by two testers (Alice and John).

Test Analysis—Work Products
Typical work products of the test analysis are:

- (Prioritized) Test conditions
- Acceptance criteria (see Sect. 4.5.2)
- Defect reports in the test basis (if not fixed directly)

Test conditions are usually defined and prioritized. Acceptance criteria can be thought of as the equivalent of test conditions in agile software development methodologies. For exploratory testing, the work products may contain test charters. Test analysis can also result in the detection and reporting of defects in the test basis, which means that test analysis work products are also defect reports.

Example When testing the login functionality of a service, we can identify the following test conditions:

- Correct login (existing user, correct login, and correct password)
- Incorrect login (existing user but incorrect password)
- Incorrect login with account blocking (entering the wrong password three times, which should result in the blocking of the account)
- Incorrect login (non-existent user)

Test Design—Work Products
The work products of test design are in particular:

- High-level (logical) test cases
- Coverage items
- Test data requirements
- Test environment design

It is often good practice to design high-level test cases, which do not include specific input data values and expected results. High-level test cases can be reused multiple times in different test cycles with different test data, properly documenting the scope of the test case. Ideally, for each test case, there is bidirectional traceability between that test case and the test condition(s) it covers. The advantage of using high-level (logical) test cases is that they allow flexibility. On the other hand, these test cases endanger reproducibility.

Among the work products of the test design phase may also be the design and/or identification of the necessary test data, the design of the test environment, and the identification of infrastructure and tools. The degree of documentation of these work products may vary.

Test conditions defined in the test analysis phase can be refined during test design.

Example The following example shows a test case with test execution *steps* (*test steps*). Notice that this is an example of a low-level test case, because it contains the detailed test data (concrete website address, login, and password).

TC No. 20.002.11	Author: Joan Smith
Priority: medium	Date: 11.07.2020
Function: logging	
Description: trying to log in with the correct login and password using the *Enter* key instead of clicking on the login button	
Prerequisites: the user has a login and password	
Step/Test data	**Expected result**
Open the login page www.our.company.loginpage.com.	Page loaded
Enter login "johnsmith".	Login accepted
Enter the password "password123".	Password accepted, default active *Login* button
Press *Enter*.	Logging into the service
Exit conditions: user logged in, the fact of logging in saved to the database along with the time of logging in	

Test Implementation—Work Products
Within this group of activities, the following work products are created:

- Low-level (concrete) test cases
- Test procedures (including the order in which they are executed);
- Automated test scripts
- Test sets
- Test data
- Test execution schedule
- Elements of the test environment

Test data is used to assign specific values to the input data and expected results of test cases. These specific values, along with specific guidelines for their use, transform high-level test cases into executable low-level test cases. Typically, one high-level test case results in multiple low-level test cases. The same high-level test case can be executed using different test data for different versions of the test object. Specific expected results associated with specific test data are identified using the test oracle.

Examples of test environment components are:

- Mock objects (e.g., stubs, drivers)
- Simulators
- Service virtualization

In exploratory testing, some work products related to test design and implementation can be created during test execution, although the extent to which exploratory tests are documented and traceable to specific elements of the test basis varies widely.

Sometimes, the work product of this group is a description of the use of tools (e.g., for service virtualization). The most typical work products of this phase are automated test scripts.

Example Suppose we are testing the function `multiply(x, y)`, which takes two integers as input and returns their product. During the test analysis, the following test condition was defined: "check the correctness of multiplication involving zero value." The test design transforms this test condition into three test cases involving multiplication by zero:

- TC1: the first parameter is zero, and the second parameter is different from zero.
- TC2: the second parameter is zero, and the first parameter is different from zero.
- TC3: both parameters are equal to zero.

The following inputs and expected outputs were defined for these cases:

- TC1: $x = 0$, $y = 10$, expected output: 0
- TC2: $x = 10$, $y = 0$, expected output: 0
- TC3: $x = 0$, $y = 0$, expected output: 0

The following automated test script is an example of implementing three test cases, TC1, TC2, and TC3, in Java using the JUnit library (used to help create component tests). The assertEquals statement checks whether the first two of its parameters have the same value. In our case, we are checking whether the results of the products of 10*0, 0*10, and 0*0 will actually equal to 0.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class MyTests {
    @Test
    public void multiplyValueNumbersBy0Day0() {
      MyClass tests = new MyClass(); // MyClass tests

      // assertions implementing the cases PT1, PT2, PT3
      assertEquals(0, tests.multiply(10, 0), "10 x 0 must be 0");
      assertEquals(0, tests.multiply(0, 10), "0 x 10 must be 0");
      assertEquals(0, tests.multiply(0, 0), "0 x 0 must be 0");
    }
}
```

Test Execution—Work Products
Typical test execution work products are:

- Test logs
- Documentation of the status of test procedures
- Defect reports (see Sect. 5.5)
- Documentation indicating what was used in testing (e.g., test objects, test tools, and testware)

In ideal situations, the status of individual test basis elements can also be reported. Such reports are feasible, thanks to the bidirectional traceability to the corresponding test procedures (see Sect. 1.4.4). It is possible, for example, to indicate which requirements have been fully tested, which have failed testing and/or involve defects, and which have not yet been fully tested. This makes it possible to check whether test coverage criteria have been met and to present test results in reports in a way that stakeholders can understand.

Example Figure 1.12 shows the report screen of the execution of two unit tests, `testCaseA` and `testCaseB`, using the JUnit5 library. The colors of the icons next to the names of the tests symbolize the results of the tests: green color means that the test is passed and red color failed. In our case, all colors are green, which means that both test cases pass.

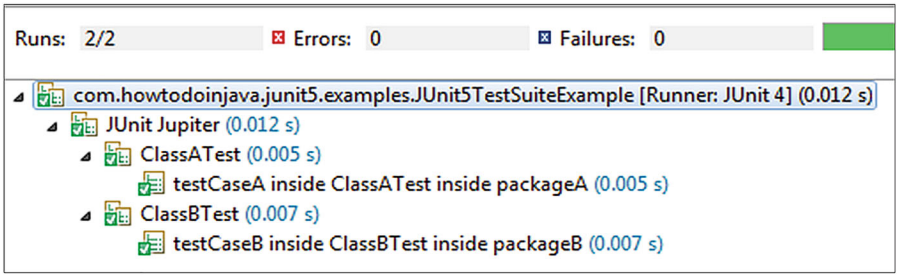

Fig. 1.12 Component test execution result in JUnit (source: howtodoinjava.com)

Test Completion—Work Products
Typical work products of the test completion are:

- Test completion report (see Sect. 5.3.2)
- Action items to improve subsequent projects or iterations (e.g., retrospective action items transformed to Product Backlog items for future iterations)
- Change requests (e.g., as elements of a product backlog)

Test completion reports are created when individual milestones are reached. These reports should include detailed information on the progress of the testing process to date, a summary of the results of test execution, and information on any deviations from the plan and corrective actions.

1.4.4 Traceability Between the Test Basis and Testware

In order to ensure effective test monitoring and control, it is important to establish and maintain a mechanism for traceability between each element of the test basis and its corresponding testing work products (e.g., test conditions, test cases, risks, etc.) throughout the testing process. Efficient traceability enables:

- Evaluation of test coverage
- Analyzing the impact of change
- Conducting test audits
- Meeting criteria related to IT management
- Creating easy-to-understand test status reports and summary test completion reports
- Presenting the status of test basis elements (requirements for which tests have been passed, failed, or are waiting to be executed)
- Providing stakeholders with information about the technical aspects of testing in a form they can understand
- Providing the information needed to assess product quality, process capabilities, and project progress against business objectives

Table 1.3 Traceability matrix

	Risk 1	Risk 2	Risk 3	Risk 4
TC001	X	X		
TC002		X		
TC003			X	
TC004	X		X	X

Coverage criteria can function as KPIs to demonstrate achievement of test objectives (see Sect. 1.1.1). For example, by leveraging the ability to track from:

- Test cases to requirements, we can verify that the test cases' coverage of the requirements is met
- Test case results to risks, the residual risk level in the test object can be assessed

Example Consider the traceability matrix for test cases and identified risks shown in Table 1.3.

An "X" in row corresponding to test case T and column corresponding to risk R indicates that a test case T covers a risk R. For example, TC001 covers Risks 1 and 2, TC003 covers Risk 3, and so on. Let us assume that TC001, TC002, and TC004 pass, while TC003 fails. This could mean that Risk 3 is not covered (if we require that risk is covered when *all* related tests pass) or, for example, that it is covered in 50% (if we define risk coverage as the percentage of related test cases that pass).

1.4.5 Roles in Testing

The Foundation Level syllabus distinguishes two fundamental roles in testing: a test management role (managerial) and a testing role (technical). The activities and tasks assigned to these roles depend on a number of factors, such as the context of the project or product, the adopted model of the development life cycle, the skills of people, and the organization of work in the team.

The person (or team) in the test management role is responsible for implementing the test process, organizing the work of the test team, and directing test activities. Test management tasks mainly focus on activities related to:

- Test planning
- Test monitoring and test control
- Test completion

A person in the testing role has overall responsibility for the engineering (technical) aspect of testing. Testing tasks mainly focus on:

- Test analysis
- Test design
- Test implementation
- Test execution

How the test management role is defined and understood depends in particular on the software development life cycle model adopted. Sometimes, this role is performed by a single person, usually called a test manager. In projects based on agile methodologies, for example, some management tasks may be left to the entire (self-organizing) team. Tasks affecting multiple teams or the entire organization, on the other hand, may be handled by test managers outside the development team.

It is also important to distinguish between *roles* and *positions* within the company. A position is usually permanently assigned to a specific person: each team member is usually employed in a specific position. On the other hand, each team member may perform different roles at different times. In particular, the test management role in testing may be that of a team leader, project manager, quality manager, etc. In larger projects, the test manager or coordinator may coordinate several test teams, led by test leaders or lead testers.

The testing role is performed by anyone who performs any testing activities at any given time. In this sense, for example, a developer at the time of component (unit) testing or a customer performing acceptance testing enters the testing role. It is also possible that one person performs both testing and management roles at one time.

Two roles will be discussed in detail below.

Test Management Role

The way in which the test manager's duties are carried out depends on the software development cycle. Typical tasks of a test manager are:

- Developing or reviewing test strategies and test policies
- Context-sensitive test project planning (see Sect. 5.1)
  - Creating and updating test plans
  - Iteration and release planning in agile projects
  - Choosing the test approach
  - Defining entry criteria and exit criteria
  - Introducing appropriate metrics for measuring the test progress and assessing the quality of testing and the product
  - Defining test levels and test cycles
  - Estimating the time, effort and cost of testing
  - Test prioritization
- Risk management (see Sect. 5.2)
- Monitoring test results, checking the status of exit criteria (definition of done), and performing test control, e.g., by adjusting plans according to test results and progress (see Sect. 5.3)
- Process supervision:
  - Configuration management (see Sect. 5.4)
  - Defect management (see Sect. 5.5)
- Resource acquisition
- Coordinating test strategy and test plan with project managers and other stakeholders

- Presenting the testers' point of view as part of other project activities, such as integration planning
- Initiating processes for test analysis, test design, test implementation and test execution
- Reporting the test progress, creating a test completion report
- Supporting the team in the use of tools to implement the testing process (e.g., raising funds for the purchase of the tool, purchasing licenses, controlling the implementation of the tool in the organization)
- Deciding on the implementation of test environments
- Promoting testers and the test team and representing their point of view within the organization
- Developing testers' skills and careers through a training plan, performance evaluations, and coaching

Testing Role
Typical tasks of a tester are:

- Reviewing test plans and participating in their development
- Co-authoring the requirements (user stories) while performing collaborative user story writing
- Deriving testable acceptance criteria for each Product Backlog item
- Analyzing, reviewing, and evaluating the test basis (i.e., requirements, user stories and acceptance criteria, specifications, and models) for testability
- Identifying and documenting test conditions and recording the relationship between test cases, test conditions, and test basis
- Designing, configuring, and verifying test environments (generally in consultation with system and network administrators)
- Designing and implementing test cases, test procedures, and test scripts
- Preparing and acquiring test data
- Co-creating the test execution schedule
- Performing tests, evaluating results, and documenting deviations from expected results
- Using appropriate tools to streamline the testing process (e.g., test automation tools)
- Evaluating and measuring nonfunctional characteristics of the software (see Sect. 2.2.2)
- Collaborating within the team (e.g., reviewing tests designed by others)
- Using—if necessary—tools for test management
- Test automation (e.g., creating, running, and modifying test scripts)

It is important to make sure that the people working on the test team (and other people performing testing in general), that is, those involved in test analysis, test design, specific test types, or automation are specialists in their roles. As mentioned earlier, depending on the test level and product and project risk, the testing role can be performed by different people:

- At the component and component integration levels—usually by developers
- At the system testing level—usually by testers, members of an independent test team
- At the acceptance test level—usually by business experts and users
- At the operational acceptance testing level—usually by system operators and system administrators

1.5 Essential Skills and Good Practices in Testing

FL-1.5.1 (K2) Give examples of the generic skills required for testing.
FL-1.5.2 (K1) Recall the advantages of the whole team approach.
FL-1.5.3 (K2) Distinguish the benefits and drawbacks of independence of testing.

1.5.1 Generic Skills Required for Testing

The process of software development, including testing, is carried out by people, and therefore, the skills of individual testers and psychological aspects of human behavior are of great importance for the course of testing.

Essential Skills Required in Testing
A good tester should be characterized by a number of qualities (skills), which will make their work effective and efficient. In particular, a good tester has the following characteristics (see also [20]):

- Testing knowledge (to increase effectiveness of testing, e.g., by using test techniques)
- Thoroughness, carefulness, curiosity, attention to detail, being methodical (to identify different types of defects, especially those difficult to find)
- Good communication skills, active listening, being a team player (to interact effectively with all stakeholders, communicate information to others, be understood, be able to report and discuss defects)
- Analytical thinking, critical thinking, creativity (to increase effectiveness of testing)
- Technical knowledge (to increase efficiency of testing, e.g., by using appropriate test tools)
- Domain knowledge (to be able to understand and to communicate with end users and business representatives)

Psychological Aspects in Testing
Identifying defects during static testing or identifying failures during dynamic testing can be perceived as criticism of the product or its author. There is a psychological phenomenon called the *confirmation bias*—the tendency to prefer

information that confirms previous expectations and hypotheses, regardless of whether that information is true or not. Confirmation bias may make it difficult to accept information that contradicts existing beliefs. It causes people to seek out information and remember it selectively, interpreting it in an erroneous way. This effect is particularly strong for issues that evoke high emotions and involve strongly held opinions. For example, when reading about gun access policy, people tend to prefer sources that confirm what they themselves think about the subject. They also tend to interpret inconclusive evidence as supporting their own opinions.

A series of experiments conducted in the 1960s showed that people tend to seek confirmation for their prior beliefs. Later research explained this as the result of a tendency to test hypotheses very selectively, focusing on one possibility and ignoring alternatives. Combined with other effects, such a strategy affects the conclusions people reach. This error may be due to the human brain's limited ability to process information or to evolutionary optimization, when the estimated costs of being stuck in error are no greater than the costs of analysis conducted in an objective, scientific manner.

Example Testers are convinced that the failures they report are the result of defects in the code; there is a confirmation effect, by which they find it difficult to accept situations where there is no defect and the failure was caused by an incorrect test execution (a false positive).

There are also other cognitive errors that make it difficult for people to understand or accept information obtained through testing. People often tend to blame the person bringing the bad news, and such situations often arise from testing, since testers are usually the bearers of bad news. Moreover, some people see testing as a destructive activity, even if it contributes significantly to project progress and product quality. To reduce similar reactions, information about defects and failures should be communicated as constructively as possible. Efforts should be made to reduce tension between testers and business analysts, product owners, designers, and developers. This applies to both static testing and dynamic testing.

Interpersonal Skills and Efficient Communication
Testers and test managers need to have strong interpersonal skills to efficiently communicate information about defects, failures, test results, testing progress, or risks and build positive relationships with colleagues. In this regard, the following rules of conduct are suggested:

- Cooperate, not fight.
- Emphasize the benefits of testing (authors can use defect information to improve work products and develop skills, and for organizations, detecting and fixing defects during testing mean saving time and money and reducing overall product quality risks).

(continued)

- Communicate test results and other findings in a neutral manner (focus on the facts, and do not criticize the authors of the flawed work product or solution).
- Create defect reports and review conclusions objectively and based on facts.
- Try to understand why the other person reacts negatively to the information given.
- Make sure the caller understands the information being conveyed and *vice versa*. This is important, especially if you are working on a distributed project.

Unambiguously defining the right set of test objectives has important psychological implications, because most people tend to align their plans and behavior with the goals set by the team, management, and other stakeholders. One should strive to ensure that testers adhere to the set objectives and that their personal mindset has as little impact as possible on the work they do. It is also important to remember that a person's mindset is determined by the assumptions they make and the ways they prefer to make decisions and solve problems.

1.5.2 Whole Team Approach

One important testing skill is to be a team player—to have the ability to work effectively in a team and make a positive contribution to the team's goal. This skill is the basis of the "whole team" approach.

The "whole team" approach is characterized by the following attributes:

- Involving all those with the necessary knowledge and skills to ensure the success of the project
- Relatively small teams of a few people
- Sharing the same workspace (be it physical or virtual) to make communication and interaction much easier

In agile software development, the "whole team" approach:

- Ensures that the team includes representatives of the customer and other business stakeholders who decide on product features
- Is supported through daily *stand-up meetings* involving all team members, during which progress is communicated and any obstacles to progress are pointed out
- Promotes more effective and efficient team dynamics
- Improves communication and cooperation in the team
- Enables the use of different skills of team members to the benefit of the project
- Makes *everyone responsible for quality*

The essence of the "whole team" approach is that developers, business representatives, and testers work together at every stage of the software development process. The close cooperation of these three groups is aimed at ensuring that the desired quality levels are achieved. This includes working with business representatives to help them create appropriate acceptance tests (see Sect. 4.5) and working with developers to agree on a testing strategy and decide on a test automation approach. Testers can also transfer testing knowledge to other team members and influence product development.

The entire team is involved in any consultations or meetings where product features are determined, analyzed, or estimated. The concept of involving testers, developers, and business representatives in all discussions about the features of the product under development is known as the "power of three" [21] or the "Three Amigos."

1.5.3 Independence of Testing

Tasks related to testing can be performed both by people with specific roles in the testing process and by people with other roles (such as customers). On the one hand, a certain degree of independence often increases the effectiveness of defect detection, since the author and tester may be subject to different cognitive errors (see Sect. 1.5.1). On the other hand, independence is not a substitute for product knowledge, and developers can effectively detect many defects in the code they develop. However, it should be remembered that the author often fails to see their own errors. When developers create code, they make certain assumptions about it. When they then have to test that code, they may—even unconsciously—write tests that verify their assumptions. As a result, tests written by the developers will generally detect few problems in their own code.

Independence of testing performed by an independent testing team has several advantages, in particular:

- Increases emphasis on testing
- Increases efficiency in finding defects and failures
- Provides additional benefits, such as the independent view of a trained and professional test team

Testing independence can take place at any level of testing. There are different levels of independence (ordered below from lowest to highest; see also Fig. 1.13):

- No independent testers—developers test their own code.
- Low independence—independent developers or testers working as part of a development team; developers can test products developed by colleagues in what is known as peer testing.

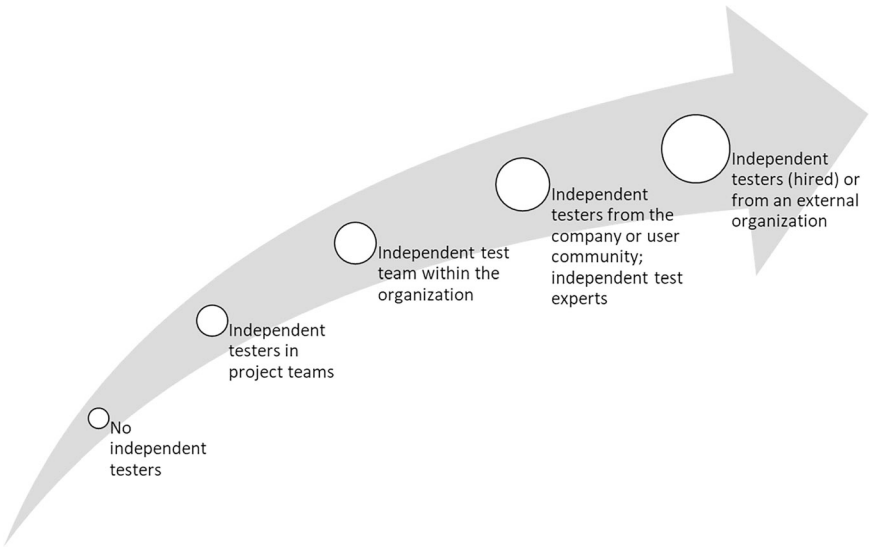

Fig. 1.13 Levels of testing independence

- High independence—testing carried out by an independent testing team operating within the organization (reporting to the project management or to the board of directors) or by representatives of other departments of the organization.
- Very high independence—independent testers from outside the organization, working *on-site* or off-site (*outsourcing*).

The optimal solution (for large, complex projects or in teams producing safety-critical products) is testing at multiple levels:

- Developers at lower levels—responsible for component testing and component integration testing (they can thus control the quality of their work; however, it should be remembered that the lack of objectivity limits their effectiveness)
- Independent testers at higher levels—responsible for system integration testing, system testing, and acceptance testing

When considering levels of independence, the software development life cycle model should also be taken into account. In the agile approach, it is accepted that testers may be assigned to work as part of the development team but may sometimes be considered members of a broader independent test team. Often product owners perform—at the end of each iteration—acceptance testing to validate user stories.

Independent testers have a different perspective on the product under test; they often see errors other than those noticed by developers due to different experiences, technical review points of view, and cognitive errors; they can verify correctness (reviews) and challenge or refute assumptions made by stakeholders in both specifications and system architecture. They also approach the test object without ready-made expectations and biases.

Independence of testing brings not only benefits but also some risks. In particular:

- Isolation of testers from the team, which can cause lack of communication, delays in providing feedback or poor relations with the production team (*we form one team*): this occurs most often in the case of full independence of testing from the team.
- Developers' loss of a sense of responsibility for quality ("since the product is being tested by independent experts, I no longer need to care about quality").
- The possibility of a bottleneck at the end of the project and holding testers responsible for the untimely commissioning of the product ("blame game").
- The risk that independent testers will not have important information (e.g., about the test object).

Sample Questions

Question 1.1
(FL-1.1.1, K1)

Which of the following is NOT a typical test objective?

A. Verifying that the test object is complete.
B. Triggering as many failures as possible during acceptance testing.
C. Reducing the risk level of previously undetected failures during software operation.
D. Building confidence in the quality level of the system being tested.

Choose one answer.

Question 1.2
(FL-1.1.2, K2)

The following is a list of activities related to defects and failures:

i. Finding defects in code.
ii. Triggering failures.
iii. Analyzing the defects found.
iv. Performing retests.

Which are the testing activities, and which are the debugging activities?

A. (ii) and (iv) are the testing activities; (i) and (iii) are the debugging activities.
B. (i) and (iv) are the testing activities; (ii) and (iii) are the debugging activities.
C. (ii) and (iii) are the testing activities; (i) and (iv) are the debugging activities.
D. (i), (iii), and (iv) are the testing activities; (ii) is the debugging activity.

Choose one answer.

Question 1.3
(FL-1.2.1, K2)

You are a tester on a project developing a game based on Greek myths. You are currently creating an acceptance criterion for the following user story:

As a	level 4 player
I want	to be able to use the Midas wand
So that	I can turn the object standing in front of me into gold and increase my financial resources

You noticed that there is no information regarding the time of turning an item into gold.

Which of the following statements BEST illustrates the testing's contribution to project success?

A. Inform the team that the user story author incorrectly performed their task.
B. Reduce the risk of producing an incorrect or untestable feature.
C. Force the product owner to fill in the missing data immediately.
D. Improve the time behavior, a sub-characteristic of performance.

Choose one answer.

Question 1.4
(FL-1.2.2, K1)

Consider the following statements about testing and quality assurance.

i. Quality assurance focuses on preventing failures by verifying that quality requirements are met
ii. Quality assurance focuses on controlling the quality of the product created
iii. Testing focuses on evaluating software and related products to determine whether they meet specified requirements
iv. Testing focuses on removing defects from software

Which of these are true?

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

Choose one answer.

Question 1.5
(FL-1.2.3, K1)
Which of the following is the correct example of a defect?

A. A human action causing an incorrect result.
B. A materialization in the code of the software developer's error.
C. A deviation from the expected software behavior.
D. A test case to check the system's response to erroneous data.

Choose one answer.

Question 1.6
(FL-1.3.1, K2)
According to the Pareto principle, most problems are caused by a small number of causes. This is the basis of one of the principles of testing. Which one?

A. Early testing saves time and money.
B. Testing is context dependent.
C. Tests wear out.
D. Defects cluster together.

Choose one answer.

Question 1.7
(FL-1.4.1, K2)
During which phase of the test process is the testability of the test basis checked?

A. Test planning.
B. Test design.
C. Test analysis.
D. Test implementation.

Choose one answer.

Question 1.8
(FL-1.4.2, K2)
Which of the following has the LEAST impact on an organization's test process?

A. Project budget.
B. External norms and standards.
C. Number of certified testers employed.
D. Testers' knowledge of the business domain.

Choose one answer.

Question 1.9
(FL-1.4.3, K2)
Which of the following is NOT a work product resulting from test monitoring and test control?

A. Test progress report.
B. Information about the current risk level in the product.
C. Documentation describing the control actions taken.
D. Test completion report.

Choose one answer.

Question 1.10
(FL-1.4.4, K2)
Which of the following is made possible by a mechanism for tracking the relationship between test basis elements and their corresponding testing work products?

A. Calculation of the risk level in the product based on test results.
B. Defining an acceptable level of code coverage for each component.
C. Using a test oracle to automatically determine the expected result for a test case.
D. Deriving test data that achieve full equivalence partitioning coverage.

Choose one answer.

Question 1.11
(FL 1.4.5, K2)
Which of the following are typical test management activities, and which are typical testing activities?

i. Coordinate the implementation of the test strategy and test plan.
ii. Defining test conditions.
iii. Creation of a test completion report.
iv. Implementing automated test scripts.
v. Deciding on the implementation of test environments.
vi. Verifying test environments.

A. (iv), (v), and (vi) are the test management activities; (i), (ii), and (iii) are the testing activities.
B. (ii), (iii), and (vi) are the test management activities; (i), (iv), and (v) are the testing activities.
C. (i), (ii), and (v) are the test management activities; (iii), (iv), and (vi) are the testing activities.
D. (i), (iii), and (v) are the test management activities; (ii), (iv), and (vi) are the testing activities.

Choose one answer.

Question 1.12
(FL-1.5.1, K2)
Which of the following skills is LEAST critical for the tester?

A. Analytical thinking.
B. Domain knowledge.
C. Programming skills.
D. Communication skills.

Choose one answer.

Question 1.13
(FL-1.5.2, K1)
Which of the following TWO activities MOST closely match the responsibilities related to the "whole team" approach?

A. Testers are responsible for the implementation of component tests, which they pass on to developers for execution.
B. Testers work with business representatives and developers in creating acceptance tests.
C. Business representatives select the tools that developers and testers will use during the project.
D. Nonfunctional tests are designed by the client and executed by testers and developers.
E. Not only testers but also developers and business representatives are responsible for the product quality.

Select TWO answers.

Question 1.14
(FL 1.5.3, K2)
Why is testing usually performed by independent testers?

A. It helps to increase the emphasis on testing and allows for the independent opinion of professional testers.
B. Because developers do not have the ability to test their code due to cognitive bias.
C. Because the failures detected are reported in a constructive manner.
D. Because finding defects is not seen as a criticism of developers.

Choose one answer.

Chapter 2 Testing Throughout the Software Development Life Cycle

Keywords

Acceptance testing	a test level that focuses on determining whether to accept the system.
Black-box testing	testing based on an analysis of the specification of the component or system. *Synonyms:* specification-based testing.
Component integration testing	the integration testing of components. *Synonyms:* module integration testing, unit integration testing.
Component testing	a test level that focuses on individual hardware or software components. *Synonyms:* module testing, unit testing.
Confirmation testing	a type of change-related testing performed after fixing a defect to confirm that a failure caused by that defect does not reoccur. *Synonyms:* re-testing.
Functional testing	testing performed to evaluate if a component or system satisfies functional requirements. *References:* ISO 24765.
Integration testing	a test level that focuses on interactions between components or systems.
Maintenance testing	testing the changes to an operational system or the impact of a changed environment to an operational system.
Nonfunctional testing	testing performed to evaluate that a component or system complies with nonfunctional requirements.

L. Stapp et al., *ISTQB® Certified Tester Foundation Level*,
https://doi.org/10.1007/978-3-031-42767-1_5

Regression testing	a type of change-related testing to detect whether defects have been introduced or uncovered in unchanged areas of the software.
Shift-left	an approach to performing testing and quality assurance activities as early as possible in the software development life cycle.
System integration testing	the integration testing of systems.
System testing	a test level that focuses on verifying that a system as a whole meets specified requirements.
Test level	a specific instantiation of a test process. *Synonyms*: test stage.
Test object	the work product to be tested.
Test type	a group of test activities based on specific test objectives aimed at specific characteristics of a component or system. *After* TMap.
White-box testing	testing based on an analysis of the internal structure of the component or system. *Synonyms:* clear-box testing, code-based testing, glass-box testing, logic-coverage testing, logic-driven testing, structural testing, structure-based testing.

2.1 Testing in the Context of a Software Development Cycle

FL-2.1.1 (K2)	Explain the impact of the chosen software development life cycle on testing
FL-2.1.2 (K1)	Recall good testing practices that apply to all software development life cycles
FL-2.1.3 (K1)	Recall the examples of test-first approaches to development
FL-2.1.4 (K2)	Summarize how DevOps might have an impact on testing
FL-2.1.5 (K2)	Explain the shift-left approach
FL-2.1.6 (K2)	Explain how retrospectives can be used as a mechanism for process improvement

A software development life cycle (SDLC) model is an abstract, high-level representation of the software development process. The SDLC model describes the activities included in the software development process and the relationships between them, both logical and temporal. SDLC models are used to determine how software will be developed. They also make it easier to understand the succession of different phases of the manufacturing process.

Examples of classic SDLC model types include:

- Sequential models (e.g., waterfall model, V model)
- Iterative models (e.g., Boehm's spiral model, prototyping)
- Incremental models (e.g., Unified Process)

Some activities within the software development process can also be described by more detailed models, development methodologies, or agile practices. Examples of such approaches include:

- Scrum
- Kanban
- Lean IT (development based on so-called lean management)
- eXtreme Programming (XP)
- Test-Driven Development (TDD)
- Acceptance Test-Driven Development (ATDD)
- Behavior-Driven Development (BDD)
- Domain-Driven Design (DDD)
- Feature-Driven Development (FDD)

For more information on SDLC models in the context of software engineering, see, for example, [22].

2.1.1 Impact of the Software Development Life Cycle on Testing

Testing, to be effective, must be integrated with the project's SDLC model. The choice of the SDLC model affects the following testing issues:

- Scope and timing of test activities
- Selection and timing of test levels and test types
- Level of detail of the test documentation
- Selection of test techniques and practices
- Scope of test automation

Sequential SDLC Models

Sequential SDLC models assume the execution of individual development activities one after another, linearly. The consequence of adopting such a model of software development is that a given phase cannot begin until the previous phase is completed. In practice, it can sometimes happen that phases overlap. However, sequential models assume, at least in theory, that before the next phase begins, there is a moment of verification that all the activities of the preceding phase have been done correctly.

In the early phases of a project run in sequential models, the tester usually participates in static testing: requirement reviews and test design. The product in executable form is usually delivered in later phases, so most often, dynamic testing cannot be performed early in the development cycle.

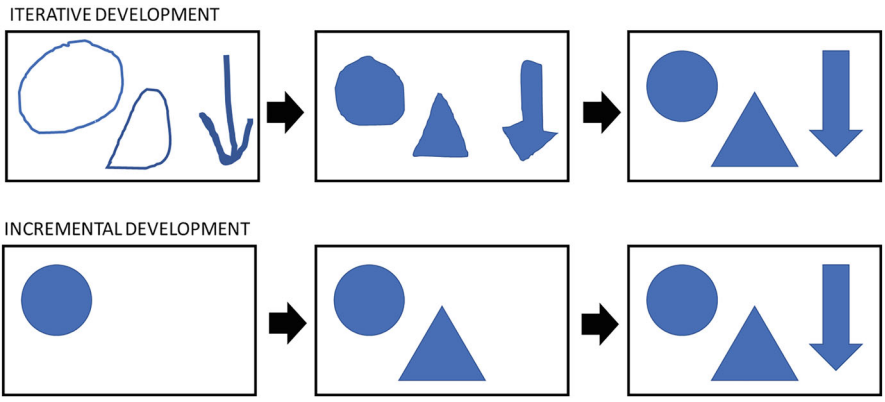

Fig. 2.1 Symbolic difference between iterative and incremental SDLC models

Examples of the most important sequential models are discussed below.

Iterative and Incremental SDLC Models

All iterative and incremental SDLC models are based on the same fundamental principle stating that software development is done in cycles. There are some differences between these two types of models. These are shown symbolically in Fig. 2.1.

Incremental software development is the process of specifying requirements and designing, building, and testing the system in parts, which means that software functionality grows incrementally. The size of individual increments of functionality depends on the specific model of the development cycle. Some models provide for the breakdown into larger chunks, while others provide for smaller chunks. A single increment of functionality may even be limited to a single change on a user interface screen or a new query option.

The bottom part of Fig. 2.1 shows this approach. Suppose our product is an image composed of three elements: a circle, a triangle, and an arrow. In the incremental model, we create pieces of this image one at a time, such as each figure in successive increments. Specifically, this means that incrementally added functionality should already have the form it will have in the final, target product. Of course, changes are inevitable, but this is the general principle that distinguishes incremental from iterative models. A single increment can be conducted in a sequential or iterative model.

Iterative development, on the other hand, involves specifying, designing, building, and testing together groups of functionalities in a series of cycles, often of a strict duration. Iterations may include changes to functionality produced in earlier iterations, together with changes in the scope of the project. Each iteration delivers working software that is an increasing subset of the total set of functionalities until the final software version is delivered or development stops.

The top part of Fig. 2.1 presents an iterative approach. We realize the construction of our image, so to speak, in sketches, each of which encompasses the whole of the

final idea, but this whole is presented at different levels of detail: first we have some initial drawing, an outline of the image concept. In subsequent iterations, we refine this concept, "fleshing out" more and more details. Of course, this approach does not necessarily apply to the entire product but, for example, to some highlighted group of functionalities, as described earlier.

In some iterative and incremental models, it is assumed that each iteration or increment ends with a working product. This means that in each iteration (increment), both static testing and dynamic testing can be performed at all test levels. Frequent delivery of working pieces of software requires rapid feedback and extensive regression testing.

Agile Practices
Agile software development methods assume that change can occur at any point in the project. This assumption leads these methods to favor lightweight documentation and extensive test automation to make regression testing, in particular, easier to perform. In addition, a large portion of manual testing is carried out using experience-based test techniques that do not require lengthy, thorough, prior planning (see Sect. 4.4).

We will now discuss some of the most important sequential and iterative SDLC models.

Sequential Models

Waterfall Model
In the waterfall model (see Fig. 2.2), software development activities (such as requirements analysis, design, code development, and testing) are performed one after another. According to the assumptions of this model, testing activities occur only after all other development activities have been completed. This is a problematic situation from the tester's point of view. In Chap. 1, we noted that testing should start as soon as possible, because the later this happens, the more expensive it will be to fix the defects found.

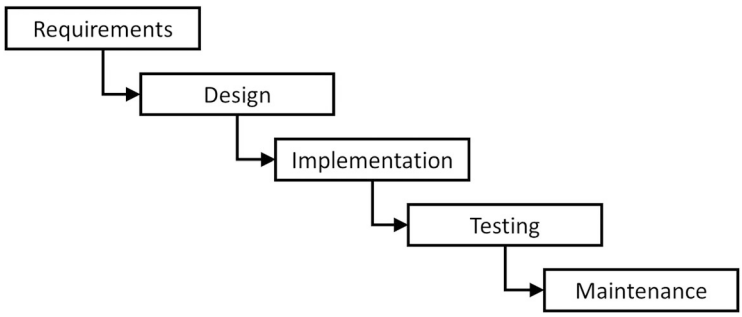

Fig. 2.2 Waterfall model

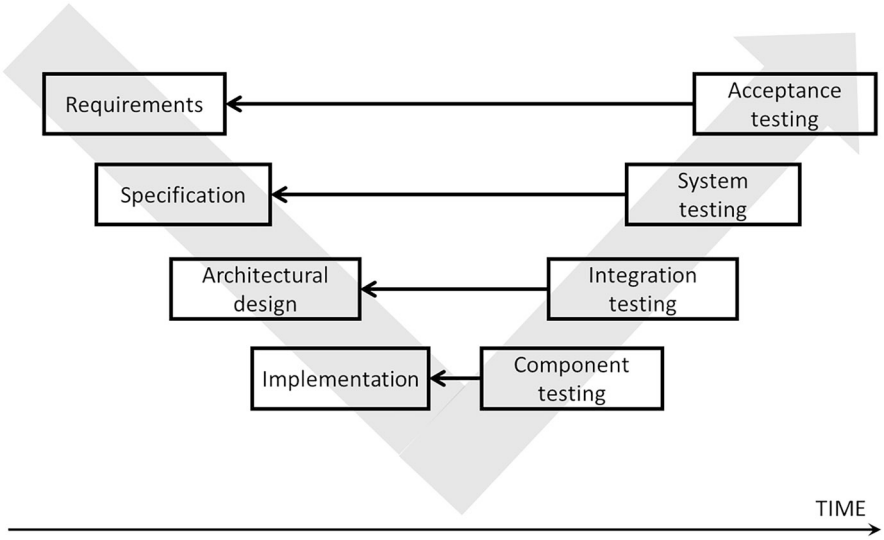

Fig. 2.3 V-model

The waterfall model prevents us from starting testing early. So what is the benefit of using this approach? Well, it works well in projects where the requirements are well-known and well developed, and the team can be sure (or almost sure) that they will not change during the development activities. The waterfall model is a good solution in typical, "batch" projects (e.g., implementation projects requiring more complex configuration) or in those where there are very well identified requirements and there is little or no risk of their changing (e.g., requirements resulting directly from legislation).

V-Model

The V-model is a modified waterfall model that attempts to address the drawbacks of the waterfall model, associated with late testing. The model (see Fig. 2.3) assumes the integration of the testing process into the software development process, thereby implementing the principle of early testing. In addition, the V model includes levels of testing linked to each corresponding phase of software development, which further promotes early testing (see Sect. 2.2). In this model, test execution associated with all test levels occurs sequentially, but in some cases, there may be overlapping phases.

The V-model is thus an attempt to address the problems posed by the waterfall model in the context of testing and quality assurance. It draws attention to the testing activities that occur in each phase—not only in the dynamic testing phase (the right branch of the model) but also in the development phases (the left branch of the model). In the latter, the testers are not always able to perform tests (e.g., in the design phase there is no testable product to run yet), but they can always review the test basis and design the related tests. A version of the V-model, the so-called

W-model, proposes that testing activities in the manufacturing phases should also involve static testing, i.e., that reviews of work products (e.g., requirements review, architecture review, code review, etc.) should be conducted.

Iterative and Incremental Models

Unified Process (UP) Model

The UP model (see Fig. 2.4) is an iterative and incremental model. In essence, UP is not a single process but a flexible *process framework* within which individual processes are defined, such as business modeling, requirements, analysis and design, implementation, testing, or deployment. These processes can be selected and adapted to the needs of the project. Individual iterations usually take a relatively long time (e.g., 2 or 3 months), and the incremental parts of the system are correspondingly large, covering, for example, two or three groups of related functionality.

Boehm's Spiral Model

Spiral model is an approach in which incremental experimental components are created that can then be extensively rebuilt or even abandoned at later stages of software development (see Fig. 2.5). Components or systems using the above models often include overlapping and repeated test levels in the software

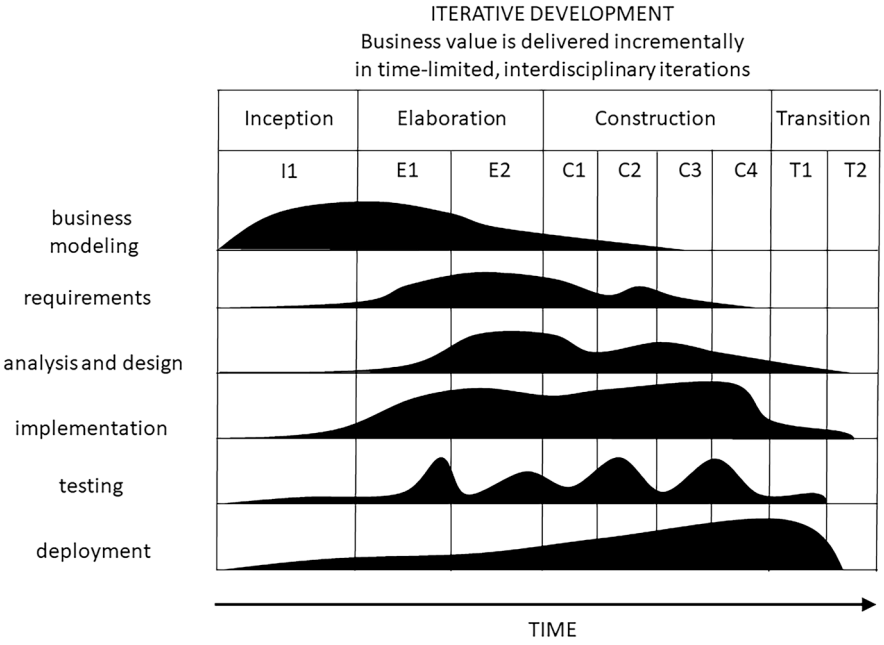

Fig. 2.4 Unified process

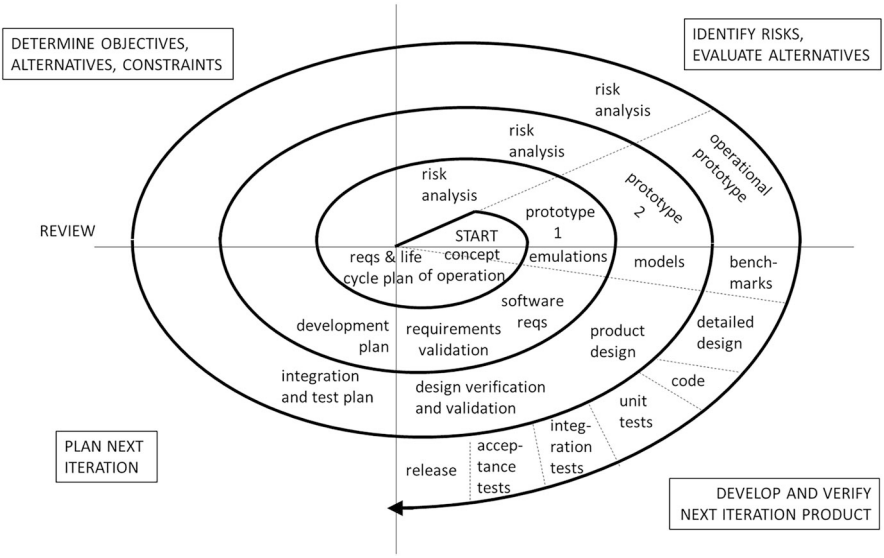

Fig. 2.5 Boehm's spiral model

development cycle. Ideally, each functionality is tested at multiple levels, approaching the final release. In some cases, teams use *continuous delivery* or *continuous deployment*, approaches that make extensive use of automation across multiple levels of testing as part of the software delivery pipeline. In addition, many such models incorporate the concept of self-organizing teams, which can change the way work is organized in relation to testing and the relationship between testers and developers.

The spiral model was proposed by Barry Boehm, and in Boehm's own words, it is essentially a "model of models," since it can be configured to obtain virtually any other model of the development cycle (e.g., by restricting it to the last quarter of the outer spiral, you get a waterfall model). Its characteristic and one of its most important features is the risk analysis performed before the start of each successive development cycle.

Prototyping Model

The prototyping model is often used where design goals are not strictly defined or specified in advance. For example, the customer defines a set of general goals for the software, but does not specify detailed functional requirements, the developer may be uncertain about the effectiveness of the implemented algorithm or the form of the program's interaction with the user, etc. In these types of situations, the prototyping model may offer the best approach.

Prototyping helps stakeholders better understand what is to be built when requirements are fuzzy. The process (see Fig. 2.6) begins with communication that defines the overall goals (requirements) for the software. The planning phase is followed by a series of rapid prototyping iterations. In each iteration, the team

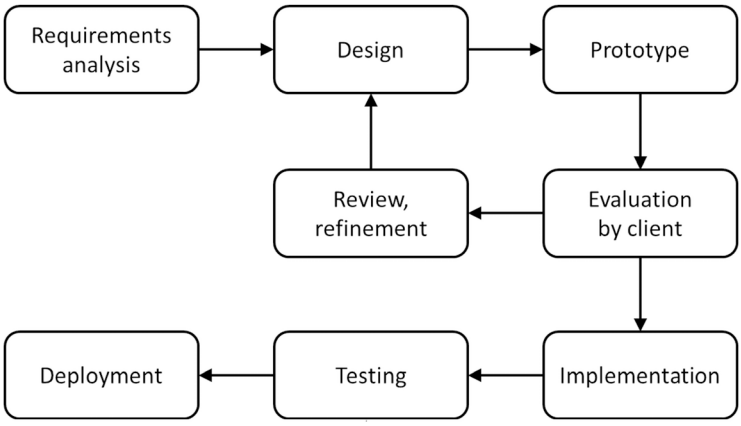

Fig. 2.6 Prototyping model

focuses on the representation of those aspects of the software that will be visible to end users and ends with the building of a working prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback. This information is used to further refine the requirements.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, existing program fragments can be used, or tools can be employed that enable rapid generation of working programs (e.g., "dynamic" GUI mock-ups).

Development Methodologies and Agile Practices

Scrum

Scrum (see Fig. 2.7) is currently one of the most common software development methods.[1] Scrum divides software development into short iterations of the same length (usually 1–4 weeks), and the iterative parts of the system are correspondingly small, that is, they include, for example, a few enhancements or new functionalities.

In a model like Scrum, testing becomes challenging due to short iterations and frequent changes. Therefore, instead of a thorough and lengthy test case design process, in Scrum, it is more common to see practices such as exploratory testing

[1] Often one can encounter the opinion that Scrum is not a method but a framework. Sometimes, too, the word "methodology" is used instead of "method." These misunderstandings are due to the common, unfortunately, failure to distinguish the meaning of the words: "method," "methodology," and "framework." A method is a set of rules for doing some work. Framework, on the other hand, is the arrangement and interrelationship of elements that make up a whole. Methodology, on the other hand, is the science that studies methods. Scrum is therefore a method, and against all appearances, it is quite prescriptive (i.e., imposing a certain way of doing things) about many elements of the model, such as the length of sprints or the duration of so-called daily stand-up meetings.

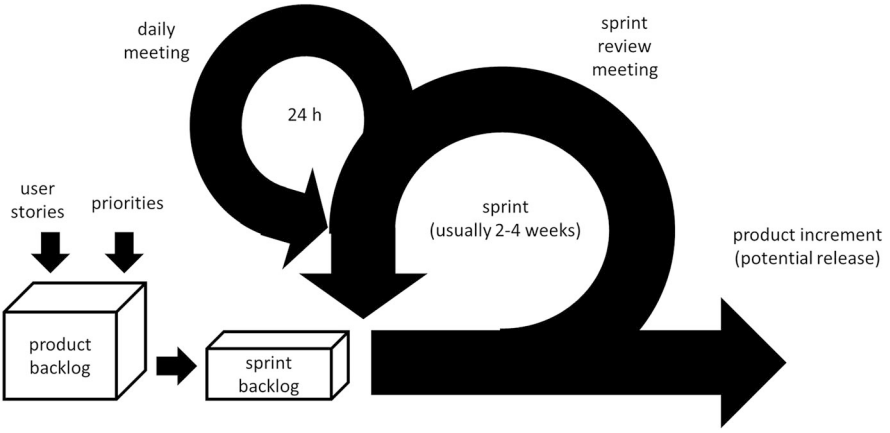

Fig. 2.7 Scrum

(see Sect. 4.4.2) or an emphasis on extensive test automation—this is especially true for regression tests, the number of which tends to grow rapidly with each iteration.

Scrum is a mix of a *push* and *pull* system, because before each iteration, the team selects ("push") from the product backlog to the sprint backlog a predetermined set of tasks to be completed in that iteration, and within a sprint, each team member starts working on the next task when the previous one is finished ("pull").

Kanban

Kanban makes it possible to deliver one enhancement or functionality at a time (immediately after preparation) or to group more functionality for simultaneous transfer to the production environment. Originally, the method was used in manufacturing companies such as Toyota, where products are developed in series. However, it can also be adapted to IT projects. Kanban uses a so-called *Kanban board* to visualize, plan, and supervise the development process (see Fig. 2.8).

The Kanban method is based on so-called product cards and organizes the development process in such a way that each production station produces exactly as much as is needed at any given moment. In addition, each workstation has a top-down limit of work it can do within a certain unit of time (the so-called Work in Process Limit—WIP). Such an organization of work makes it possible to run production efficiently, so that no unnecessary elements are created, and the "mass" of tasks to be performed "flows" smoothly through the production system, thereby optimizing resource consumption and production time.

Kanban is a *pull* system, because a worker at a given job "pulls" a task from the previous workstation where a product has already been completed and is ready to be passed on.

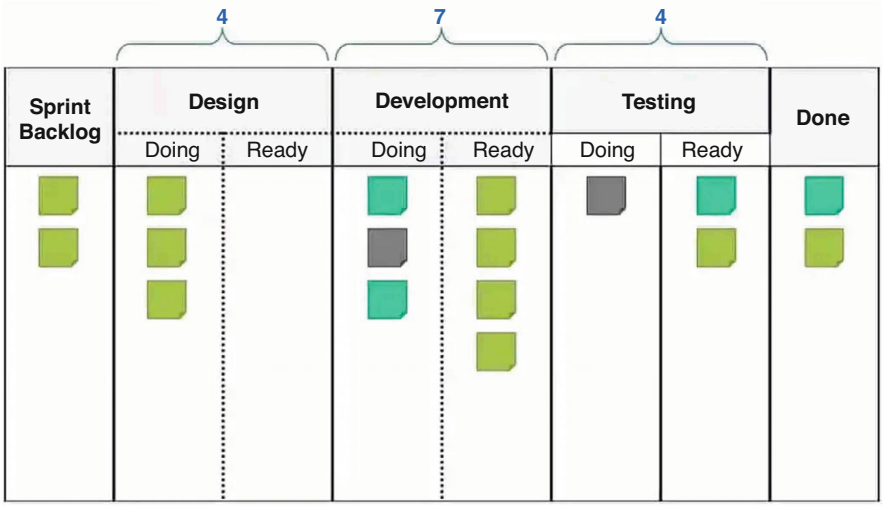

Fig. 2.8 Sample Kanban board (source: scrum.org)

Principles for Selecting a Software Development Life Cycle Model

SDLC models should be selected and tailored to the context resulting from the characteristics of the project and product. Accordingly, the following should be taken into account when selecting and adjusting the appropriate model:

- The purpose of the project
- The type of product being developed
- Business priorities (such as time-to-market)
- Identified product risks and project risks

For example, the development and testing of a minor internal administrative system should be done differently than the development and testing of a safety-critical system, such as a car's brake control system. As another example, in some cases, organizational and cultural issues can hinder communication among team members, which can result in slowing down software development in an iterative model.

There exist many SDLC models in the IT industry, and often choosing the right one can be difficult. However, one should keep in mind the aforementioned selection criteria. You should also be aware that the model is not an inviolable sanctity. It is only a proposal that tells us how to organize the software development process. Such a model can and even should be adapted to the needs and requirements of the organization in which the software is developed.

From the tester's point of view, the most relevant aspect of the SDLC will of course be the test process. Depending on the project context, it may be necessary to combine or reorganize some test levels and/or test activities. For example, in the case of integration of *commercial off-the-shelf* (COTS) software with a larger system, the buyer may perform interoperability testing at the system integration test level (e.g.,

for integration with infrastructure and other systems) and at the acceptance test level (functional testing and nonfunctional testing along with user acceptance testing and operational acceptance testing). The test levels and types are described in detail in Sects. 2.2.1 and 2.2.2.

In addition, different SDLC models can be combined. An example would be using the V model for development and testing of the back-end part of the system and the agile model for development and testing of the front-end (user interface). Another variant is to use a prototyping model early in the project and then replace it with an incremental model after the experimental phase.

For systems related to the *Internet of Things* (IoT), which consist of many different objects—such as devices, products, and services—separate SDLC models are usually applied to individual parts of the system. This poses a major challenge especially for the development of individual versions of IoT systems. In addition, in the case of the above objects, more emphasis is placed on the later stages of the SDLC, after the objects are already in operation (e.g., the production, upgrade, and decommissioning phases).

The process of selecting the right SDLC model can proceed as follows. In the first step, we define the criteria by which we will evaluate the suitability of each model in our project. Examples of criteria that can be considered are:

- Size and experience of the team
- Size, type and level of complexity of the project
- Customer relations, stability of requirements, frequency of their changes
- Geographic dispersion of the team
- Model's compatibility with the company's business/organizational strategy

In step two, we compare potential SDLC models, considering both their advantages and disadvantages. Keep in mind that there are no perfect or universal solutions. Each SDLC model will have some positive sides but will also present us with some challenges. For example, using agile models, especially in a team previously using sequential models, requires changing the thinking of team members, which is often a very difficult task.

In step three, we assess the needs of our organization. The SDLC model should reflect the nature and operations of the company or team we are working for. Producing software for the aerospace, medical, or military industries will require completely different procedures and approaches than, for example, creating a game for a mobile device in a just-formed small start-up company.

In the final step, we apply the previously selected criteria in the context of our organization's needs.

2.1.2 Software Development Life Cycle and Good Testing Practices

Knowledge of SDLC models is important from the tester's point of view, because which selection of the model is adopted in the development process affects when and how testing activities will be performed. In Sect. 2.1.1, we described a number of example SDLC models. However, regardless of which model a tester will work in, there are several principles of good testing practices that should be followed:

- **For every software development activity, there is a corresponding test activity**. This principle draws attention to the "totality" of testing activities: every work product that was produced during software development should also be subject to quality control.
- **Each testing level corresponds to testing objectives appropriate to the phase or group of activities within the software development cycle**. This principle draws attention to the fact that testing objectives may differ significantly (see Sect. 1.1); at one level, we will be mainly interested in detecting as many defects as possible, while at another level, we will be more interested in validating that the system meets the customer's requirements and needs.
- **Test analysis and test design for a given test level should be started already during the execution of the corresponding software development activity**. This rule is related to the principle of early testing (Sect. 1.3), which assumes that testing activities start as early as possible in order to detect problems as soon as possible. Defects are usually cheap to fix in the early phases and often very expensive in later phases. Note that sometimes defects are found not through physical testing of the application but through test design activities (see Chap. 4).
- **Testers should participate in reviews of work products (e.g., requirements, design, or user stories) as soon as draft versions of the relevant documents are available**. This principle highlights the role of the tester as a specialist in software quality control and is an example of the implementation of the "shift-left" principle (see Sect. 2.1.5). During such reviews, testers very often are able to find a lot of ambiguities, errors, or contradictions, which—if unnoticed—could result in serious problems in the future.

Example An organization on behalf of the Ministry of Justice is developing the product JudgeLottery, a system for random assignment of cases to judges. Table 2.1 collects examples of development activities and corresponding test activities.

2.1.3 Testing as a Driver for Software Development

Test-Driven Development (TDD), *Acceptance Test-Driven Development* (ATDD), and *Behavior-Driven Development* (BDD) are three popular software development methods based on the *test-first approach*. This approach assumes that *before* the

Table 2.1 Examples of manufacturing activities and corresponding testing activities

Phase	Manufacturing activity	Tester activity
Requirements	Gathering system requirements, creating requirements documentation	Inspection of requirements for testability and their compliance with business requirements, such as the requirements of the Law on General Courts
Design	Database design	Inspection of the database design for compliance with system requirements
Implementation	Implementation of the front-end (web-based application)	Usability testing, system-database integration testing, system testing
Deployment	Installation of the system in the ministry	Acceptance (beta) testing of the system in the target environment by court presidents

source code that implements a specific function is written, a test for that function is created. It is an example of how testing can be a guiding factor in software development.

The "test first" approach originates from the extreme programming (XP) and is one of the core practices of agile software development. This approach may seem to be counterintuitive (we usually create something first and test it later), but it has some important benefits [23]:

- **Testing replaces the trial-and-error method**. Instead of, for example, checking by trial-and-error whether an implemented function works correctly, a set of meaningful test cases is created with strictly defined input and output values, and then code is written to pass these tests.
- **Test cases provide objective feedback on progress**. Each test that passed is an objective evidence that the project is moving forward.
- **Test cases replace specifications**. Since tests are written before code is created, they are not just a collection of control procedures but also define sample behavior of the test object, becoming an example of "living documentation" [24]. A change in requirements requires a change in tests, so it forces a change in documentation, making the documentation relevant at any point in the project.
- **The "test-first" approach improves the quality of public interfaces** (APIs). Component testing and component integration testing use the public methods of the classes under test, and since the tests are written before the code is created, the tests define the names of the public methods of the class under test, their parameters, and examples of method usage. The design of the tests practically becomes the definition of the interface.
- **The "test-first" approach improves the testability of the code being developed**. Once the tests are implemented, the developer must then write code that passes them. This implies that the code must invoke the interfaces required by the tests. So instead of writing tests that check existing code, the developer must write code that is "compatible" with the tests that were written earlier. At the same time, because the code is written to pass specific tests, the programmer has better control over and more easily achieves a higher level of code coverage with tests.

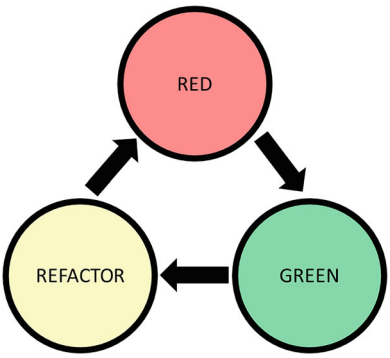

Fig. 2.9 Test-driven development process

The main difference between TDD, BDD, and ADD is that TDD focuses on component testing, BDD focuses on system behavior testing, and ATDD focuses on system acceptance testing. Thus, it can be said that TDD works at the component and component integration test levels, BDD works at the system and system integration test levels, and ATDD works at the acceptance test level (see Sect. 2.2.1). The resulting test cases should be suitable for use in automated regression testing. Each of these approaches implements the testing principle "Early testing saves time and money" (see Sect. 1.3) and follows the "shift-left" approach (see Sect. 2.1.5) in that tests are defined before the code is written.

Test-Driven Development (TDD)
TDD uses automated component test cases to guide code development and is usually performed by a developer. The TDD process goes as follows:

- The developer creates a new test case, corresponding to a specific requirement for the code.
- All existing tests are run. The new test should fail because there is no corresponding code yet (the "RED" step in Fig. 2.9). The first run of the new test is to check whether the test compiles at all. In turn, if the test passes on the first run, it most likely means that there is a problem with the test itself, because it should fail, since there is no corresponding code written yet. Running the remaining tests acts as regression tests (see Sect. 2.2.3).
- The developer writes a minimum amount of code sufficient to pass the new test. All previously written tests must also pass (the "GREEN" step in Fig. 2.9).
- The developer refactors the code (if necessary) to keep the quality of the code high, since TDD takes place in many short "test-code" cycles, which can degrade the quality of the code itself (the "REFACTOR" step in Fig. 2.9). After refactoring, the developer re-runs all tests to make sure the refactoring didn't break anything.
- The above steps are repeated as long as there is still some code left to write.

The TDD approach typically uses a test framework like xUnit to support automated testing. A single "test-code" cycle is often very short and can take even less than 1 or 2 min.

Using the TDD approach, developers gain a certain independence of testing. They do not have an emotional connection to the code, because they have not created it yet. Thus, through test design alone, the developers can test or establish implementation assumptions for a component. Their tests will therefore be stronger than if they were to be written *after the component has been* implemented.

Behavior-Driven Development (BDD)
In TDD (above), tests can even refer to a few individual lines of code. In BDD, behavior-driven development, tests focus on the expected behavior of the system [25], so they operate at a slightly higher level than the component tests in the TDD approach. BDD uses a collaborative approach to generate acceptance criteria in plain language, usually as part of a user story that can be understood by all stakeholders (see Sect. 4.5).

Acceptance criteria are usually written using frameworks. The typical format for acceptance criteria is Given/When/Then. BDD can be automated. The framework, based on the user story and associated acceptance criteria, automatically generates test case code and executes it.

The principle of the BDD framework is shown below. The test can be recorded in the form of a readable and understandable text document written in language like Gherkin (the test along with the code below is a slightly modified example from https://automationrhapsody.com/introduction-to-cucumber-and-bdd-with-examples/):

```
Feature: search for an entry in Wikipedia
Scenario: direct article search
Given Input search term 'testing'
When Click search
Then Page containing the phrase 'testing' appears
```

This test verifies that if a user types the search term "Testing" into Wikipedia's search engine, when the user clicks the "Search" button, the system will return a page containing the text "testing." These test case steps are physically implemented by code using the BDD framework. A fragment of such code is shown in the following listing. In the line annotated with `@Given`, the developer tells the Gherkin script above to search for the phrase "Enter search term X" in the line starting with the word "Given," where the value of X is automatically assigned to the `searchTerm` variable that is a parameter of the `searchFor` method associated with the "Given" line. The framework identifies this snippet through the use of so-called regular expression, which defines a pattern and finds the string that matches it.

If the string is matched with a pattern, the `searchFor` method is run with the X parameter (in our case—the string "testing"). It physically does what it needs to do—it searches for a text search box on the web page and sends the value of the X variable (i.e., the word "testing") to it, using the corresponding statements of the Selenium WebDriver library, a tool for automatic web page testing. Similarly, the

methods associated with the `@When` and `@Then` annotations perform the corresponding actions: clicking the "Search" button and verifying that the specified text appears on the page.

```
package com.automationrhapsody.cucumber.parallel.tests.wikipedia;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

import cucumber.api.java.After;
import cucumber.api.java.Before;
import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;
import cucumber.api.java.en.When;

import static junit.framework.Assert.assertTrue;
import static junit.framework.TestCase.assertFalse;
import static org.junit.Assert.assertEquals;

public class WikipediaSteps {
 private WebDriver driver;

 @Before
 public void before() {
     driver = new FirefoxDriver();
     driver.navigate().to("http://en.wikipedia.org");
 }

 @After
 public void after() {
     driver.quit();
 }

 @Given("^Input search term '(.*?)'$")
 public void searchFor(String searchTerm) {
  WebElement searchField = driver.findElement(By.id("searchInput"));
  searchField.sendKeys(searchTerm);
 }

 @When("^Click search$")
 public void clickSearchButton() {
  WebElement searchButton = driver.findElement(By.id
("searchButton"));
  searchButton.click();
 }
```

```
 @Then("^Page containing the phrase '(.*?)' appears$")
 public void assertSingleResult(String searchResult) {
  WebElement results = driver
   .findElement(By.cssSelector("div#mw-content-text.mw-content-ltr
     p"));
  assertFalse(results.getText().contains(searchResult + "may
     refer to:"));
  assertTrue(results.getText().startsWith(searchResult));
      }
}
```

BDD follows the concept of "living documentation" as the specification is executable in automated testing. The specification then usually follows the principle of "Specification By Example" [24], which describes the requirement by typical (testable) examples rather than explaining all possible needs and exceptions.

Acceptance Test-Driven Development (ATDD)
ATDD follows the same procedure that is defined for TDD and BDD, but with automated test cases that are at the appropriate level for acceptance testing [26]. These acceptance test cases are derived from acceptance criteria that are generated jointly by developers, testers, and business representatives. Acceptance criteria are written from the user's perspective and are usually in an easy-to-understand format (e.g., Given/When/Then known from the BDD approach). ATDD is explained in detail later (see Sect. 4.5).

The implementation scheme of ATDD (see Fig. 2.10) is as follows:

- Select a user story
- Write an acceptance test
- Implement a user story (coding)
- Run an acceptance test
- Refactor code, if necessary
- Get user story approval (sign-off)

ATDD typically uses an automated acceptance testing framework (such as FitNesse) to support automated acceptance testing. ATDD, like BDD, is guided by the concept of "living documentation."

2.1.4 DevOps and Testing

Traditionally, the areas of software development, testing, and operations remained in separate silos, with completely different priorities. Development focused on creating and delivering software, testing was concerned with controlling the quality of the software produced, and operations focused on implementing and supporting the software. The separation of these three areas often caused conflicts between them.

DevOps is an approach to create synergy by having development, testing, and operations work together to achieve common goals (see Fig. 2.11). DevOps requires

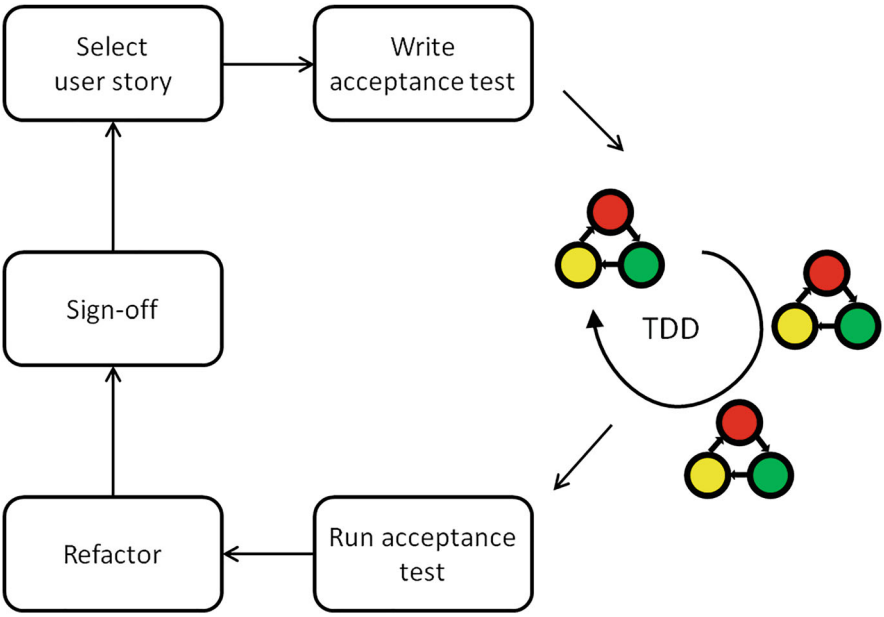

Fig. 2.10 Acceptance test-driven development process

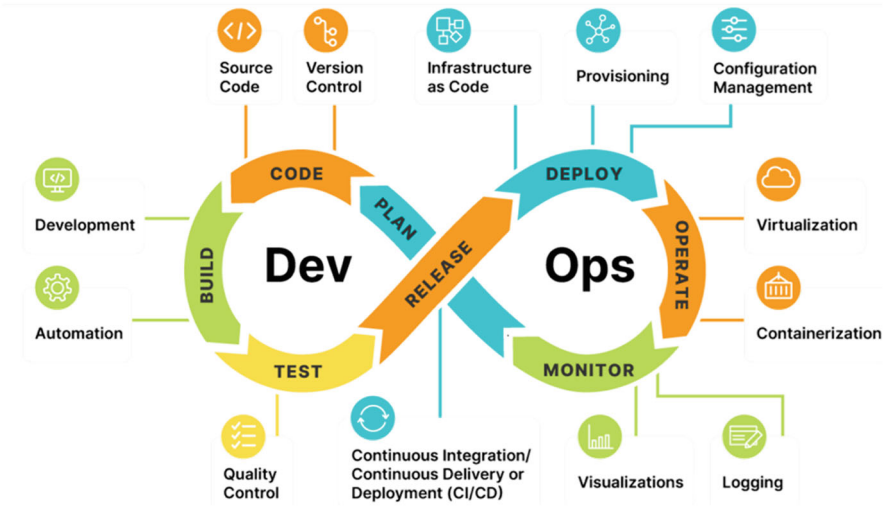

Fig. 2.11 DevOps deployment process (source: orangematter.solarwinds.com)